Serge Haddad
Lucia Pomello (Eds.)

LNCS 7347

Application and Theory of Petri Nets

33rd International Conference, PETRI NETS 2012
Hamburg, Germany, June 2012
Proceedings

Springer

Serge Haddad   Lucia Pomello (Eds.)

Application and Theory of Petri Nets

33rd International Conference, PETRI NETS 2012
Hamburg, Germany, June 25-29, 2012
Proceedings

Springer

Volume Editors

Serge Haddad
Ecole Normale Supérieure de Cachan
Laboratoire Spécification et Vérification
61, avenue du Président Wilson, 94235 Cachan, France
E-mail: haddad@lsv.ens-cachan.fr

Lucia Pomello
Università degli Studi di Milano–Bicocca
Dipartimento di Informatica, Sistemistica e Comunicazione
Viale Sarca, 336, 20126 Milano, Italy
E-mail: pomello@disco.unimib.it

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-31130-7 e-ISBN 978-3-642-31131-4
DOI 10.1007/978-3-642-31131-4
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2012939390

CR Subject Classification (1998): D.2.2, D.2.4, F.1, F.3, G.3, J.1, E.1

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

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

This volume constitutes the proceedings of the 33rd International Conference on Application and Theory of Petri Nets and Concurrency (PETRI NETS 2012). The Petri Net conferences serve as annual meeting places to discuss the progress in the field of Petri nets and related models of concurrency. They provide a forum for researchers to present and discuss both applications and theoretical developments in this area. Novel tools and substantial enhancements to existing tools can also be presented. The satellite program of the conference comprised five workshops, a Petri net course including tutorials and a model checking contest. PETRI NETS 2012 was co-located with the 12th International Conference on Application of Concurrency to System Design (ACSD 2012). The two conferences shared five invited speakers. The PETRI NETS 2012 conference was organized by the University of Hamburg. It took place in Hamburg, Germany, during June 25-29, 2012. We would like to express our deepest thanks to the Organizing Committee chaired by Daniel Moldt (Germany) for all the time and effort invested in the local organization of the conference.

This year the number of submitted papers amounted to 55, which included 48 full papers and 7 tool papers. The authors of the papers represented 25 different countries. We thank all the authors who submitted papers. Each paper was reviewed by at least four referees. The Program Committee (PC) meeting took place electronically, using the EasyChair conference system for the paper selection process. The PC selected 21 papers: 18 regular papers and 3 tool papers for presentation. After the conference, some authors were invited to publish an extended version of their contribution in the *Fundamenta Informaticae* journal. We thank the PC members and other reviewers for their careful and timely evaluation of the submissions before the meeting, and the fruitful discussions during the electronic meeting. Finally, we are grateful to the invited speakers for their contribution: Tony Hoare, Alain Finkel, Bart Jacobs, Joost-Pieter Katoen and Jens Spars. The Springer LNCS Team and the EasyChair system provided high-quality support in the preparation of this volume.

April 2012

Serge Haddad
Lucia Pomello

Organization

Steering Committee

W. van der Aalst, The Netherlands	M. Koutny, UK (Chair)
J. Billington, Australia	C. Lin, China
G. Ciardo, USA	W. Penczek, Poland
J. Desel, Germany	L. Pomello, Italy
S. Donatelli, Italy	W. Reisig, Germany
S. Haddad, France	G. Rozenberg, The Netherlands
K. Hiraishi, Japan	M. Silva, Spain
K. Jensen, Denmark	A. Valmari, Finland
J. Kleijn, The Netherlands	A. Yakovlev, UK

Program Committee

Gianfranco Balbo	University of Turin, Italy
Marek Bednarczyk	Institute of Computer Science, PAS, Poland
Jonathan Billington	University of South Australia, Australia
Josep Carmona	Universitat Politècnica Catalunya, Spain
Gianfranco Ciardo	University of California at Riverside, USA
José-Manuel Colom	University of Zaragoza, Spain
Jörg Desel	Fernuniversität in Hagen, Germany
Raymond Devillers	ULB, Bruxelles, Belgium
Javier Esparza	Technische Universität München, Germany
Dirk Fahland	Technische Universiteit Eindhoven, The Netherlands
Alessandro Giua	DIEE - University of Cagliari, Italy
Luis Gomes	Universidade Nova Lisboa / UNINOVA, Portugal
Serge Haddad (Co-chair)	LSV, ENS Cachan, France
Kunihiko Hiraishi	Japan Advanced Institute of Science and Technology
Ryszard Janicki	McMaster University, Canada
Victor Khomenko	Newcastle University, UK
Ekkart Kindler	DTU, Denmark
Jetty Kleijn	LIACS, Leiden University, The Netherlands
Lars Michael Kristensen	Bergen University College, Norway
Ranko Lazic	University of Warwick, UK
Chuang Lin	Tsingnhua University, China
Andrew Miner	Iowa State University, USA

Madhavan Mukund	Chennai Mathematical Institute, India
Laure Petrucci	Université Paris 13, France
Lucia Pomello (Co-chair)	Università di Milano-Bicocca, Italy
Wolfgang Reisig	Humboldt-Universitaet zu Berlin, Germany
Satoshi Taoka	Hiroshima University, Japan
Rüdiger Valk	University of Hamburg, Germany
Antti Valmari	Tampere University of Technology, Finland

Organizing Committee

Sofiane Bendoukha, Germany
Tobias Betz, Germany
Lawrence Cabac, Germany
Michael Duvigneau, Germany
Erik Flick, Germany
Frank Heitmann, Germany
Marcin Hewelt, Germany
Michael Köhler-Bußmeier, Germany
Manfred Kudlek, Germany
Daniel Moldt (Chair), Germany
Felix Ortmann, Germany
Rüdiger Valk, Germany
Thomas Wagner, Germany
Margit Wichmann, Germany

Workshops and Tutorials Chairs

A. Yakovlev, UK
W. van der Aalst, The Netherlands

Tools Exhibition Chair

L. Cabac, Germany

Publicity Chairs

M. Duvigneau, Germany T. Wagner, Germany

Additional Reviewers

Alekseyev, Arseniy	Beccuti, Marco	Bonsangue, Marcello
André, Étienne	Betz, Tobias	Bremehr, Tall
Barros, Joao Paulo	Bonnet, Remi	Böttcher, Sven

Cabac, Lawrence
Choppy, Christine
Clarisó, Robert
Costa, Aniko
D'Souza, Deepak
De Pierro, Massimiliano
Donatelli, Susanna
Dong, Yangwei
Down, Doug
Duvigneau, Michael
Evangelista, Sami
Fernandes, João M.
Flick, Nils Eric
Framceschinis, Giuliana
Franceschelli, Mauro
Gallasch, Guy
Geeraerts, Gilles
Gierds, Christian
Gupta, Amar
Hansen, Henri
Heitmann, Frank
Hewelt, Marcin
Hoogeboom, Hendrik Jan
Huang, Jiwei
Jaskolka, Jason
Jimenez, Emilio
Jin, Xiaoqing
Julvez, Jorge
Jóźwiak, Piotr
Karandikar, Prateek
Klai, Kais
Kosters, Walter
Kretinsky, Jan
Lembachar, Yousra
Li, Yin
Liu, Lin
Lodaya, Kamal
López-Grao, Juan-Pablo
Machado, Ricardo
Madalinski, Agnes
Mahulea, Cristian
Meng, Kun
Merseguer, Jose
Miyamoto, Toshiyuki
Mokhov, Andrey
Mumme, Malcolm
Munoz-Gama, Jorge
Müller, Richard
Ohta, Atsushi
Ouyang, Chun
Pawłowski, Wiesław
Poliakov, Ivan
Prüfer, Robert
Ribeiro, Oscar
Rosa-Velardo, Fernando
Schlund, Maximilian
Schwarick, Martin
Seatzu, Carla
Simonsen, Kent Inge Fagerland
Soltys, Michael
Solé, Marc
Sproston, Jeremy
Stahl, Christian
Steggles, Jason
Sürmeli, Jan
Takahashi, Koji
Tian, Yuan
Trivedi, Ashutosh
Tsuji, Kohkichi
Turner, Benjamin
Wagner, Christoph
Wagner, Thomas
Wan, Jianxiong
Wester-Ebbinghaus, Matthias
Westergaard, Michael
Wolf, Karsten
Xiang, Xudong
Yamaguchi, Shin'Nosuke
Yamaguchi, Shingo
Yao, Min
Yin, Xiang
Yoneda, Tomohiro
Zhao, Yang
Zubkova, Nadezhda

Table of Contents

Invited Papers

Net Models for Concurrent Object Behaviour 1
Tony Hoare

The Theory of WSTS: The Case of Complete WSTS 3
Alain Finkel and Jean Goubault-Larrecq

Regular Papers

Using Colored Petri Nets to Construct Coalescent Hidden Markov Models: Automatic Translation from Demographic Specifications to Efficient Inference Methods 32
Thomas Mailund, Anders E. Halager, and Michael Westergaard

An SMT-Based Discovery Algorithm for C-Nets 51
Marc Solé and Josep Carmona

Decomposing Process Mining Problems Using Passages 72
Wil M.P. van der Aalst

Complexity of the Soundness Problem of Bounded Workflow Nets 92
Guan Jun Liu, Jun Sun, Yang Liu, and Jin Song Dong

Cost Soundness for Priced Resource-Constrained Workflow Nets 108
María Martos-Salgado and Fernando Rosa-Velardo

On the α-Reconstructibility of Workflow Nets 128
Eric Badouel

On Profiles and Footprints – Relational Semantics for Petri Nets 148
Matthias Weidlich and Jan Martijn van der Werf

Data and Abstraction for Scenario-Based Modeling with Petri Nets 168
Dirk Fahland and Robert Prüfer

Maximal Confluent Processes 188
Xu Wang

Old and New Algorithms for Minimal Coverability Sets 208
Antti Valmari and Henri Hansen

Stubborn Sets for Simple Linear Time Properties 228
Andreas Lehmann, Niels Lohmann, and Karsten Wolf

Hybrid On-the-Fly LTL Model Checking with the Sweep-Line Method 248
Sami Evangelista and Lars Michael Kristensen

Safety Slicing Petri Nets 268
Astrid Rakow

Capacity Planning for Vertical Search Engines: An Approach Based on Coloured Petri Nets 288
Veronica Gil-Costa, Jair Lobos, Alonso Inostrosa-Psijas, and Mauricio Marin

An Infrastructure for Cost-Effective Testing of Operational Support Algorithms Based on Colored Petri Nets 308
Joyce Nakatumba, Michael Westergaard, and Wil M.P. van der Aalst

Designing Weakly Terminating ROS Systems 328
Debjyoti Bera, Kees M. van Hee, and Jan Martijn van der Werf

Nets-within-Nets to Model Innovative Space System Architectures 348
Frédéric Cristini and Catherine Tessier

P- and T-Systems in the Nets-within-Nets-Formalism 368
Frank Heitmann and Michael Köhler-Bußmeier

Tool Papers

Stochastic Modeling and Analysis Using QPME: Queueing Petri Net Modeling Environment v2.0 388
Simon Spinner, Samuel Kounev, and Philipp Meier

Snoopy – A Unifying Petri Net Tool 398
Monika Heiner, Mostafa Herajy, Fei Liu, Christian Rohr, and Martin Schwarick

CPN Assistant II: A Tool for Management of Networked Simulations ... 408
Štefan Korečko, Ján Marcinčin, and Viliam Slodičák

Author Index 419

Net Models for Concurrent Object Behaviour

Tony Hoare

Principal Researcher, Microsoft Research Ltd. Ave., Cambridge CB3 0FB

Summary

The behaviour of an object allocated and used by a computer program consists of a set of events involving the object which occur in and around a computer during execution of the program. Object behaviour can be modelled by an occurrence net (a Petri net without places), in which each event is a transition (drawn as a box), and the arrows between the transitions represent dependency between the events. The total behaviour of the program is just the sum of the behaviours of the objects which it allocates. A program (perhaps expressed as a Petri net with places) is mathematically defined as just the set of all its possible behaviours, in all its possible environments of execution. An object class is similarly defined as the set of all the possible behaviours of all its possible objects, as used in any possible program.

Events occurring in a program execution can be classified according to the time and place at which they occur. We draw space boundaries in a net as straight horizontal lines and time boundaries as (possibly skewed) vertical lines. Obviously, space boundaries cannot cross other space boundaries, and a similar constraint applies to time boundaries. Space and time boundaries cross each other orthogonally. No dependency can cross a time boundary backwards. While preserving these constraints, a skewed time boundary can be drawn straight by stretching the net horizontally.

We distinguish a class of local dependency arrows to denote a causal relationship between events occurring at the same place. A local arrow is drawn horizontally, and represents ownership of an object by a particular thread. This representation is defined formally by a rule that no local arrow can cross a space boundary. This constraint does not apply to non-local arrows, which are drawn vertically. They usually represent communication between threads, but they can represent communication within the same thread. Transmission of object ownership has to be accomplished by vertical arrows. In this way occurrence nets provide a simple model for Concurrent Separation Logic [1].

We permit the introduction of labels for the events and transitions of an occurrence net. The set of possible labels is part of the definition of the object class. An event is usually labelled by the syntactic format of the command whose execution gave rise to the event. It may also have labels identifying the values and the names or addresses of the objects involved, and the thread in which the event occurred. An arrow may be labelled by the value which is transmitted, and the identity of the variable or channel involved in the communication. A horizontal arrow may be labelled by its serial number within the horizontal chain to which it belongs.

These modelling conventions are illustrated by application to various kinds of object: semaphores, local variables, volatile variables and channels. Several variations of each concept are explored, for example, channels which are synchronous, asynchronous or finitely buffered. We also give examples of weaker classes of object, such as lossy or re-ordering channels, and the weak memory of modern multicore chips. In several cases, the definition of the whole set of behaviours of an object class is presented as a Petri net.

The behaviour of a complete program, just like any other object, is modelled as an occurrence net. Its atomic actions are modelled as boxes containing just one event from each object involved in the action. Groups of boxes in an occurrence net may be surrounded by a larger box, giving a structure of nested boxes to the given execution. Each such box in an occurrence net is attributed to the compound command in the program whose execution it records. If the program is a Petri net with boxes, as in the Box Algebra [2] for Petri nets, each box in the occurrence net is one of the executions of the corresponding box in the Petri net.

This net model of program behaviour can be proved (with the aid of pictures) to be a model of the Laws of Programming [3], extended by a new exchange law which relates concurrency to sequential composition [4]. An operational semantics (Milner style) and a deductive semantics (Hoare style) can be derived algebraically from the Laws [5]. Consequently, the net model is a valid model for both kinds of semantics, and no further demonstration is needed of the mutual consistency of proofs with the implementation of a programming language.

References

1. Brookes, S.D.: A Semantics for Concurrent Separation Logic. Theoretical Computer Science 375(1-3), 227–270 (2007)
2. Best, E., Devillers, R., Koutny, M.: The Box Algebra = Petri nets + Process Expressions. Information and Computation 178, 44–100 (2002)
3. Hoare, C.A.R., et al.: Laws of Programming. Commun. ACM 30(8), 672–686 (1987)
4. Hoare, C.A.R., Wehrman, I., O'Hearn, P.: Graphical Models of Separation Logic. In: Engineering Methods and Tools for Software Safety and Security, pp. 177–202. IOS Press (2009)
5. Hoare, T., van Staden, S.: In Praise of Algebra, under consideration for publication in Formal Aspects of Computing

The Theory of WSTS: The Case of Complete WSTS[⋆]

Alain Finkel and Jean Goubault-Larrecq

ENS Cachan
{finkel,goubault}@lsv.ens-cachan.fr

Abstract. We describe a simple, conceptual forward analysis procedure for ∞-complete WSTS $\mathfrak{S}$. This computes the so-called *clover* of a state. When $\mathfrak{S}$ is the completion of a WSTS $\mathfrak{X}$, the clover in $\mathfrak{S}$ is a finite description of the downward closure of the reachability set. We show that such completions are ∞-complete exactly when $\mathfrak{X}$ is an ω^2-*WSTS*, a new robust class of WSTS. We show that our procedure terminates in more cases than the generalized Karp-Miller procedure on extensions of Petri nets. We characterize the WSTS where our procedure terminates as those that are *clover-flattable*. Finally, we apply this to well-structured Presburger counter systems.

1 Introduction

Context. Well-structured transition systems (WSTS) [Fin87, Fin90, FS01, AČJT00] are a general class of infinite-state systems where coverability—given states s, t, decide whether $s \geq s_1 \rightarrow^* t_1 \geq t$ for some s_1, t_1—is decidable, using a simple algorithm that works backwards. The starting point of the series of papers entitled *Forward analysis for WSTS, part I: Completions* [FG09a], and *Forward analysis for WSTS, part II: Complete WSTS* [FG09b] Simplis our desire to derive similar algorithms working *forwards*, namely algorithms computing the *cover* $\downarrow Post^*(\downarrow s)$ of s. While the cover allows one to decide coverability as well, by testing whether $t \in \downarrow Post^*(\downarrow s)$, it can also be used to decide the boundedness problem, i.e., to decide whether the reachability set, $Post^*(s)$, is finite. No backward algorithm can decide this. In fact, boundedness is undecidable in general, e.g., on reset Petri nets [DFS98]. So the reader should be warned that computing the cover is not possible for general WSTS. Despite this, the known forward algorithms are felt to be more efficient than backward procedures in general: e.g., for lossy channel systems, although the backward procedure always terminates, only a (necessarily non-terminating) forward procedure is implemented in the TREX tool [ABJ98]. Another argument in favor of forward procedures is the following: for depth-bounded processes, a fragment of the π-calculus, the backward algorithm of [AČJT00] is not applicable when the maximal depth of configurations is not known

⋆ This paper is an extended abstract of a complete paper that will appear in the journal LMCS [FG12b]. A short version has already appeared in [FG09b] at ICALP'09.

in advance because, in this case, the predecessor configurations are not effectively computable [WZH10]. But the *forward* Expand, Enlarge and Check algorithm of [GRvB07], which operates on complete WSTS, solves coverability even though the depth of the process is not known a priori [WZH10].

State of the Art. Karp and Miller [KM69] proposed an algorithm, for Petri nets, which computes a finite representation of the *cover*, i.e., of the downward closure of the reachability set of a Petri net. Finkel [Fin87, Fin90] introduced the framework of WSTS and generalized the Karp-Miller procedure to a class of WSTS. This was achieved by building a non-effective completion of the set of states, and replacing ω-accelerations of increasing sequences of states (in Petri nets) by least upper bounds. In [EN98, Fin90] a variant of this generalization of the Karp-Miller procedure was studied; but no guarantee was given that the cover could be represented finitely. In fact, no effective finite representations of downward-closed sets were given in [Fin90]. Finkel [Fin93] modified the Karp-Miller algorithm to reduce the size of the intermediate computed trees. Geeraerts *et al.* [GRvB07] recently proposed a weaker acceleration, which avoids some possible underapproximations in [Fin93]. Emerson and Namjoshi [EN98] take into account the labeling of WSTS and consequently adapt the generalized Karp-Miller algorithm to model-checking. They assume the existence of a compatible dcpo, and generalize the Karp-Miller procedure to the case of broadcast protocols (which are equivalent to transfer Petri nets). However, termination is then not guaranteed [EFM99], and in fact neither is the existence of a finite representation of the cover. We solved the latter problem in [FG09a].

Abdulla, Collomb-Annichini, Bouajjani and Jonsson proposed a forward procedure for lossy channel systems [ACABJ04] using downward-closed regular languages as symbolic representations. Ganty, Geeraerts, Raskin and Van Begin [GRvB06b, GRvB06a] proposed a forward procedure for solving the coverability problem for WSTS equipped with an effective adequate domain of limits, or equipped with a finite set D used as a parameter to tune the precision of an abstract domain. Both solutions ensure that every downward-closed set has a finite representation. Abdulla *et al.* [ACABJ04] applied this framework to Petri nets and lossy channel systems. Abdulla, Deneux, Mahata and Nylén proposed a symbolic framework for dealing with downward-closed sets for Timed Petri nets [ADMN04].

Our Contribution. First, we define a *complete WSTS* as a WSTS $\mathfrak{S}$ whose well-ordering is also a continuous dcpo (a dcpo is a directed complete partial ordering). This allows us to design a conceptual procedure $\mathbf{Clover}_{\mathfrak{S}}$ that looks for a finite representation of the downward closure of the reachability set, i.e., of the cover [Fin90]. We call such a finite representation a *clover* (for *clo*sure of *cover*). This clearly separates the fundamental ideas from the data structures used in implementing Karp-Miller-like algorithms. Our procedure also terminates in more cases than the well-known (generalized) Karp-Miller procedure [EN98, Fin90]. We establish the main properties of clovers in Section 4 and use them to prove $\mathbf{Clover}_{\mathfrak{S}}$ correct, notably, in Section 6.

Second, we characterize complete WSTS for which $\mathbf{Clover}_{\mathfrak{S}}$ terminates. These are the ones that have a (continuous) flattening with the same clover. This establishes a surprising relationship with the theory of flattening [BFLS05]. The result (Theorem 7), together with its corollary on covers, rather than clovers (Theorem 8), is the main achievement of this paper.

Third, and building on our theory of completions [FG09a], we characterize those WSTS whose completion is a complete WSTS in the sense above. They are exactly the ω^2-*WSTS*, i.e., those whose state space is ω^2-wqo (a wqo is a well quasi-ordering), as we show in Section 5. All naturally occurring WSTS are in fact ω^2-WSTS. We shall also explain why this study is important: despite the fact that $\mathbf{Clover}_{\mathfrak{S}}$ cannot terminate on all inputs, that $\mathfrak{S}$ is an ω^2-WSTS will ensure *progress*, i.e., that every opportunity of accelerating a loop will eventually be taken by $\mathbf{Clover}_{\mathfrak{S}}$.

Finally, we apply our framework of complete WSTS to counter systems in Section 7. We show that affine counter systems may be completed into ∞-complete WSTS iff the domains of the monotonic affine functions are upward-closed.

2 Preliminaries

We borrow from theories of order, as used in model-checking [AČJT00, FS01], and also from domain theory [AJ94, GHK$^+$03]. A *quasi-ordering* $\leq$ is a reflexive and transitive relation on a set X. It is a (partial) *ordering* iff it is antisymmetric.

We write $<$ for the associated strict ordering $(\leq \setminus \geq)$, There is also an associated equivalence relation $\equiv$, defined as $\leq \cap \geq$.

A set X with a partial ordering $\leq$ is a *poset* $(X, \leq)$, or just X when $\leq$ is clear. If X is merely quasi-ordered by $\leq$, then the quotient $X/{\equiv}$ is ordered by the relation induced by $\leq$ on equivalence classes. So there is not much difference in dealing with quasi-orderings or partial orderings, and we shall essentially be concerned with the latter.

The *upward closure* $\uparrow E$ of a set E in X is $\{y \in X \mid \exists x \in E \cdot x \leq y\}$. The *downward closure* $\downarrow E$ is $\{y \in X \mid \exists x \in E \cdot y \leq x\}$. A subset E of X is *upward-closed* if and only if $E = \uparrow E$. *Downward-closed* sets are defined similarly. A *basis* of a downward-closed (resp. upward-closed) set E is a subset A such that $E = \downarrow A$ (resp. $E = \uparrow A$); E has a *finite basis* iff A can be chosen to be finite.

A quasi-ordering is *well* iff from any infinite sequence $x_0, x_1, \ldots, x_i, \ldots$, one can extract an infinite ascending chain $x_{i_0} \leq x_{i_1} \leq \ldots \leq x_{i_k} \leq \ldots$, with $i_0 < i_1 < \ldots < i_k < \ldots$. While *wqo* stands for well-quasi-ordered set, we abbreviate well posets as *wpos*.

An *upper bound* $x \in X$ of $E \subseteq X$ is such that $y \leq x$ for every $y \in E$. The *least upper bound (lub)* of a set E, if it exists, is written $\mathrm{lub}(E)$. An element x of E is *maximal* (resp. minimal) iff $\uparrow x \cap E = \{x\}$ (resp. $\downarrow x \cap E = \{x\}$). Write $\mathrm{Max}\, E$ (resp. $\mathrm{Min}\, E$) for the set of maximal (resp. minimal) elements of E.

A *directed subset* of X is any non-empty subset D such that every pair of elements of D has an upper bound in D. Chains, i.e., totally ordered subsets, and one-element sets are examples of directed subsets. A *dcpo* is a poset in which every directed subset has a least upper bound. For any subset E of a dcpo X, let $\mathrm{Lub}(E) = \{\mathrm{lub}(D) \mid D$ directed subset of $E\}$. Clearly, $E \subseteq \mathrm{Lub}(E)$; $\mathrm{Lub}(E)$ can be thought of E plus all limits from elements of E.

The *way below* relation $\ll$ on a dcpo X is defined by $x \ll y$ iff, for every directed subset D such that $\mathrm{lub}(D) \leq y$, there is a $z \in D$ such that $x \leq z$. Note that $x \ll y$ implies $x \leq y$, and that $x' \leq x \ll y \leq y'$ implies $x' \ll y'$. Write $\Downarrow E = \{y \in X \mid \exists x \in E \cdot y \ll x\}$, and $\Downarrow x = \Downarrow\{x\}$. X is *continuous* iff, for every $x \in X$, $\Downarrow x$ is a directed subset, and has x as least upper bound.

When $\leq$ is a well partial ordering that also turns X into a dcpo, we say that X is a *directed complete well order*, or *dcwo*. We shall be particularly interested in continuous dcwos.

A subset U of a dcpo X is (Scott-)*open* iff U is upward-closed, and for any directed subset D of X such that $\mathrm{lub}(D) \in U$, some element of D is already in U. A map $f : X \to X$ is (Scott-)*continuous* iff f is monotonic ($x \leq y$ implies $f(x) \leq f(y)$) and for every directed subset D of X, $\mathrm{lub}(f(D)) = f(\mathrm{lub}(D))$. Equivalently, f is continuous in the topological sense, i.e., $f^{-1}(U)$ is open for every open U.

A weaker requirement is ω-continuity: f is *ω-continuous* iff $\mathrm{lub}\{f(x_n) \mid n \in \mathbb{N}\} = f(\mathrm{lub}\{x_n \mid n \in \mathbb{N}\})$, for every countable chain $(x_n)_{n \in \mathbb{N}}$. This is all we require when we define accelerations, but general continuity is more natural in proofs. We won't discuss this any further: the two notions coincide when X is countable, which will always be the case of the state spaces X we are interested in, where the states should be representable on a Turing machine, hence at most countably many.

The *closed* sets are the complements of open sets. Every closed set is downward-closed. On a dcpo, the closed subsets are the subsets B that are both downward-closed and *inductive*, i.e., such that $\mathrm{Lub}(B) = B$. An inductive subset of X is none other than a sub-dcpo of X.

The *closure* $cl(A)$ of $A \subseteq X$ is the smallest closed set containing A. This should not be confused with the *inductive closure* $\mathrm{Ind}(A)$ of A, which is obtained as the smallest inductive subset B containing A. In general, $\downarrow A \subseteq \mathrm{Lub}(\downarrow A) \subseteq \mathrm{Ind}(\downarrow A) \subseteq cl(A)$, and all inclusions can be strict. All this nitpicking is irrelevant when X is a *continuous* dcpo, and A is downward-closed in X. In this case indeed, $\mathrm{Lub}(A) = \mathrm{Ind}(A) = cl(A)$. This is well-known, see e.g., [FG09a, Proposition 3.5], and will play an important role in our constructions. As a matter in fact, the fact that $\mathrm{Lub}(A) = cl(A)$, in the particular case of continuous dcpos, is required for lub-accelerations to ever reach the closure of the set of states that are reachable in a transition system.

3 A Survey on Well-Structured Transition Systems

WSTS were originally thought of as generalizations of Petri nets, in which the set of states (called markings) of a Petri net with n places, $\mathbb{N}^n$, is abstracted into a set X equipped with a wpo $\leq$; the Petri net transitions (which are affine translations from $\mathbb{N}^n$ into $\mathbb{N}^n$) are abstracted to general recursive monotonic functions from X to X. WSTS were defined and studied in the first author's PhD thesis in 1986, the results were presented at ICALP'87 [Fin87] and published in [Fin90]. The theory of WSTS has now been used for 25 years as a foundation for verification in various models, such as (monotonic extensions of) Petri nets, broadcast protocols, fragments of the pi-calculus fragments, rewriting systems, lossy systems, timed Petri nets, etc.

3.1 Well-Structured Transition Systems: From 1986 to 1996

A *transition system* is a pair $\mathfrak{S} = (S, \rightarrow)$ of a set S, whose elements are called *states*, and a *transition relation* $\rightarrow \subseteq S \times S$. We write $s \rightarrow s'$ for $(s, s') \in \rightarrow$. Let $\xrightarrow{*}$ be the transitive and reflexive closure of the relation $\rightarrow$. We write $Post_{\mathfrak{S}}(s) = \{s' \in S \mid s \rightarrow s'\}$ for the set of immediate successors of the state s. The *reachability set* of a transition system $\mathfrak{S} = (S, \rightarrow)$ from an initial state s_0 is $Post^*_{\mathfrak{S}}(s_0) = \{s \in S \mid s_0 \xrightarrow{*} s\}$.

We shall be interested in effective transition systems. Intuitively, a transition system $(S, \rightarrow)$ is *effective* iff one can compute the set of successors $Post_{\mathfrak{S}}(s)$ of any state s. We shall take this to imply that $Post_{\mathfrak{S}}(s)$ is finite, and each of its elements is computable.

An *ordered* transition system is a triple $\mathfrak{S} = (S, \rightarrow, \leq)$ where $(S, \rightarrow)$ is a transition system and $\leq$ is a partial ordering on S. We say that $(S, \rightarrow, \leq)$ is *effective* if $(S, \rightarrow)$ is effective and if $\leq$ is decidable.

We say that $\mathfrak{S} = (S, \rightarrow, \leq)$ is *monotonic* (resp. *strictly monotonic*) iff for all $s, s', s_1 \in S$ such that $s \rightarrow s'$ and $s_1 \geq s$ (resp. $s_1 > s$), there exists an $s'_1 \in S$ such that $s_1 \xrightarrow{*} s'_1$ and $s'_1 \geq s'$ (resp. $s'_1 > s'$). $\mathfrak{S}$ is *transitive monotonic* iff for all $s, s', s_1 \in S$ such that $s \rightarrow s'$ and $s_1 \geq s$, there exists an $s'_1 \in S$ such that $s_1 \xrightarrow{+} s'_1$ and $s'_1 \geq s'$. $\mathfrak{S}$ is *strongly monotonic* iff for all $s, s', s_1 \in S$ such that $s \rightarrow s'$ and $s_1 \geq s$, there exists an $s'_1 \in S$ such that $s_1 \rightarrow s'_1$ and $s'_1 \geq s'$. These variations on monotonicity were studied in [Fin87, FS01].

Finite representations of $Post^*_{\mathfrak{S}}(s)$, e.g., as Presburger formulae or finite automata, usually don't exist even for monotonic transition systems (not even speaking of being computable). However, the *cover* $Cover_{\mathfrak{S}}(s) = {\downarrow} Post^*_{\mathfrak{S}}({\downarrow} s)$ $(= {\downarrow} Post^*_{\mathfrak{S}}(s)$ when $\mathfrak{S}$ is monotonic) will be much better behaved. Note that being able to compute the cover allows one to decide *coverability* ($t \in Cover_{\mathfrak{S}}(s)$?), and *boundedness* (is $Post^*_{\mathfrak{S}}(s)$ finite?). Let us recall that the *control-state reachability problem* (when the set of states is $Q \times X$ with Q a finite set of control states) can be reduced to coverability. However, the *repeated control state reachability problem* (does there exist an infinite computation that visits infinitely often a control state q?) cannot be reduced to coverability.

The *eventuality* property for a given upward closed set I, is the following property: $\mathrm{EG}\, I$ is true in a state s_0 iff there is a computation from s_0 in which all states are in I. Given two labeled transition systems $\mathfrak{S}_1 = (S_1, \rightarrow_1)$ and $\mathfrak{S}_2 = (S_2, \rightarrow_2)$, on the same alphabet Σ, the relation $R \subseteq S_1 \times S_2$ is a *simulation* of $\mathfrak{S}_1$ by $\mathfrak{S}_2$ if for each $(s_1, s_2) \in R$, $s'_1 \in S_1$ and $a \in \Sigma$, if $s_1 \xrightarrow{a} s'_1$ then there exists $s'_2 \in S_2$ such that $s_2 \xrightarrow{a} s'_2$ and $(s'_1, s'_2) \in R$. We say that $s_1 \in S_1$ is *simulated* by $s_2 \in S_2$ if there is a simulation R of $\mathfrak{S}_1$ by $\mathfrak{S}_2$ such that $(s_1, s_2) \in R$. An ordered transition system $\mathfrak{S} = (S, \rightarrow, \leq)$ has the *effective PredBasis* property if there exists an algorithm which computes $\uparrow Pre(\uparrow s)$ for each $s \in S$; $\mathfrak{S}$ is *intersection effective* if there is an algorithm which computes a finite basis of $\uparrow s \cap \uparrow s'$, for all states $s, s' \in S$.

Definition 1. *An ordered transition system* $\mathfrak{S} = (S, \rightarrow, \leq)$ *is a* Well Structured Transition System *(*WSTS*) iff* $\mathfrak{S}$ *is monotonic and* $(S, \leq)$ *is wpo. A WSTS* $\mathfrak{S} = (S, \rightarrow, \leq)$ *is* effective *if* $(S, \rightarrow)$ *is effective (i.e.,* $Post(s)$ *is finite and computable for any* s*) and* $\leq$ *is decidable.*

In particular, an effective WSTS is finitely banching. Some of the decidability results do not require this but, for simplicity, we will make this assumption. Originally, three different definitions of monotonicity (hence six definitions with the strict variant) were given in [Fin87] and four (resp. eight) were studied in [FS01].

We now summarize the main decidability results on WSTS obtained between 1986 and 1996.

Theorem 1. *The following are* decidable*:*

- *Termination, for effective transitive monotonic WSTS [Fin87, FS01].*
- *Boundedness, for effective strictly monotonic transitive WSTS [Fin87, FS01].*
- *Coverability (hence control-state reachability), for effective WSTS with effective PredBasis ([AČJT00], extended in [FS01]).*
- *Eventuality, for effective strongly monotonic finitely branching WSTS (see [KS96, AČJT00], extended in [FS01]).*
- *Simulation of a labeled WSTS by a finite automaton, for intersection effective and effective strongly monotonic WSTS with effective PredBasis [AČJT00].*
- *Simulation of a finite automaton by a labeled WSTS, for effective strongly monotonic WSTS [AČJT00].*

The following are undecidable*:*

- *Reachability, for effective strongly strictly monotonic WSTS (Transfer Petri nets, [DFS98]).*
- *Repeated control-state reachability (hence LTL), for effective strongly strictly monotonic WSTS (Transfer Petri nets, [DFS98]).* □

To prove these decidability results we alternatively use forward and backward algorithms. Termination, boundedness, eventuality and one part of simulation can be proved by using a forward algorithm that builds the so-called Finite Reachability Tree (FRT) [Fin87]: we develop the reachability tree until a state larger than or equal to one of its ancestors is encountered, in which case the current branch is definitely closed. The place-boundedness problem (to decide whether a place can contain an unbounded number of tokens) is undecidable for transfer Petri nets [DFS98], although they are strongly and strictly monotonic WSTS. It is decidable for Petri nets. This requires a richer structure than the

FRT, the Karp-Miller tree. The set of labels of the Karp-Miller tree is a finite representation of the cover.

Almost all the assumptions used above are necessary:

Theorem 2. *The following are* undecidable*:*

- *Termination, for transitive monotonic WSTS.*
- *Boundedness, for effective strongly monotonic WSTS.*
- *Coverability, for effective strongly strictly monotonic WSTS.* □

For termination, Turing machines are transitive monotonic WSTS for which the termination ordering $\leq_{termination}$ is undecidable, [FS01]. For the second claim, Reset Petri nets have an undecidable bounded problem, and are effective strongly monotonic WSTS; but they are not strictly monotonic [DFS98]. For the last claim, there are WSTS composed of two recursive strictly monotonic functions from $\mathbb{N}^2$ into $\mathbb{N}^2$ that are not recursive on $\mathbb{N}^2_\omega$ hence there are no algorithm computing a PredBasis, [FMP04].

In writing this paper, we realized that the status of eventuality and simulation is open: for each of these properties, we know of no natural class of WSTS for which this property would be undecidable.

3.2 WSTS Everywhere: From 1997 to 2012

To the best of our knowledge, there have been no essential new results in the theory of WSTS between 1997 and 2003 (this does not mean that there are no interesting results about *particular* classes of WSTS). Let us just mention two kinds of results: the study of better quasi ordering (bqo) as an alternative to wqo [AN00], and the study of specific models such as Reset/Transfer Petri nets [DFS98], or Lossy Channel Systems [ABJ98]. Moreover, two papers synthesise the known results and show the possible applications: [AČJT00, FS01].

Many papers appeared during the period 2004-2012. We will not make an exhaustive list. Here are some of the papers that introduced new points of view, in our opinion:

2006. P. Ganty, G. Geeraerts, J.-F. Raskin and L. Van Begin proposed [GRvB06a, GRvB06b] a forward procedure for deciding the coverability problem. This is the first forward procedure for this problem in the general framework of WSTS. Their procedure computes a sufficient part (to decide coverability) of the finite representation of the cover.

2007 & 2011. P. Abdulla, G. Delzanno, G. Geeraerts, J.-F. Raskin and L. Van Begin studied [ADB07, GRVB07] the expressive power of WSTS by means of the set of coverability languages which are well-adapted to WSTS. Another, new approach, proposed by R. Bonnet, A. Finkel, S. Haddad and F. Rosa-Velardo in [BFHRV11], is to use the order type of posets to prove, for example, that the class of all WSTS with set of states of type $\mathbb{N}^n$ are less expressive than WSTS with set of states of type $\mathbb{N}^{n+1}$. This strategy unifies the previous proofs and allows us to compare models of different natures, such as lossy channel systems and timed Petri nets.

2004 & 2007 & 2011. R. Lazic, T. Newcomb, J. Ouaknine, A.W. Roscoe, J. Worrell, F. Rosa-Velardo, D. de Frutos-Escrig studied classes of Petri net extensions where tokens carry data: data nets, Petri data nets and ν-Petri nets [LNORW07, VF07, RMF11]. Affine and recursive Petri nets extensions were studied by A. Finkel, P. McKenzie, C. Picaronny in [FMP04]; affine well-structured nets are less expressive than ν-Petri nets.

Since 2009. We began in 2009 a series entitled "Forward analysis for WSTS, Part I: Completions" and "Forward Analysis for WSTS, Part II: Complete WSTS" in which we provide the missing theoretical fundations of finite representations of downward closed sets. This work, based on both order and topology, allowed us to design a new conceptual Karp and Miller procedure. Bounded WSTS [CFS11] are a particular recursive class of WSTS for which the new Karp and Miller procedure terminates.

Since 2010. D. Figueira, S. Figueira, S. Schmitz and Ph. Schnoebelen began the study of the complexity of general WSTS. They characterized the ordinal length of bad sequences of vectors of integers (using the Dickson lemma) and of words (using the Higman lemma) [FFSS11, SS11].

4 Complete WSTS Are Better

We will now present the recent papers on the computation of a finite representation of the cover. The material of what follows is a part of [FG09b]. We will consider transition systems that are *functional*, i.e., defined by a finite set of transition functions. This is, as in [FG09a], for reasons of simplicity. However, our **Clover**$_{\mathfrak{S}}$ procedure (Section 6), and already the technique of *accelerating loops* (Definition 4) depends on the considered transition system being functional. Formally, a *functional transition system* $(S, \xrightarrow{F})$ is a labeled transition system where the transition relation $\xrightarrow{F}$ is defined by a finite set F of partial functions $f : S \longrightarrow S$, in the sense that for every $s, s' \in S$, $s \xrightarrow{F} s'$ iff $s' = f(s)$ for some $f \in F$. If additionally, a partial ordering $\leq$ is given, a map $f : S \to S$ is *partial monotonic* iff $\operatorname{dom} f$ is upward-closed and for all $x, y \in \operatorname{dom} f$ with $x \leq y$, $f(x) \leq f(y)$. An *ordered functional transition system* is an ordered transition system $\mathfrak{S} = (S, \xrightarrow{F}, \leq)$ where F consists of partial monotonic functions. This is always strongly monotonic. A *functional WSTS* is an ordered functional transition system where $\leq$ is a well-ordering.

A functional transition system $(S, \xrightarrow{F})$ is *effective* if every $f \in F$ is computable: given a state s and a function f, we can decide whether $s \in \operatorname{dom} f$ and in this case, one can also compute $f(s)$.

For example, every Petri net, every reset/transfer Petri net, and in fact every affine counter system (see Definition 15) is an effective, functional WSTS.

4.1 Complete WSTS and Their Clovers

All forward procedures for WSTS rest on completing the given WSTS to one that includes all limits. E.g., the state space of Petri nets is $\mathbb{N}^k$, the set of all markings on k places, but the Karp-Miller algorithm works on $\mathbb{N}_\omega^k$, where $\mathbb{N}_\omega$ is $\mathbb{N}$ plus a new top element ω, with the usual componentwise ordering. We have defined general completions of wpos, serving as state spaces, and have briefly described completions of (functional) WSTS in [FG09a]. We temporarily abstract away from this, and consider *complete* WSTS directly.

Generalizing the notion of continuity to partial maps, we define:

Definition 2. *A* partial continuous *map $f : X \to X$, where $(X, \leq)$ is a dcpo, is a partial map whose domain* $\operatorname{dom} f$ *is open (not just upward-closed), and such that for every directed subset D in* $\operatorname{dom} f$, $\operatorname{lub}(f(D)) = f(\operatorname{lub}(D))$.

This is the special case of a more topological definition: in general, a partial continuous map $f : X \to Y$ is a partial map whose domain is open in X, and such that $f^{-1}(U)$ is open (in X, or equivalently here, in $\operatorname{dom} f$) for any open U of Y.

The composition of two partial continuous maps again yields a partial continuous map.

Definition 3 (Complete WSTS). *A* complete *transition system is a functional transition system $\mathfrak{S} = (S, \xrightarrow{F}, \leq)$ where $(S, \leq)$ is a continuous dcwo and every function in F is partial continuous. A* complete WSTS *is a functional WSTS that is complete as a functional transition system.*

The point in complete WSTS is that one can *accelerate* loops:

Definition 4 (Lub-acceleration). *Let $(X, \leq)$ be a dcpo, $f : X \to X$ be partial continuous. The* lub-acceleration $f^\infty : X \to X$ *is defined by:* $\operatorname{dom} f^\infty = \operatorname{dom} f$, *and for any $x \in \operatorname{dom} f$, if $x < f(x)$ then $f^\infty(x) = \operatorname{lub}\{f^n(x) \mid n \in \mathbb{N}\}$, else $f^\infty(x) = f(x)$.*

Note that if $x \leq f(x)$, then $f(x) \in \operatorname{dom} f$, and $f(x) \leq f^2(x)$. By induction, we can show that $\{f^n(x) \mid n \in \mathbb{N}\}$ is an increasing sequence, so that the definition makes sense.

Complete WSTS are strongly monotonic. One cannot decide, in general, whether a recursive function f is monotonic [FMP04] or continuous, whether an ordered set $(S, \leq)$ with a decidable ordering $\leq$, is a dcpo or whether it is a wpo. To show the latter claim for example, fix a finite alphabet Σ, and consider subsets S of Σ^* specified by a Turing machine $\mathcal{M}$ with tape alphabet Σ, so that S is the language accepted by $\mathcal{M}$.

We can also prove that given an effective ordered functional transition system, one cannot decide whether it is a WSTS, or a complete WSTS, in a similar way. However, the completion of *any* functional ω^2-WSTS is complete, as we shall see in Theorem 3.

In a complete WSTS, there is a *canonical* finite representation of the cover: the *clover* (a succinct description of the *cl*osure of the c*over*).

Definition 5 (Clover). *Let $\mathfrak{S} = (S, \xrightarrow{F}, \leq)$ be a complete WSTS. The* clover *$Clover_{\mathfrak{S}}(s_0)$ of the state $s_0 \in S$ is* $\operatorname{Max}\operatorname{Lub}(Cover_{\mathfrak{S}}(s_0))$.

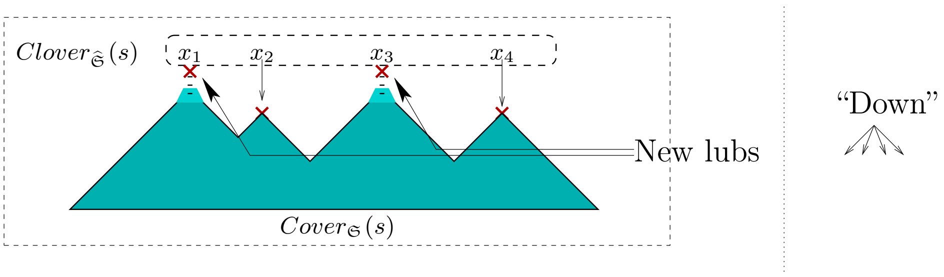


Fig. 1. The clover and the cover, in a complete space

This is illustrated in Figure 1. The "down" part on the right is meant to illustrate in which directions one should travel to go down in the chosen ordering. The cover $Cover_{\mathfrak{S}}(s_0)$ is a downward-closed subset, illustrated in blue (grey if you read this in black and white). $\mathrm{Lub}(Cover_{\mathfrak{S}}(s_0))$ has some new least upper bounds of directed subsets, here x_1 and x_3. The clover is given by just the maximal points in $\mathrm{Lub}(Cover_{\mathfrak{S}}(s_0))$, here x_1, x_2, x_3, x_4.

The fact that the clover is indeed a representation of the cover follows from the following.

Lemma 1. *Let $(S, \leq)$ be a continuous dcwo. For any closed subset F of S, $\mathrm{Max}\, F$ is finite and $F = {\downarrow} \mathrm{Max}\, F$.*

Proposition 1. *Let $\mathfrak{S} = (S, \xrightarrow{F}, \leq)$ be a complete WSTS, and $s_0 \in S$. Then $Clover_{\mathfrak{S}}(s_0)$ is finite, and $cl(Cover_{\mathfrak{S}}(s_0)) = {\downarrow} Clover_{\mathfrak{S}}(s_0)$.*

For any other representative, i.e., for any finite set R such that ${\downarrow} R = {\downarrow} Clover_{\mathfrak{S}}(s_0)$, $Clover_{\mathfrak{S}}(s_0) = \mathrm{Max}\, R$. Indeed, for any two finite sets $A, B \subseteq S$ such that ${\downarrow} A = {\downarrow} B$, $\mathrm{Max}\, A = \mathrm{Max}\, B$. So *Clover* is the *minimal representative* of the cover, i.e., there is no representative R with $|R| < |Clover_{\mathfrak{S}}(s_0)|$. The clover was called the minimal coverability set in [Fin93].

Despite the fact that the clover is always finite, it is non-computable in general (for example for Reset Petri nets) Nonetheless, it is computable on *flat* complete WSTS, and even on the larger class of *clover-flattable* complete WSTS (Theorem 7 below).

4.2 Completions

Many WSTS are not complete: the set $\mathbb{N}^k$ of states of a Petri net with k places is not even a dcpo. The set of states of a lossy channel system with k channels, $(\Sigma^*)^k$, is not a dcpo for the subword ordering either. We have defined general completions of wpos, and of WSTS, in [FG09a], a construction which we recall quickly.

The *completion* $\widehat{X}$ of a wpo $(X, \leq)$ is defined in any of two equivalent ways. First, $\widehat{X}$ is the *ideal completion* $\mathrm{Idl}(X)$ of X, i.e., the set of ideals of X, ordered by inclusion, where an *ideal* is a downward-closed directed subset of X. The least upper bound of a directed family of ideals $(D_i)_{i\in I}$ is their union. $\widehat{X}$ can also be described as the sobrification $\mathcal{S}(X_a)$ of the Noetherian space X_a, but this is probably harder to understand.

There is an embedding $\eta_X : X \to \widehat{X}$, i.e., an injective map such that $x \le x'$ in X iff $\eta_X(x) \le \eta_X(x')$ in $\widehat{X}$. This is defined by $\eta_X(x) = {\downarrow} x$. This allows us to consider X as a subset of $\widehat{X}$, by equating X with its image $\eta_X\langle X\rangle$, i.e., by equating each element $x \in X$ with ${\downarrow} x \in \widehat{X}$. However, we shall only do this in informal discussions, as this tends to make proofs messier.

For instance, if $X = \mathbb{N}^k$, e.g., with $k = 3$, then $(1,3,2)$ is equated with the ideal ${\downarrow}(1,3,2)$, while $\{(1,m,n) \mid m,n \in \mathbb{N}\}$ is a *limit*, i.e. an element of $\widehat{X} \setminus X$; the latter is usually written $(1,\omega,\omega)$, and is the least upper bound of all $(1,m,n)$, $m,n \in \mathbb{N}$. The downward-closure of $(1,\omega,\omega)$ in $\widehat{X}$, intersected with X, gives back the set of non-limit elements $\{(1,m,n) \mid m,n \in \mathbb{N}\}$.

This is a general situation: one can always write $\widehat{X}$ as the disjoint union $X \cup L$, so that any downward-closed subset D of X can be written as $X \cap {\downarrow} A$, where A is a *finite* subset of $X \cup L$. Then L, the set of limits, is a *weak adequate domain of limits* (WADL) for X—we slightly simplify Definition 3.1 of [FG09a], itself a slight generalization of [GRvB06b]. In fact, $\widehat{X}$ (minus X) is the *smallest* WADL [FG09a, Theorem 3.4].

$\widehat{X} = \mathrm{Idl}(X)$ is always a continuous dcpo. In fact, it is even algebraic [AJ94, Proposition 2.2.22]. It may however fail to be well, hence to be a continuous dcwo, see [FG12b, Section 4.2].

We have also described a hierarchy of datatypes on which completions are effective [FG09a, Section 5]. Notably, $\widehat{\mathbb{N}} = \mathbb{N}_\omega$, $\widehat{A} = A$ for any finite poset, and $\widehat{\prod_{i=1}^k X_i} = \prod_{i=1}^k \widehat{X}_i$. Also, $\widehat{X^*}$ is the space of *word-products* on X. These are the products, as defined in [ABJ98], i.e., regular expressions that are products of *atomic expressions* A^* ($A \in \mathbb{P}_{\text{fin}}(\widehat{X})$, $A \neq \emptyset$) or $a^?$ ($a \in \widehat{X}$). In any case, elements of completions $\widehat{X}$ have a finite description, and the ordering $\subseteq$ on elements of $\widehat{X}$ is decidable [FG09a, Theorem 5.3].

Having defined the completion $\widehat{X}$ of a wpo X, we can define the completion $\mathfrak{S} = \widehat{\mathfrak{X}}$ of a (functional) WSTS $\mathfrak{X} = (X, \xrightarrow{F}, \le)$ as $(\widehat{X}, \xrightarrow{\mathcal{S}F}, \subseteq)$, where $\mathcal{S}F = \{\mathcal{S}f \mid f \in F\}$ [FG09a, Section 6]. For each partial monotonic map $f \in F$, the partial continuous map $\mathcal{S}f : \widehat{X} \to \widehat{X}$ is such that $\mathrm{dom}\,\mathcal{S}f = \{C \in \widehat{X} \mid C \cap \mathrm{dom}\, f \neq \emptyset\}$, and $\mathcal{S}f(C) = {\downarrow} f\langle C\rangle$ for every $C \in \widehat{X}$. In the cases of Petri nets or functional-lossy channel systems, the completed WSTS is effective [FG09a, Section 6].

The important fact, which assesses the importance of the clover, is Proposition 2 below. We first require a useful lemma. Up to the identification of X with its image $\eta_X\langle X\rangle$, this states that for any downward-closed subset F of $\widehat{X}$, $cl(F) \cap X = F \cap X$, i.e., taking the closure of F only adds new limits, no proper elements of X.

Up to the identification of X with $\eta_X\langle X\rangle$, the next proposition states that $Cover_{\mathfrak{X}}(s_0) = Cover_{\mathfrak{S}}(s_0) \cap X = {\downarrow} Clover_{\mathfrak{S}}(s_0) \cap X$. In other words, to compute the cover of s_0 in the WSTS $\mathfrak{X}$ on the state space X, one can equivalently

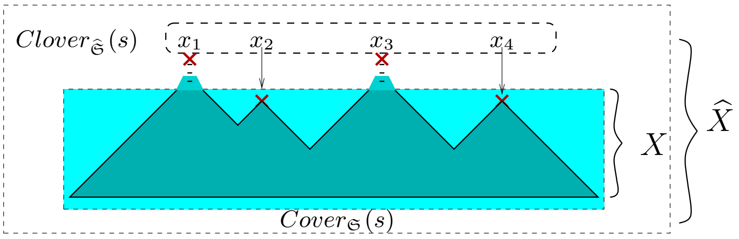


Fig. 2. The clover and the cover, in a completed space

compute the cover s_0 in the completed WSTS $\widehat{\mathfrak{X}}$, and keep only those non-limit elements (first equality of Proposition 2). Or one can equivalently compute the *closure* of the cover in the completed WSTS $\widehat{\mathfrak{X}}$, in the form of the downward closure $\downarrow Clover_{\mathfrak{S}}(s_0)$ of its clover. The closure of the cover will include extra limit elements, compared to the cover, but no non-limit element. This is illustrated in Figure 2.

Proposition 2. *Let $\mathfrak{S} = \widehat{\mathfrak{X}}$ be the completion of the functional WSTS $\mathfrak{X} = (X, \xrightarrow{F}, \leq)$. For every state $s_0 \in X$, $Cover_{\mathfrak{X}}(s_0) = \eta_X^{-1}(Cover_{\mathfrak{S}}(\eta_X(s_0))) = \eta_X^{-1}(\downarrow Clover_{\mathfrak{S}}(\eta_X(s_0)))$.*

$Cover_{\mathfrak{S}}(s_0)$ is contained, usually strictly, in $\downarrow Clover_{\mathfrak{S}}(s_0)$. The above states that, when restricted to non-limit elements (in X), both contain the same elements. Taking lub-accelerations $(\mathcal{S}f)^\infty$ of any composition f of maps in F may leave $Cover_{\mathfrak{S}}(s_0)$, but is always contained in $\downarrow Clover_{\mathfrak{S}}(s_0) = cl(Cover_{\mathfrak{S}}(s_0))$. So we can safely lub-accelerate in $\mathfrak{S} = \widehat{\mathfrak{X}}$ to compute the clover in $\mathfrak{S}$. While the clover is larger than the cover, taking the intersection back with X will produce exactly the cover $Cover_{\mathfrak{X}}(s_0)$.

In more informal terms, the cover is the set of states reachable by either following the transitions in F, or going down. The closure of the cover $\downarrow Clover_{\mathfrak{S}}(s_0)$ contains not just states that are reachable in the above sense, but also the limits of chains of such states. One may think of the elements of $\downarrow Clover_{\mathfrak{S}}(s_0)$ as being those states that are "reachable in infinitely many steps" from s_0. And we hope to find the finitely many elements of $Clover_{\mathfrak{S}}(s_0)$ by doing enough lub-accelerations.

5 Completion of WSTS into Complete WSTS Is (Almost) Always Possible

It would seem clear that the construction of the completion $\mathfrak{S} = \widehat{\mathfrak{X}}$ of a WSTS $\mathfrak{X} = (X, \xrightarrow{F}, \leq)$ be, again, a WSTS. We shall show that this is not the case. The only missing ingredient to show that $\mathfrak{S}$ is a complete WSTS is to check that $\widehat{X}$ is well-ordered by inclusion. We have indeed seen that $\widehat{X}$ is a continuous dcpo;

and $\mathfrak{S}$ is strongly monotonic, because $\mathcal{S}f$ is continuous, hence monotonic, for every $f \in F$.

Next, we shall concern ourselves with the question: under what condition on $\mathfrak{X}$ is $\mathfrak{S} = \widehat{\mathfrak{X}}$ again a WSTS? Equivalently, when is $\widehat{X}$ well-ordered by inclusion? We shall see that there is a definite answer: when X is ω^2-wqo.

5.1 Motivation

The question may seem mostly of academic interest. Instead, we illustrate that it is crucial to establish a *progress* property described below.

Let us imagine a procedure in the style of the Karp-Miller tree construction. We shall provide an abstract version of one, $\mathbf{Clover}_{\mathfrak{S}}$, in Section 6. However, to make things clearer, we shall use a direct imitation of the Karp-Miller procedure for Petri nets for now, generalized to arbitrary WSTS. This is a slight variant of the *generalized Karp-Miller procedure* of [Fin87, Fin90], and we shall therefore call it as such.

We build a tree, with nodes labeled by elements of the completion $\widehat{X}$, and edges labelled by transitions $f \in F$. During the procedure, nodes can be marked extensible or non-extensible. We start with the tree with only one node labeled s_0, and mark it extensible. At each step of the procedure, we pick an extensible leaf node N, labeled with $s \in \widehat{X}$, say, and add new children to N. For each $f \in F$ such that $s \in \operatorname{dom} \mathcal{S}f$, let $s' = \mathcal{S}f(s)$, and add a new child N' to N. The edge from N to N' is labeled f. If s' already labels some ancestor of N', then we label N' with s' and mark it non-extensible. If $s'' \leq s'$ for no label s'' of an ancestor of N', then we label N' with s' and mark it extensible. Finally, if $s'' < s'$ for some label s'' of an ancestor N_0 of N' (what we shall refer to as case (*) below), then the path from N_0 to N' is labeled with a sequence of functions $f_1, \ldots, f_p$ from F, and we label N' with the lub-acceleration $(f_p \circ \ldots \circ f_1)^\infty(s'')$. (There is a subtle issue here: if there are several such ancestors N_0, then we possibly have to lub-accelerate several sequences $f_1, \ldots, f_p$ from the label s'' of N_0: in this case, we must create several successor nodes N', one for each value of $(f_p \circ \ldots \circ f_1)^\infty(s'')$.) When $X = \mathbb{N}^k$ and each $f \in F$ is a Petri net transition, this is the Karp-Miller procedure, up to the subtle issue just mentioned, which we shall ignore.

Let us recall that the Karp-Miller tree (and also the reachability tree) is *finitely branching*, since the set F of functions is finite. This will allow us to use König's Lemma, which states that any finitely branching, infinite tree has at least one infinite branch.

The reasons why the original Karp-Miller procedure terminates on (ordinary) Petri nets are two-fold. First, when $\widehat{X} = \mathbb{N}^k_\omega$, one cannot lub-accelerate more than k times, because each lub-acceleration introduces a new ω component to the label of the produced state, which will not disappear in later node extensions. This is specific to Petri nets, and already fails for reset Petri nets, where ω components do disappear.

The second reason is of more general applicability: $\widehat{X} = \mathbb{N}^k_\omega$ is wpo, and this implies that along every infinite branch of the tree thus constructed, case (*)

will eventually happen, and in fact will happen infinitely many times. Call this *progress*: along any infinite path, one will lub-accelerate infinitely often. In the original Karp-Miller procedure for Petri nets, this will entail termination.

As we have already announced, for WSTS other than Petri nets, termination cannot be ensured. But at least we would like to ensure progress. The argument above shows that progress is obtained provided $\widehat{X}$ is wpo (or even just wqo). *This* is our main motivation in characterizing those wpos X such that $\widehat{X}$ is wpo again.

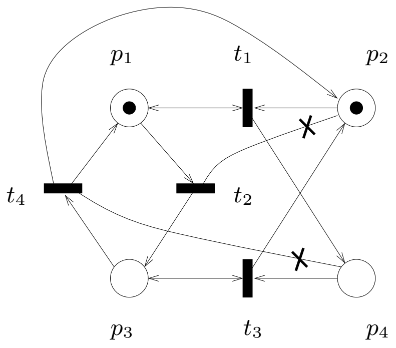


Fig. 3. The reset Petri net from [DFS98]

Before we proceed, let us explain why termination cannot be ensured. Generally, this will follow from undecidability arguments Here is a concrete case of non-termination. Consider the reset Petri net of [DFS98, Example 3], see Figure 3. This net has 4 places and 4 transitions, hence defines an transition system on $\mathbb{N}^4$. Its transitions are: $t_1(n_1, n_2, n_3, n_4) = (n_1, n_2 - 1, n_3, n_4 + 1)$ if $n_1, n_2 \geq 1$, $t_2(n_1, n_2, n_3, n_4) = (n_1 - 1, 0, n_3 + 1, n_4)$ if $n_1 \geq 1$, $t_3(n_1, n_2, n_3, n_4) = (n_1, n_2 + 1, n_3, n_4 - 1)$ if $n_3, n_4 \geq 1$, and $t_4(n_1, n_2, n_3, n_4) = (n_1 + 1, n_2 + 1, n_3 - 1, 0)$ if $n_3 \geq 1$. Note that $t_4(t_3^{n_2}(t_2(t_1^{n_2}(1, n_2, 0, 0)))) = (1, n_2 + 1, 0, 0)$ whenever $n_2 \geq 1$. The generalized Karp-Miller tree procedure, starting from $s_0 = (1, 1, 0, 0)$, will produce a child labeled $(1, 0, 0, 1)$ through t_1, then $(0, 0, 1, 1)$ through t_2, then $(0, 1, 1, 0)$ through t_3. Using t_4 leads us to case (*) with $s' = (1, 2, 0, 0)$. So the procedure will lub-accelerate the sequence $t_1 t_2 t_3 t_4$, starting from $s_0 = (1, 1, 0, 0)$. However $(t_4 \circ t_3 \circ t_2 \circ t_1)(s') = (1, 1, 0, 0) = s'$ again, so the sequence of iterates $(t_4 \circ t_3 \circ t_2 \circ t_1)^n(s_0)$ stabilizes at s', and $(t_4 \circ t_3 \circ t_2 \circ t_1)^\infty(s_0) = s'$. So the procedure adds a node labeled $s' = (1, 2, 0, 0)$. Similarly, starting from the latter, the procedure will eventually lub-accelerate the sequence $t_1^2 t_2 t_3^2 t_4$, producing a node labeled $(1, 3, 0, 0)$, and in general produce nodes labeled $(1, i + 1, 0, 0)$ for any $i \geq 1$ after having lub-accelerated the sequence $t_1^i t_2 t_3^i t_4$ from a node labeled $(1, i, 0, 0)$. In particular, the generalized Karp-Miller tree procedure will generate infinitely many nodes, and therefore fail to terminate.

This example also illustrates the following: progress does *not* mean that we shall eventually compute limits $g^\infty(s)$ that could not be reached in finitely many steps. In the example above, we do lub-accelerate infinitely often, and compute $(t_4 \circ t_3^i \circ t_2 \circ t_1^i)^\infty(1, i, 0, 0)$, but none of these lub-accelerations actually serve any purpose, since $(t_4 \circ t_3^i \circ t_2 \circ t_1^i)^\infty(1, i, 0, 0) = (1, i+1, 0, 0)$ is already equal to $(t_4 \circ t_3^i \circ t_2 \circ t_1^i)(1, i, 0, 0)$.

Progress will take a slightly different form in the actual procedure **Clover**$_{\mathfrak{S}}$ of Section 6. In fact, the latter will not build a tree, as the tree is in fact only algorithmic support for ensuring a fair choice of a state in $\widehat{X}$, and essentially acts as a distraction. However, progress will be crucial (Proposition 5 states that if the set of values computed by the procedure **Clover**$_{\mathfrak{S}}$ is finite then **Clover**$_{\mathfrak{S}}$ terminates) in our characterization of the cases where **Clover**$_{\mathfrak{S}}$ terminates (Theorem 7), as those states that are clover-flattable (see Section 6). Without it, **Clover**$_{\mathfrak{S}}$ would terminate in strictly less cases.

5.2 ω^2-WSTS

Recall here the working definition in [Jan99]: a well-quasi-order X is *ω^2-wqo* if and only if $(\mathbb{P}(X), \leq^\sharp)$ is wqo (where $A \leq^\sharp B$ iff for every $b \in B$, there is an $a \in A$ such that $a \leq b$ or equivalently iff $\uparrow B \subseteq \uparrow A$ iff $B \subseteq \uparrow A$). We show that the above is the only case that can go bad:

Proposition 3. *Let S be a well-quasi-order. Then $\widehat{S}$ is well-quasi-ordered by inclusion iff S is ω^2-wqo.*

Let an *ω^2-WSTS* be any WSTS whose underlying poset is ω^2-wqo. It follows:

Theorem 3. *Let $\mathfrak{S} = (S, \xrightarrow{F}, \leq)$ be a functional WSTS. Then $\widehat{\mathfrak{S}}$ is a (complete, functional) WSTS iff $\mathfrak{S}$ is an ω^2-WSTS.* $\square$

5.3 Are ω^2-wqos Ubiquitous?

It is natural to ask whether this is the norm or an exception. We claim that all wpos used in the verification literature are in fact ω^2-wpo.

Consider the following grammar of datatypes, which extends that of [FG09a, Section 5] with the case of finite trees (last line):

$$\begin{array}{rll} D ::= & \mathbb{N} & \text{natural numbers} \\ \mid & A_\leq & \text{finite set } A\text{, ordered by } \leq \\ \mid & D_1 \times \ldots \times D_k & \text{finite product} \\ \mid & D_1 + \ldots + D_k & \text{finite, disjoint sum} \\ \mid & D^* & \text{finite words} \\ \mid & D^\circledast & \text{finite multisets} \\ \mid & \mathcal{T}(D) & \text{finite trees} \end{array} \tag{1}$$

Then:

Proposition 4. *Every datatype defined in (1) is ω^2-wqo, and in fact bqo.*

In fact, all naturally occurring wqos are bqos, perhaps to the notable exception of finite graphs quasi-ordered by the graph minor relation, which are wqo [RS04] but not known to be bqo.

5.4 Effective Complete WSTS

The completion $\widehat{\mathfrak{S}}$ of a WSTS $\mathfrak{S}$ is effective iff the completion $\widehat{S}$ of the set of states is effective and $\mathcal{S}f$ is recursive for all $f \in F$. $\widehat{S}$ is effective for all the data types of [FG09a, Section 5]

Also, $\mathcal{S}f$ is indeed recursive for all $f \in F$, whether in Petri nets, functional-lossy channel systems, and reset/transfer Petri nets notably.

In the case of ordinary or reset/transfer Petri nets, and in general for all affine counter systems (which we shall investigate from Definition 15 on), $\mathcal{S}f$ coincides with the extension $\overline{f}$ defined in [FMP04, Section 2]: whenever $\operatorname{dom} f$ is upward-closed and $f : \mathbb{N}^k \to \mathbb{N}^k$ is defined by $f(\boldsymbol{s}) = A\boldsymbol{s} + \boldsymbol{a}$, for some matrix $A \in \mathbb{N}^{k\times k}$ and vector $\boldsymbol{a} \in \mathbb{Z}^k$, then $\operatorname{dom} \mathcal{S}f = \uparrow_S \operatorname{dom} f$, and $\mathcal{S}(f)(\boldsymbol{s})$ is again defined as $A\boldsymbol{s} + \boldsymbol{a}$, this time for all $\boldsymbol{s} \in \mathbb{N}^k_\omega$, and using the convention that $0 \times \omega = 0$ when computing the matrix product $A\boldsymbol{s}$ [FMP04, Theorem 7.9].

6 A Conceptual Karp-Miller Procedure

There are some advantages in using a forward procedure to compute (part of) the clover for solving coverability. For depth-bounded processes, a fragment of the π-calculus, the simple algorithm that works backward (computing the set of predecessors of an upward-closed initial set) of [AČJT00] is not applicable when the maximal depth of configurations is not known in advance because, in this case, the predecessor configurations are not effectively computable [WZH10]. It has been also proved that, unlike backward algorithms (which solve coverability without computing the clover), the Expand, Enlarge and Check forward algorithm of [GRvB07], which operates on complete WSTS, solves coverability by computing a *sufficient* part of the clover, even though the depth of the process is not known a priori [WZH10]. Recently, Zufferey, Wies and Henzinger proposed to compute a part of the clover by using a particular widening, called a *set-widening operator* [ZWH12], which loses some information, but always terminates and seems sufficiently precise to compute the clover in various case studies.

Model-checking safety properties of WSTS can be reduced to coverability, but there are other properties, such as *boundedness* (is $Post^*_{\mathfrak{S}}(s)$ finite?) that cannot be reduced to coverability: boundedness is decidable for Petri nets but undecidable for Reset Petri nets [DFS98], hence for general WSTS.

Recall that being able to compute the clover allows one to decide not only *coverability* since t is coverable from s iff $t \in Cover_{\mathfrak{S}}(s)$ iff $\exists t' \in Clover_{\mathfrak{S}}(s)$ such that $t \leq t'$ but also boundedness, and place-boundedness. To the best of our knowledge, the only known algorithms that decide place-boundedness (and also some formal language properties such as regularity and context-freeness of Petri net languages) *require one to compute the clover.*

Another argument in favor of computing clovers is Emerson and Namjoshi's [EN98] approach to model-checking *liveness* properties of WSTS, which uses a finite (coverability) graph based on the clover. Since WSTS enjoy the finite path property ([EN98], Definition 7), model-checking liveness properties is decidable for complete WSTS for which the clover is computable.

All these reasons motivate us to *try* to compute the clover for classes of complete WSTS, even though it is not computable in general.

The key to designing some form of a Karp-Miller procedure, such as the generalized Karp-Miller tree procedure (Section 5.1) or the $\mathbf{Clover}_{\mathfrak{S}}$ procedure below is being able to *compute* lub-accelerations. Hence:

Definition 6 (∞-Effective). *An effective complete functional WSTS* $\mathfrak{S} = (S, \xrightarrow{F}, \leq)$ *is* ∞-effective *iff every function* g^∞ *is computable, for every* $g \in F^*$*, where* F^* *is the set of all compositions of maps in* F.

E.g., the completion of a Petri net is ∞-effective: not only is $\mathbb{N}_\omega^k$ a wpo, but every composition of transitions $g \in F^*$ is of the form $g(\boldsymbol{x}) = \boldsymbol{x} + \delta$, where $\delta \in \mathbb{Z}^k$. If $\boldsymbol{x} < g(\boldsymbol{x})$ then $\delta \in \mathbb{N}^k \setminus \{0\}$. Write $\boldsymbol{x}_i$ the ith component of $\boldsymbol{x}$, it follows that $g^\infty(\boldsymbol{x})$ is the tuple whose ith component is $\boldsymbol{x}_i$ if $\delta_i = 0$, ω otherwise.

Let $\mathfrak{S}$ be an ∞-effective WSTS, and write $A \leq^\flat B$ iff $\downarrow A \subseteq \downarrow B$, i.e., iff every element of A is below some element of B. This is the *Hoare quasi-ordering*, also known as the *domination* quasi-ordering. The following is a simple procedure which computes the clover of its input $s_0 \in S$ (when it terminates):

```
Procedure Clover_𝔖(s_0) :
1. $A \leftarrow \{s_0\}$;
2. while $Post_{\mathfrak{S}}(A) \not\leq^\flat A$ do
   (a) Choose fairly (see below) $(g, a) \in F^* \times A$ such that $a \in \mathrm{dom}\, g$;
   (b) $A \leftarrow A \cup \{g^\infty(a)\}$;
3. return $\mathrm{Max}\, A$;
```

Note that $\mathbf{Clover}_{\mathfrak{S}}$ is well-defined and all its lines are computable by assumption, provided we make clear what we mean by fair choice in line (a). Call A_m the value of A at the start of the $(m-1)$st turn of the loop at step 2 (so in particular $A_0 = \{s_0\}$). The choice at line (a) is *fair* iff, on every infinite execution, every pair $(g, a) \in F^* \times A_m$ will be picked at some later stage $n \geq m$.

A possible implementation of this fair choice is the generalized Karp-Miller tree construction of Section 5.1: organize the states of A as labeling nodes of a tree that we grow. At step m, A_m is the set of leaves of the tree, and case (*) of the generalized Karp-Miller tree construction ensures that all pairs $(g, a) \in F^* \times A_m$ will eventually be picked for consideration. However, the generalized Karp-Miller tree construction does some useless work, e.g., when two nodes of the tree bear the same label.

Most existing proposals for generalizing the Karp-Miller construction do build such a tree [KM69, Fin90, Fin93, GRvB07], or a graph [EN98]. We claim that this is mere algorithmic support for ensuring fairness, and that the goal of such procedures is to compute a finite representation of the cover. Our $\mathbf{Clover}_{\mathfrak{S}}$ procedure computes the clover, which is the minimal such representation, and isolates algorithmic details from the core construction.

We shall also see that termination of $\mathbf{Clover}_{\mathfrak{S}}$ has strong ties with the theory of *flattening* [BFLS05]. However, Bardin *et al.* require one to enumerate sets of the form $g^*(\boldsymbol{x})$, which is sometimes harder than computing the single element $g^\infty(\boldsymbol{x})$. For example, if $g : \mathbb{N}^k \to \mathbb{N}^k$ is an affine map $g(\boldsymbol{x}) = A\boldsymbol{x} + \boldsymbol{b} - \boldsymbol{a}$ for some matrix $A \in \mathbb{N}^{k\times k}$ and vectors $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{N}^k$, then $g^\infty(\boldsymbol{x})$ is computable as a vector in $\mathbb{N}^k_\omega$, as we have seen in Section 5.4. But $g^*(\boldsymbol{x})$ is not even definable by a Presburger formula in general, in fact even when g is a composition of Petri net transitions; this is because reachability sets of Petri nets are not semi-linear in general [HP79].

Finally, we use a *fixpoint test* (line 2) that is not in the Karp-Miller algorithm; and this improvement allows $\mathbf{Clover}_\mathfrak{S}$ to terminate in *more cases* than the Karp-Miller procedure when it is used for extended Petri nets (for reset Petri nets for instance, which are a special case of the affine maps above), as we shall see. To decide whether the current set A, which is always an under-approximation of $Clover_\mathfrak{S}(s_0)$, is the clover, it is enough to decide whether $Post_\mathfrak{S}(A) \leq^\flat A$. The various Karp-Miller procedures only test each branch of a tree separately, to the partial exception of the minimal coverability tree algorithm [Fin90] and Geeraerts *et al.*'s recent coverability algorithm [GRvB07], which compare nodes across branches. That the simple test $Post_\mathfrak{S}(A) \leq^\flat A$ does all this at once does not seem to have been observed until now.

6.1 Correctness and Termination of the Clover Procedure

We cannot hope to have $\mathbf{Clover}_\mathfrak{S}$ terminate on all inputs. But we can at least start by showing that it is correct, whenever it terminates. This will be Theorem 4 below.

We first show that if $\mathbf{Clover}_\mathfrak{S}$ terminates then the computed set A is contained in $\mathrm{Lub}(Post^*_\mathfrak{S}(s_0))$. It is crucial that $\mathrm{Lub}(F) = cl(F)$ for any downward-closed set F, which holds because the state space S is a continuous dcpo. We use this through invocations to Proposition 1.

If the procedure $\mathbf{Clover}_\mathfrak{S}$ does not stop, it will compute an infinite sequence of sets of states. In other words, $\mathbf{Clover}_\mathfrak{S}$ does not deadlock. This is the progress property mentioned in Section 5.1.

Proposition 5 (Progress). *Let $\mathfrak{S}$ be an ∞-effective complete functional WSTS and A_n be the value of the set A, computed by the procedure* $\mathbf{Clover}_\mathfrak{S}$ *on input s_0, after n iterations of the while statement at line 2. If $\bigcup_n A_n$ is finite, then the procedure* $\mathbf{Clover}_\mathfrak{S}$ *terminates on input s_0.*

While $\mathbf{Clover}_\mathfrak{S}$ is non-deterministic, this is *don't care non-determinism*: if one execution does not terminate, then no execution terminates. If $\mathbf{Clover}_\mathfrak{S}$ terminates, then it computes the clover, and if it does not terminate, then at each step n, the set A_n is contained in the clover. Let us recall that $A_n \leq^\flat A_{n+1}$. We can now prove:

Theorem 4 (Correctness). *If* $\mathbf{Clover}_\mathfrak{S}(s_0)$ *terminates, then it computes $Clover_\mathfrak{S}(s_0)$.*

If the generalized Karp-Miller tree procedure (see Section 5.1) terminates then it has found a finite set $g_1, g_2, ..., g_n$ of maps to lub-accelerate. These lub-accelerations will also be found by $\mathbf{Clover}_{\mathfrak{S}}$, by fairness. From the fixpoint test, $\mathbf{Clover}_{\mathfrak{S}}$ will also stop. So $\mathbf{Clover}_{\mathfrak{S}}$ terminates on at least all inputs where the generalized Karp-Miller tree procedure terminates. We can say more:

Proposition 6. *The procedure* $\mathbf{Clover}_{\mathfrak{S}}$ *terminates on strictly more input states* $s_0 \in S$ *than the generalized Karp-Miller tree procedure.*

Proof. Consider the reset Petri net of [DFS98, Example 3] again (Figure 3). Add a new transition $t_5(n_1, n_2, n_3, n_4) = (n_1 + 1, n_2 + 1, n_3 + 1, n_4 + 1)$. The generalized Karp-Miller procedure does not terminate on this modified reset Petri net starting from $s_0 = (1, 1, 0, 0)$, because it already does not terminate on the smaller one of Section 5.1. On the other hand, by fairness, $\mathbf{Clover}_{\mathfrak{S}}$ will sooner or later decide to pick a pair of the form (t_5, a) at line (a), and then immediately terminate with the maximal state $(\omega, \omega, \omega, \omega)$, which is the sole element of the clover. □

Deciding when $\mathbf{Clover}_{\mathfrak{S}}$ terminates is itself impossible. We first observe that $\mathbf{Clover}_{\mathfrak{S}}$ terminates on each bounded state.

Lemma 2. *Let* $\mathfrak{S} = (S, \xrightarrow{F})$ *be an* ∞*-effective complete WSTS, and* $s_0 \in S$ *a state such that the reachability set* $Post^*_{\mathfrak{S}}(s_0)$ *is finite. Then* $\mathbf{Clover}_{\mathfrak{S}}(s_0)$ *terminates.*

Proof. Since $Post^*_{\mathfrak{S}}(s_0)$ is finite, $g^\infty(s)$ is in $Post^*_{\mathfrak{S}}(s_0)$ for every $s \in Post^*_{\mathfrak{S}}(s_0)$ and every $g \in F^*$ with $s \in \operatorname{dom} g$. So, defining again A_n as the value of the set A computed by $\mathbf{Clover}_{\mathfrak{S}}$ on input s_0, after n iterations of the while statement at line 2, $\bigcup_{n \in \mathbb{N}} A_n$ is contained in $Post^*_{\mathfrak{S}}(s_0)$, hence finite. By Proposition 5, $\mathbf{Clover}_{\mathfrak{S}}(s_0)$ terminates. □

Proposition 7. *There is an* ∞*-effective complete WSTS* $\mathfrak{S} = (S, \xrightarrow{F})$ *such that we cannot decide, given* $s_0 \in S$*, whether* $\mathbf{Clover}_{\mathfrak{S}}(s_0)$ *will terminate.*

The following result was more generally stated in [Fin87] (but without sufficient effective and completeness hypotheses) and it was also expressed for Recursive Well Structured Nets in [FMP04] where the ∞-effective hypothesis was replaced by a weaker condition that allows to compute a sufficient underapproximation of the limit of the $f^n(x)$ when n goes to infinity and for $x < f(x)$.

Theorem 5. *[BF12] For* ∞*-effective strictly monotonic complete WSTS* $\mathfrak{S} = (\mathbb{N}^n, \xrightarrow{F}, \leq)$*, the procedure* $\mathbf{Clover}_{\mathfrak{S}}(s_0)$ *terminates.*

There is another case in which the procedure $\mathbf{Clover}_{\mathfrak{S}}$ terminates. A functional transition system $\mathfrak{S} = (S, \xrightarrow{F})$ with initial state s_0 is *flat* iff there are finitely many words $w_1, w_2, ..., w_k \in F^*$ such that any fireable sequence of transitions from s_0 is contained in the language $w_1^* w_2^* ... w_k^*$. (We equate functions in F with letters from the alphabet F.) corresponding composition of maps, i.e., fg denotes $g \circ f$.) Ginsburg and Spanier [GS64] call this a *bounded* language, and show that it is decidable whether any context-free language is flat.

Theorem 6. *For* ∞*-effective complete flat WSTS* $\mathfrak{S} = (\mathbb{N}^n, \xrightarrow{F}, \leq)$*, the procedure* $\mathbf{Clover}_{\mathfrak{S}}(s_0)$ *terminates.*

6.2 Clover-Flattable Complete WSTS

We now characterize those ∞-effective complete WSTS on which $\mathbf{Clover}_{\mathfrak{S}}$ terminates.

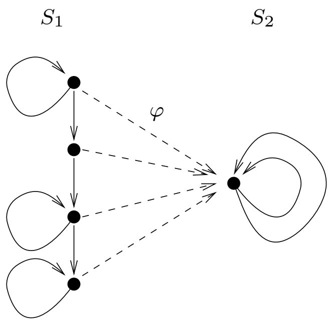


Fig. 4. Flattening

Not all systems of interest are flat. The simplest example of a non-flat system has one state q and two transitions $q\xrightarrow{a}q$ and $q\xrightarrow{b}q$.

For an arbitrary system S, *flattening* [BFLS05] consists in finding a flat system S', equivalent to S with respect to reachability, and in computing on S' instead of S. We adapt the definition in [BFLS05] to functional transition systems, without an explicit finite control graph for now (but see Definition 11).

Definition 7 (Flattening). *A* flattening *of a functional transition system* $\mathfrak{S}_2 = (S_2, \xrightarrow{F_2})$ *is a pair* $(\mathfrak{S}_1, \varphi)$*, where:*

1. $\mathfrak{S}_1 = (S_1, \xrightarrow{F_1})$ *is a* flat *functional transition system;*
2. *and* $\varphi : \mathfrak{S}_1 \to \mathfrak{S}_2$ *is a* morphism *of transition systems. That is,* φ *is a pair of two maps, both written* φ*, from* S_1 *to* S_2 *and from* F_1 *to* F_2*, such that for all* $(s, s') \in S_1^2$*, for all* $f_1 \in F_1$ *such that* $s \in \operatorname{dom} f_1$ *and* $s' = f_1(s)$*,* $\varphi(s) \in \operatorname{dom} \varphi(f_1)$ *and* $\varphi(s') = \varphi(f_1)(\varphi(s))$ *(see Figure 4).*

Let us recall that a pair $(\mathfrak{S}, s_0)$ of a transition system and a state is *Post*-flattable* iff there is a flattening $\mathfrak{S}_1$ of $\mathfrak{S}$ and a state s_1 of $\mathfrak{S}_1$ such that $\varphi(s_1) = s_0$ and $Post^*_{\mathfrak{S}}(s_0) = \varphi(Post^*_{\mathfrak{S}_1}(s_1))$.

Recall that we equate ordered functional transition systems $(S, \xrightarrow{F}, \leq)$ with their underlying function transition system $(S, \xrightarrow{F})$. The notion of flattening then extends to ordered functional transition systems. However, it is then natural to consider *monotonic flattenings*, where in addition $\varphi : S_1 \to S_2$ is monotonic. In the case of complete transition systems, the natural extension requires φ to be continuous:

Definition 8 (Continuous Flattening). *Let* $\mathfrak{S}_2 = (S_2, \xrightarrow{F_2}, \leq_2)$ *be a complete transition system. A flattening* $(\mathfrak{S}_1, \varphi)$ *of* $\mathfrak{S}_2$ *is* continuous *iff:*

1. $\mathfrak{S}_1 = (S_1, \xrightarrow{F_1}, \leq_1)$ *is a* complete *transition system;*
2. *and* $\varphi : S_1 \to S_2$ *is* continuous.

Definition 9 (Clover-Flattable). *Let $\mathfrak{S}$ be a complete transition system, and s_0 be a state. We say that $(\mathfrak{S}, s_0)$ is* clover-flattable *iff there is an continuous flattening $(\mathfrak{S}_1, \varphi)$ of $\mathfrak{S}$, and a state s_1 of $\mathfrak{S}_1$ such that:*

1. *$\varphi(s_1) = s_0$ (φ maps initial states to initial states);*
2. *$cl(Cover_{\mathfrak{S}}(s_0)) = cl(\varphi\langle cl(Cover_{\mathfrak{S}_1}(s_1))\rangle)$ (φ preserves the closures of the covers of the initial states).*

On complete WSTS—our object of study—, the second condition can be simplified to $\downarrow Clover_{\mathfrak{S}}(s_0) = \downarrow \varphi(Clover_{\mathfrak{S}_1}(s_1))$ (using Proposition 1 and the fact that φ, as a continuous map, is monotonic), or equivalently to $Clover_{\mathfrak{S}}(s_0) = \mathrm{Max}\, \varphi\langle Clover_{\mathfrak{S}_1}(s_1)\rangle$. Recall also that, when $\mathfrak{S}$ is the completion $\widehat{\mathfrak{X}}$ of a WSTS $\mathfrak{X} = (X, \xrightarrow{F}, \leq)$, the clover of $s_0 \in X$ is a finite description of the *cover* of s_0 in $\mathfrak{X}$ (Proposition 2), and this is what φ should preserve, up to taking downward closures.

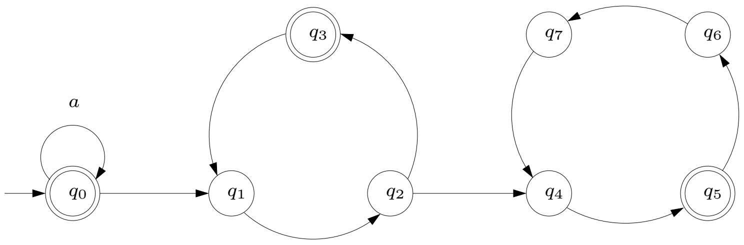


Fig. 5. An rl-automaton

Let us define the synchronized product.

Definition 10 (Synchronized Product). *Let $\mathfrak{S} = (S, \xrightarrow{F}, \leq)$ be a complete functional transition system, and $\mathcal{A} = (F, Q, \delta, q_0)$ be an rl-automaton on the same alphabet F.*

Define the synchronized product $\mathfrak{S} \times \mathcal{A}$ *as the ordered functional transition system $(S \times Q, \xrightarrow{F'}, \leq')$, where F' is the collection of all partial maps $f \bowtie \delta : (s, q) \mapsto (f(s), \delta(q, f))$, for each $f \in F$ such that $\delta(q, f)$ is defined for some $q \in Q$. Let also $(s, q) \leq' (s', q')$ iff $s \leq s'$ and $q = q'$.*

Let π_1 be the morphism of transition systems defined as first projection on states; i.e., $\pi_1(s, q) = s$ for all $(s, q) \in S \times Q$, $\pi_1(f \bowtie \delta) = f$ for all $f \in F$.

Lemma 3 (Synchronized Product). *Let $\mathfrak{S} = (S, \xrightarrow{F}, \leq)$ be a complete functional transition system, and $\mathcal{A} = (F, Q, \delta, q_0)$ be an rl-automaton on the same alphabet F.*

Then $(\mathfrak{S} \times \mathcal{A}, \pi_1)$ is a continuous flattening of $\mathfrak{S}$.

Strong flattenings are special: the decision to take the next action $f \in F$ from state (s, q) is dictated by the current control state q *only*, while ordinary flattenings allow more complex decisions to be made.

We say that a transition system is strongly clover-flattable iff we can require that the flat system $\mathfrak{S}_1$ is a synchronized product, and the continuous morphism of transition systems φ is first projection π_1:

Definition 11 (Strongly Clover-Flattable). *Let $\mathfrak{S} = (S, \xrightarrow{F})$ be a complete functional transition system. We say that $(\mathfrak{S}, s_0)$ is* strongly clover-flattable *iff there is an rl-automaton $\mathcal{A}$, say with initial state q_0, such that $cl(Cover_{\mathfrak{S}}(s_0)) = cl(\pi_1 \langle cl(Cover_{\mathfrak{S}\times\mathcal{A}}(s_0, q_0))\rangle)$.*

The following is then obvious.

Lemma 4. *On complete functional transition systems, the implications "strongly clover-flattable" $\Longrightarrow$ "clover-flattable" $\Longrightarrow$ "weakly clover-flattable" hold.*

It is also easy to show that "weakly clover-flattable" also implies "clover-flattable". However, we shall show something more general in Theorem 7 below.

We show in Proposition 8 that $\mathbf{Clover}_{\mathfrak{S}}(s_0)$ can only terminate when $(\mathfrak{S}, s_0)$ is strongly clover-flattable. We shall require the following lemma. For notational simplicity, we equate words $g_1 g_2$ with compositions $g_2 \circ g_1$.

Lemma 5. *Let $\mathfrak{S} = (S, \xrightarrow{F})$ be a complete functional transition system, and $s_0 \in F$. Assume ${g_1}^\infty {g_2}^\infty \dots {g_n}^\infty(s_0)$ is defined, and in some open subset U of S, for some $g_1, g_2, \dots, g_n \in F$. Then there are natural numbers $k_1, k_2, \dots, k_n$ such that $g_1^{k_1} g_2^{k_2} \dots g_n^{k_n}(s_0)$ is defined, and in U.*

Proposition 8. *Let $\mathfrak{S}$ be an ∞-effective complete WSTS. If $\mathbf{Clover}_{\mathfrak{S}}$ terminates on s_0, then $(\mathfrak{S}, s_0)$ is strongly clover-flattable.*

We now loop the loop and show that $\mathbf{Clover}_{\mathfrak{S}}$ terminates on s_0 whenever $(\mathfrak{S}, s_0)$ is weakly clover-flattable (Theorem 7 below). This may seem obvious. In particular, if $(\mathfrak{S}, s_0)$ is clover-flattable, then accelerate along the loops from $\mathfrak{S}_1$, where $\mathfrak{S}_1$, φ is a continuous flattening of $\mathfrak{S}$. The difficulty is that we *cannot* actually choose to accelerate whenever we want: the $\mathbf{Clover}_{\mathfrak{S}}$ procedure decides by itself when it should accelerate, independently of any flattening whatsoever.

There is an added difficulty, in the sense that one should also check that lub-accelerations, as they are used in $\mathbf{Clover}_{\mathfrak{S}}$, are enough to reach all required least upper bounds. The key point is the following lemma, which asserts the existence of finitely many subsequences $g^{p_j + \ell q_j}(s)$, $\ell \in \mathbb{N}$, whose exponents form infinite arithmetic progressions, and which generate all possible limits of directed families of elements of the form $g^n(s)$, $n \in \mathbb{N}$, except possibly for finitely many isolated points.

This is the point in our study where progress is needed. Indeed, we require S to be wpo to pick k and m in the proof below.

Lemma 6. *Let S be a dcwo, $g : S \to S$ a partial monotonic map, and $s \in S$. Consider the family G of all elements of the form $g^n(s)$, for those $n \in \mathbb{N}$ such* *that this is defined. Then there are finitely many directed subfamilies* G_0, G_1, $\ldots$, G_{m-1} *of* G *such that:*

1. $cl(G) = \bigcup_{j=0}^{m-1} cl(G_j) = \downarrow\{\mathrm{lub}(G_0), \mathrm{lub}(G_1), \ldots, \mathrm{lub}(G_{m-1})\}$*;*
2. *each* G_j *is either a one-element set* $\{g^{p_j}(s)\}$*, where* $p_j \in \mathbb{N}$*, or is a chain of the form* $\{g^{p_j+\ell q_j}(s) \mid \ell \in \mathbb{N}\}$*, where* $p_j \in \mathbb{N}$*,* $q_j \in \mathbb{N} \setminus \{0\}$*, and* $g^{p_j}(s) < g^{p_j+q_j}(s)$*;*
3. *for every* j*,* $0 \leq j < m$*,* $s \not< g^{p_j}(s)$*.*

Proposition 9. *Let* $\mathfrak{S}$ *be an* ∞*-effective complete WSTS. Assume that* $(\mathfrak{S}, s_0)$ *is weakly clover-flattable. Then* $\mathbf{Clover}_{\mathfrak{S}}$ *terminates on* s_0*.*

Putting together Lemma 4, Proposition 8, and Proposition 9, we obtain:

Theorem 7 (Main Theorem). *Let* $\mathfrak{S}$ *be an* ∞*-effective complete WSTS. The following statements are equivalent:*

1. $(\mathfrak{S}, s_0)$ *is clover-flattable;*
2. $(\mathfrak{S}, s_0)$ *is weakly clover-flattable;*
3. $(\mathfrak{S}, s_0)$ *is strongly clover-flattable;*
4. $\mathbf{Clover}_{\mathfrak{S}}(s_0)$ *terminates.* □

6.3 Cover-Flattability (without the "l" in "Cover")

Turning to non-complete WSTS, we define:

Definition 12 (Monotonic Flattening). *Let* $\mathfrak{X}_2 = (X_2, \xrightarrow{F_2}, \leq_2)$ *be an ordered functional transition system. A flattening* $(\mathfrak{X}_1, \varphi)$ *of* $\mathfrak{X}_2$ *is* monotonic *iff:*

1. $\mathfrak{X}_1 = (X_1, \xrightarrow{F_1}, \leq_1)$ *is an ordered functional transition system;*
2. *and* $\varphi : X_1 \to X_2$ *is* monotonic.

Definition 13 (Cover-Flattable). *Let* $\mathfrak{X}$ *be an ordered functional transition system, and* x_0 *be a state. We say that* $(\mathfrak{X}, x_0)$ *is* cover-flattable *iff there is a monotonic flattening* $(\mathfrak{X}_1, \varphi)$ *of* $\mathfrak{X}$*, and a state* x_1 *of* $\mathfrak{X}_1$ *such that:*

1. $\varphi(x_1) = x_0$*;*
2. $Cover_{\mathfrak{X}}(x_0) = \downarrow \varphi\langle Cover_{\mathfrak{X}_1}(x_1)\rangle$*.*

Theorem 8. *Let* $\mathfrak{X} = (X, \xrightarrow{F}, \leq)$ *be an* ω^2*-WSTS that is* ∞*-effective, in the sense that* $\widehat{\mathfrak{X}}$ *is* ∞*-effective, i.e., that* $(\mathcal{S}g)^\infty$ *is computable for every* $g \in F^*$*. The following statements are equivalent:*

1. $(\mathfrak{X}, x_0)$ *is cover-flattable;*
2. $(\widehat{\mathfrak{X}}, \eta_X(x_0))$ *is (weakly, strongly) clover-flattable;*
3. $\mathbf{Clover}_{\widehat{\mathfrak{X}}}(\eta_X(x_0))$ *terminates.*

In this case, $\mathbf{Clover}_{\widehat{\mathfrak{X}}}(\eta_X(x_0))$ *returns the clover* $A = Clover_{\mathfrak{S}}(s_0)$*, and this is a finite description of the cover, in the sense that* $Cover_{\mathfrak{X}}(x_0) = \eta_X^{-1}(\downarrow A)$*.*

By a slight abuse of language, say that a functional WSTS $\mathfrak{S} = (S, \xrightarrow{F}, \leq)$ is cover-flattable iff $(\mathfrak{S}, s_0)$ is cover-flattable for every initial state $s_0 \in S$.

Corollary 1. *Every Petri net, and every VASS, is cover-flattable.*

Proof. The state space of a Petri net on k places is $\mathbb{N}^k$, that of a VASS [HP79] is $Q \times \mathbb{N}^k$, where Q is a finite set of control states. We deal with the latter, as they are more general. Transitions of the VASS $\mathfrak{X}$ are of the form $f(q, \boldsymbol{x}) = (q', \boldsymbol{x} + \boldsymbol{b} - \boldsymbol{a})$, provided $\boldsymbol{x} \geq \boldsymbol{a}$, and where $\boldsymbol{a}$, $\boldsymbol{b}$ are fixed tuples in $\mathbb{N}^k$. It is easy to see that $\mathcal{S}f$ is defined by: $\mathcal{S}f(q, \boldsymbol{x}) = (q', \boldsymbol{x} + \boldsymbol{b} - \boldsymbol{a})$, provided $\boldsymbol{x} \geq \boldsymbol{a}$, this time for all $\boldsymbol{x} \in \mathbb{N}^k_\omega$. So the completion $\widehat{\mathfrak{S}}$ of the VASS is ∞-effective. On these, the Karp-Miller algorithm terminates [KM69], hence also the generalized Karp-Miller algorithm of Section 5.1. By Proposition 6, $\mathbf{Clover}_{\widehat{\mathfrak{S}}}$ terminates on any input $s_0 \in Q \times \mathbb{N}^k_\omega$. So $\mathfrak{X}$ is cover-flattable, by Theorem 8. □

Corollary 2. *There are reset Petri nets, and functional-lossy channel systems that are not cover-flattable.*

7 Well Structured Presburger Counter Systems

We now demonstrate how the fairly large class of counter systems fits with our theory. We show that counter systems composed of affine monotonic functions with upward-closed definition domains are complete (strongly monotonic) WSTS. This result is obtained by showing that every monotonic affine function f is continuous and its lub-acceleration f^∞ is computable [CFS11]. Moreover, we prove that it is possible to decide whether a general counter system (given by a finite set of Presburger relations) is a monotonic affine counter system, but that one cannot decide whether it is a WSTS.

Definition 14. *A* Presburger counter system *(with n counters), $\mathcal{C}$ is a tuple $\mathcal{C} = (Q, R, \rightarrow)$ where Q is a finite set of control states, $R = \{r_1, r_2, ...r_k\}$ is a finite set of Presburger relations $r_i \subseteq \mathbb{N}^n \times \mathbb{N}^n$ and $\rightarrow \subseteq Q \times R \times Q$.*

We will consider a special case of Presburger relations, those which allow us to encode the graph of affine functions. A (partial) function $f : \mathbb{N}^n \longrightarrow \mathbb{N}^n$ is *non-negative affine*, for short *affine* if there exist a matrix $A \in \mathbb{N}^{n \times n}$ with *non-negative coefficients* and a vector $b \in \mathbb{Z}^n$ such that for all $\boldsymbol{x} \in \operatorname{dom} f, f(\boldsymbol{x}) = A\boldsymbol{x} + \boldsymbol{b}$. When necessary, we will extend affine maps $f : \mathbb{N}^n \longrightarrow \mathbb{N}^n$ by continuity to $f : \mathbb{N}^n_\omega \longrightarrow \mathbb{N}^n_\omega$, by $f(\text{lub}_{i \in \mathbb{N}}(\boldsymbol{x}_i)) = \text{lub}_{i \in \mathbb{N}}(f(\boldsymbol{x}_i))$ for every countable chain $(\boldsymbol{x}_i)_{i \in \mathbb{N}}$ in $\mathbb{N}^n$. That is, we just write f instead of $\mathcal{S}f$.

Definition 15. *An* Affine Counter System *(with n counters), a.k.a. an* ACS *$\mathcal{C} = (Q, R, \rightarrow)$ is a Presburger counter system where all relations r_i are (partial) affine functions.*

The domain of maps f in an affine counter system ACS are Presburger-definable. A reset/transfer Petri net is an ACS where every line or column of every matrix contains at most one non-zero coefficient equal to 1, and, all domains are upward-closed sets. A *Petri net* is an ACS where all affine maps are translations with upward-closed domains.

Theorem 9. *One can decide whether an effective Presburger counter system is an ACS.*

Proof. The formula expressing that a relation is a function is a Presburger formula, hence one can decide whether R is the graph of a function. One can also decide whether the graph G_f of a function f is monotonic because monotonicity of a Presburger-definable function can be expressed as a Presburger formula. Finally, one can also decide whether a Presburger formula represents an affine function $f(\boldsymbol{x}) = A\boldsymbol{x} + \boldsymbol{b}$ with $A \in \mathbb{N}^{n\times n}$ and $\boldsymbol{b} \in \mathbb{Z}^n$, using results by Demri *et al.* [DFGvD06]. $\square$

For counter systems (which include Minsky machines), monotonicity is undecidable. Clearly, a counter system $\mathfrak{S}$ is well-structured iff $\mathfrak{S}$ is monotonic: so there is no algorithm to decide whether a Presburger counter system is a WSTS. However, an ACS is strongly monotonic iff each map f is partial monotonic; this is equivalent to requiring that $\operatorname{dom} f$ is upward-closed, since all matrices A have non-negative coefficients. This is easily cast as Presburger formula, and therefore decidable.

Proposition 10. *There is an algorithm to decide whether an ACS is a strongly monotonic WSTS.*

Proof. The strong monotony of an ACS $\mathcal{C}$ means that every function of $\mathcal{C}$ is monotonic and this can be expressed by a Presburger formula saying that all the (Presburger-definable) definition domains are upward-closed (the matrices are known to be positive). $\square$

We have recalled that the transitions function of Petri nets ($f(x) = x + \boldsymbol{b}$, $\boldsymbol{b} \in \mathbb{Z}^n$ and $\operatorname{dom}(f)$ upward-closed) can be lub-accelerated effectively. This result was generalized to broadcast protocols (equivalent to transfer Petri nets) by Emerson and Namjoshi [EN98] and to another class of monotonic affine functions $f(\boldsymbol{x}) = A\boldsymbol{x}+\boldsymbol{b}$ such that $A \in \mathbb{N}^{n\times n}$, $b \in \mathbb{N}^n$ (note that b is not in $\mathbb{Z}^n$) and $\operatorname{dom}(f)$ is upward closed [FMP04].

[CFS11] recently extended this result to all monotonic affine functions: for every $f(\boldsymbol{x}) = A\boldsymbol{x} + \boldsymbol{b}$ with $A \in \mathbb{N}^{n\times n}$, $\boldsymbol{b} \in \mathbb{Z}^n$ and $\operatorname{dom}(f)$ upward-closed, the function f^∞ is recursive.

We deduce the following strong relationship between well-structured ACS and complete well-structured ACS.

Theorem 10. *The completion of an ACS S is an ∞-effective complete WSTS iff S is a strongly monotonic WSTS.*

Proof. Strong monotonicity reduces to partial monotonicity of each map f, as discussed above. Well-structured *ACS* are clearly effective, since $Post(\boldsymbol{s}) = \{\boldsymbol{t} \mid \exists f \in F \cdot f(\boldsymbol{t}) = \boldsymbol{s}\}$ is Presburger-definable. Note also that monotonic affine function are continuous, and $\mathbb{N}^n_\omega$ is a continuous dcwo. Finally, for every Presburger monotonic affine function f, the function f^∞ is recursive, so the considered *ACS* is ∞-effective. $\square$

Corollary 3. *One can decide whether the completion of an ACS is an ∞-effective complete WSTS.*

So the completions of reset/transfer Petri nets [DFS98], broadcast protocols [EFM99], self-modifying Petri nets [Val78] and affine well-structured nets [FMP04] are ∞-effective complete WSTS.

8 Conclusion and Perspectives

We have provided a framework of *complete WSTS*, and of *completions* of WSTS, on which forward reachability analyses can be conducted, using natural finite representations for downward-closed sets. The central element of this theory is the *clover*, i.e., the set of maximal elements of the closure of the cover. We have shown that, for complete WSTS, the clover is finite and describes the closure of the cover exactly. When the original WSTS is not complete,

We have also defined a simple procedure, $\mathbf{Clover}_{\mathfrak{S}}$ for computing the clover for ∞-effective complete WSTS $\mathfrak{S}$. This captures the essence of generalized forms of the Karp-Miller procedure, while terminating in more cases. We have shown that that $\mathbf{Clover}_{\mathfrak{S}}$ terminates iff the WSTS is *clover-flattable*, i.e., that it is some form of projection of a flat system, with the same clover. We have also shown that several variants of the notion of clover-flattability were in fact equivalent. We believe that this characterization is an important, and non-trivial result.

In the future, we shall explore efficient strategies for choosing sequences $g \in F^*$ to lub-accelerate in the $\mathbf{Clover}_{\mathfrak{S}}$ procedure. We will also analyze whether $\mathbf{Clover}_{\mathfrak{S}}$ terminates in models such as BVASS [VG05], reconfigurable nets, timed Petri nets [ADMN04], post-self-modifying Petri nets [Val78] and strongly monotonic affine well-structured nets [FMP04]), i.e., whether they are cover-flattable.

One potential use of the clover is in deciding coverability. But the $\mathbf{Clover}_{\mathfrak{S}}$ procedure may fail to terminate. This is in contrast to the Expand, Enlarge and Check forward algorithm of [GRvB07], which always terminates, hence decides coverability. One may want to combine the best of both worlds, and the lub-accelerations of $\mathbf{Clover}_{\mathfrak{S}}$ can profitably be used to improve the efficiency of the Expand, Enlarge and Check algorithm. This remains to be explored.

Finally, recall that computing the finite clover is a first step [EN98] in the direction of solving liveness properties (and not only safety properties which reduce to coverability). We plan to clarify the construction of a *cloverability graph* which would be the basis for liveness model checking.

Acknowledgement. We wish to acknowledge Rémi Bonnet and Sylvain Schmitz for fruitful discussions.

References

[AJ94] Abdulla, P.A., Jonsson, B.: Undecidable Verification Problems for Programs with Unreliable Channels. In: Shamir, E., Abiteboul, S. (eds.) ICALP 1994. LNCS, vol. 820, pp. 327–346. Springer, Heidelberg (1994)

[ABJ98] Abdulla, P., Bouajjani, A., Jonsson, B.: On-The-Fly Analysis of Systems With Unbounded, Lossy Fifo Channels. In: Vardi, M.Y. (ed.) CAV 1998. LNCS, vol. 1427, pp. 305–318. Springer, Heidelberg (1998)

[ADB07] Abdulla, P.A., Delzanno, G., Van Begin, L.: Comparing the Expressive Power of Well-Structured Transition Systems. In: Duparc, J., Henzinger, T.A. (eds.) CSL 2007. LNCS, vol. 4646, pp. 99–114. Springer, Heidelberg (2007)

[AN00] Abdulla, P., Nylén, A.: Better is Better than Well: On Efficient Verification of Infinite-State Systems. In: 14th LICS, pp. 132–140 (2000)

[ACABJ04] Abdulla, P.A., Collomb-Annichini, A., Bouajjani, A., Jonsson, B.: Using forward reachability analysis for verification of lossy channel systems. Formal Methods in System Design 25(1), 39–65 (2004)

[AČJT00] Abdulla, P.A., Čerāns, K., Jonsson, B., Tsay, Y.-K.: Algorithmic analysis of programs with well quasi-ordered domains. Information and Computation 160(1-2), 109–127 (2000)

[ADMN04] Abdulla, P.A., Deneux, J., Mahata, P., Nylén, A.: Forward Reachability Analysis of Timed Petri Nets. In: Lakhnech, Y., Yovine, S. (eds.) FORMATS/FTRTFT 2004. LNCS, vol. 3253, pp. 343–362. Springer, Heidelberg (2004)

[AJ94] Abramsky, S., Jung, A.: Domain theory. In: Abramsky, S., Gabbay, D.M., Maibaum, T.S.E. (eds.) Handbook of Logic in Computer Science, vol. 3, pp. 1–168. Oxford University Press (1994)

[BF12] Bonnet, R., Finkel, A.: Forward Analysis for WSTS: Beyond Regular Accelerations (February 2012) (submitted)

[BFLS05] Bardin, S., Finkel, A., Leroux, J., Schnoebelen, P.: Flat Acceleration in Symbolic Model Checking. In: Peled, D.A., Tsay, Y.-K. (eds.) ATVA 2005. LNCS, vol. 3707, pp. 474–488. Springer, Heidelberg (2005)

[BFHRV11] Bonnet, R., Finkel, A., Haddad, S., Rosa-Velardo, F.: Ordinal Theory for Expressiveness of Well Structured Transition Systems. In: Hofmann, M. (ed.) FOSSACS 2011. LNCS, vol. 6604, pp. 153–167. Springer, Heidelberg (2011)

[BG11] Bozzelli, L., Ganty, P.: Complexity Analysis of the Backward Coverability Algorithm for VASS. In: Delzanno, G., Potapov, I. (eds.) RP 2011. LNCS, vol. 6945, pp. 96–109. Springer, Heidelberg (2011)

[CFP96] Cécé, G., Finkel, A., Purushothaman Iyer, S.: Unreliable channels are easier to verify than perfect channels. Information and Computation 124(1), 20–31 (1996)

[CFS11] Chambart, P., Finkel, A., Schmitz, S.: Forward Analysis and Model Checking for Trace Bounded WSTS. In: Kristensen, L.M., Petrucci, L. (eds.) PETRI NETS 2011. LNCS, vol. 6709, pp. 49–68. Springer, Heidelberg (2011)

[CFS11] Chambart, P., Finkel, A., Schmitz, S.: Forward Analysis and Model Checking for Trace Bounded WSTS. In: Kristensen, L.M., Petrucci, L. (eds.) PETRI NETS 2011. LNCS, vol. 6709, pp. 49–68. Springer, Heidelberg (2011)

[DFGvD06] Demri, S., Finkel, A., Goranko, V., van Drimmelen, G.: Towards a Model-Checker for Counter Systems. In: Graf, S., Zhang, W. (eds.) ATVA 2006. LNCS, vol. 4218, pp. 493–507. Springer, Heidelberg (2006)

[DFS98] Dufourd, C., Finkel, A., Schnoebelen, P.: Reset Nets Between Decidability and Undecidability. In: Larsen, K.G., Skyum, S., Winskel, G. (eds.) ICALP 1998. LNCS, vol. 1443, pp. 103–115. Springer, Heidelberg (1998)

[EFM99] Esparza, J., Finkel, A., Mayr, R.: On the verification of broadcast protocols. In: 14th LICS, pp. 352–359 (1999)

[EN98] Allen Emerson, E., Namjoshi, K.S.: On model-checking for non-deterministic infinite-state systems. In: 13th LICS, pp. 70–80 (1998)

[FFSS11] Figueira, D., Figueira, S., Schmitz, S., Schnoebelen, P.: Ackermannian and Primitive-Recursive Bounds with Dickson's Lemma. In: LICS, pp. 269–278 (2011)

[Fin87] Finkel, A.: A Generalization of the Procedure of Karp and Miller to Well Structured Transition Systems. In: Ottmann, T. (ed.) ICALP 1987. LNCS, vol. 267, pp. 499–508. Springer, Heidelberg (1987)

[Fin90] Finkel, A.: Reduction and covering of infinite reachability trees. Information and Computation 89(2), 144–179 (1990)

[Fin93] Finkel, A.: The Minimal Coverability Graph for Petri Nets. In: Rozenberg, G. (ed.) APN 1993. LNCS, vol. 674, pp. 210–243. Springer, Heidelberg (1993)

[FMP04] Finkel, A., McKenzie, P., Picaronny, C.: A well-structured framework for analysing Petri net extensions. Information and Computation 195(1-2), 1–29 (2004)

[FS01] Finkel, A., Schnoebelen, P.: Well-structured transition systems everywhere? Theoretical Computer Science 256(1-2), 63–92 (2001)

[FG09a] Finkel, A., Goubault-Larrecq, J.: Forward analysis for WSTS, part I: Completions. In: Albers, S., Marion, J.-Y. (eds.) Proceedings of the 26th Annual Symposium on Theoretical Aspects of Computer Science (STACS 2009). Leibniz International Proceedings in Informatics, vol. 3, pp. 433–444. Leibniz-Zentrum für Informatik, Freiburg (2009)

[FG09b] Finkel, A., Goubault-Larrecq, J.: Forward Analysis for WSTS, Part II: Complete WSTS. In: Albers, S., Marchetti-Spaccamela, A., Matias, Y., Nikoletseas, S., Thomas, W. (eds.) ICALP 2009. LNCS, vol. 5556, pp. 188–199. Springer, Heidelberg (2009)

[FG12a] Finkel, A., Goubault-Larrecq, J.: Forward analysis for WSTS, part I: Completions (in preparation, 2012); Journal version of [FG09a]

[FG12b] Finkel, A., Goubault-Larrecq, J.: Forward analysis for WSTS, Part II: Complete WSTS (in preparation, 2012); Journal version of [FG09b]

[GRVB07] Geeraerts, G., Raskin, J.-F., Van Begin, L.: Well-structured languages. Acta Inf. 44(3-4), 249–288 (2007)

[GHK$^+$03] Gierz, G., Hofmann, K.H., Keimel, K., Lawson, J.D., Mislove, M., Scott, D.S.: Continuous lattices and domains. In: Encyclopedia of Mathematics and its Applications, vol. 93. Cambridge University Press (2003)

[GRvB06a] Ganty, P., Raskin, J.-F., Van Begin, L.: A Complete Abstract Interpretation Framework for Coverability Properties of WSTS. In: Allen Emerson, E., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 49–64. Springer, Heidelberg (2005)

[GRvB06b] Geeraerts, G., Raskin, J.-F., van Begin, L.: Expand, enlarge and check: New algorithms for the coverability problem of WSTS. J. Comp. and System Sciences 72(1), 180–203 (2006)

[GRvB07] Geeraerts, G., Raskin, J.-F., Van Begin, L.: On the Efficient Computation of the Minimal Coverability Set for Petri Nets. In: Namjoshi, K.S., Yoneda, T., Higashino, T., Okamura, Y. (eds.) ATVA 2007. LNCS, vol. 4762, pp. 98–113. Springer, Heidelberg (2007)

[GS64] Ginsburg, S., Spanier, E.H.: Bounded Algol-like languages. Trans. American Mathematical Society 113(2), 333–368 (1964)

[HP79] Hopcroft, J., Pansiot, J.-J.: On the reachability problem for 5-dimensional vector addition systems. Theoretical Computer Science 8, 135–159 (1979)

[Jan99] Jančar, P.: A note on well quasi-orderings for powersets. Information Processing Letters 72(5-6), 155–160 (1999)

[KM69] Karp, R.M., Miller, R.E.: Parallel program schemata. J. Comp. and System Sciences 3(2), 147–195 (1969)

[KS96] Kouchnarenko, O., Schnoebelen, P.: A model for recursive-parallel programs. Electr. Notes Theor. Comput. Sci. 5, 30 pages (1996)

[LNORW07] Lazić, R.S., Newcomb, T., Ouaknine, J., Roscoe, A.W., Worrell, J.B.: Nets with Tokens Which Carry Data. In: Kleijn, J., Yakovlev, A. (eds.) ICATPN 2007. LNCS, vol. 4546, pp. 301–320. Springer, Heidelberg (2007)

[Mar94] Marcone, A.: Foundations of BQO theory. Trans. Amer. Math. Soc. 345(2), 641–660 (1994)

[May03a] Mayr, R.: Undecidable problems in unreliable computations. Theor. Comput. Sci. 297(1-3), 337–354 (2003)

[May03b] Mayr, R.: Undecidable problems in unreliable computations. Theoretical Computer Science 297(1-3), 337–354 (2003)

[MM81] Mayr, E.W., Meyer, A.R.: The complexity of the finite containment problem for petri nets. J. ACM 28(3), 561–576 (1981)

[Rac78] Rackoff, C.: The covering and boundedness problems for vector addition systems. Theor. Comput. Sci. 6, 223–231 (1978)

[RMF11] Rosa-Velardo, F., Martos-Salgado, M., de Frutos-Escrig, D.: Accelerations for the Coverability Set of Petri Nets with Names. Fundam. Inform. 113(3-4), 313–341 (2011)

[RS04] Robertson, N., Seymour, P.D.: Graph minors. XX. Wagner's conjecture. Journal of Combinatorial Theory, Series B 92(2), 325–357 (2004)

[Sch01] Schnoebelen, P.: Bisimulation and Other Undecidable Equivalences for Lossy Channel Systems. In: Kobayashi, N., Pierce, B.C. (eds.) TACS 2001. LNCS, vol. 2215, pp. 385–399. Springer, Heidelberg (2001)

[SS11] Schmitz, S., Schnoebelen, P.: Multiply-Recursive Upper Bounds with Higman's Lemma. In: Aceto, L., Henzinger, M., Sgall, J. (eds.) ICALP 2011, Part II. LNCS, vol. 6756, pp. 441–452. Springer, Heidelberg (2011)

[Val78] Valk, R.: Self-Modidying Nets, a Natural Extension of Petri Nets. In: Ausiello, G., Böhm, C. (eds.) ICALP 1978. LNCS, vol. 62, pp. 464–476. Springer, Heidelberg (1978)

[VF07] Rosa-Velardo, F., de Frutos-Escrig, D.: Name Creation vs. Replication in Petri Net Systems. In: Kleijn, J., Yakovlev, A. (eds.) ICATPN 2007. LNCS, vol. 4546, pp. 402–422. Springer, Heidelberg (2007)

[VG05] Verma, K.N., Goubault-Larrecq, J.: Karp-Miller trees for a branching extension of VASS. Discrete Mathematics & Theoretical Computer Science 7(1), 217–230 (2005)

[WZH10] Wies, T., Zufferey, D., Henzinger, T.A.: Forward Analysis of Depth-Bounded Processes. In: Ong, L. (ed.) FOSSACS 2010. LNCS, vol. 6014, pp. 94–108. Springer, Heidelberg (2010)

[ZWH12] Zufferey, D., Wies, T., Henzinger, T.A.: Ideal Abstractions for Well-Structured Transition Systems. In: Kuncak, V., Rybalchenko, A. (eds.) VMCAI 2012. LNCS, vol. 7148, pp. 445–460. Springer, Heidelberg (2012)

Using Colored Petri Nets to Construct Coalescent Hidden Markov Models: Automatic Translation from Demographic Specifications to Efficient Inference Methods

Thomas Mailund[1], Anders E. Halager[1,2], and Michael Westergaard[3]

[1] Bioinformatics Research Center, Aarhus University, Denmark
[2] Department of Computer Science, Aarhus University, Denmark
[3] Department of Mathematics and Computer Science, Eindhoven University of Technology, The Netherlands

Abstract. Biotechnological improvements over the last decade has made it economically and technologically feasible to collect large DNA sequence data from many closely related species. This enables us to study the detailed evolutionary history of recent speciation and demographics. Sophisticated statistical methods are needed, however, to extract the information that DNA sequences hold, and a limiting factor in this is dealing with the large state space that the ancestry of large DNA sequences spans. Recently a new analysis method, CoalHMMs, has been developed, that makes it computationally feasible to scan full genome sequences – the complete genetic information of a species – and extract genetic histories from this. Applying this methodology, however, requires that the full state space of ancestral histories can be constructed. This is not feasible to do manually, but by applying formal methods such as Petri nets it is possible to build sophisticated evolutionary histories and automatically derive the analysis models needed. In this paper we describe how to use colored stochastic Petri nets to build CoalHMMs for complex demographic scenarios.

1 Introduction

Biotechnological advances over the last decade have dramatically reduced the cost of obtaining the full genetic material of an individual – the full genome – and genomes from many closely related species are now available. For example, one or more genomes have been sequenced for each species of the great apes, the closest related species to humans. This puts us in the unique position to learn much more about human evolutionary history over the last 15 million years than what has previously been gleaned from fossils and from single gene studies.

Computational approaches to studying biology enables sophisticated analysis and provide the only feasible approach to analysis for very large data sets, such as full genome sequences. Whole genome comparisons hold the key to decipher the speciation process, selection and demographic changes in human and great ape history, but analysis methods that are statistical powerful and computationally efficient are still in their infancy.

Sequential Markov Coalescent (SMC) and its inference method Coalescent hidden Markov models (CoalHMMs) [5,8,9,12,18,22,24] is a recently developed methodology for analyzing genome relationships and make inference of speciation divergence and the mechanisms involved in speciation. CoalHMMs combine the so-called "coalescence process" model of population genetics [11] with the computational efficient statistical tool "hidden Markov models" [7] and provides the first approach to analyze the speciation process computationally scalable to whole-genome analysis. CoalHMMs model the dependence of the genealogies (tree relationships) between neighboring nucleotides along a genomic sequence as a function of the events of coalescence and recombination in the history of the sequences, and can analyze samples of entire genomes appropriately aligned.

The first CoalHMMs were designed to estimate split times and genetic diversity in the species ancestral to human, chimpanzee and gorilla by analyzing patterns of incomplete lineage sorting – i.e. patterns of genealogies inconsistent with the species phylogeny caused by deep coalescences [8,12]. The same models were later used to analyze the complete orangutan genome [17] and gave insight into the evolutionary forces forming the great ape genomes [13]. The models have also been applied to the gorilla genome [29] and bonobo genome [25] further illuminating the evolution of our own species by comparing our genome with our closest ape relatives. A different approach to CoalHMMs was recently used to infer demographic parameters of the human species [16].

A major limitation of the initial CoalHMM methods is that they do not generalize to comprise complex demographic and speciation scenarios. The methods strongly depend on patterns of incomplete lineage sorting and do not allow for complex population structures, population size bottlenecks, gene flow etc. This was amended in the method used for analyzing the orangutan subspecies [18]. Here, a mathematical model based on continuous time Markov chains (CTMCs) was used to explicitly model the probability of changes in genealogies along a genome sequence, using exact calculations from the coalescence process. While this first CTMC based method is rather simple, only capturing changes in coalescence time between two genomes, the strengths of the CTMC approach is that, in theory, it generalizes to a large variety of scenarios.

Constructing CTMC models of complex demographic scenarios, however, is at best tedious and error prone, considering the large state space of these models, and it is unlikely that they can be constructed correctly by hand. In this paper we propose using colored Petri nets (CPN) [15] as a formal method for specifying genetic models and give algorithms for translating the state space of such models into CoalHMMs.

In the next section, we will provide background information for the coalescent process and present a CPN model of the coalescence process. In the following section we describe how the coalescent CPN model can be used to define a Markov model along a genome, approximating the coalescence process. This is similar to the construction of a Markov chain from a stochastic Petri net [21].

We then present results from our prototype implementation, and finally draw our conclusions.

2 Modeling the Coalescent Process

The general idea behind coalescent hidden Markov models is to approximate the coalescent process by a Markov model along a genomic alignment. Below we first present the coalescence process and then present a CPN model of the coalescence process over two neighboring nucleotides.

2.1 The Coalescent Process

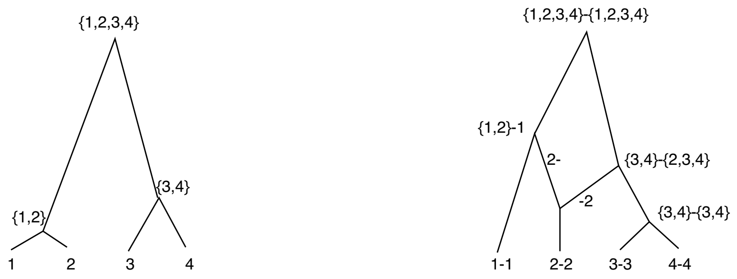
The coalescent process [11] is a statistical model describing the genetic relationship of a sample of genes. The coalescent process assumes that k genes have been sampled in a population, and models how their ancestry (or "genealogy") could be, providing probabilities to different scenarios of the genes ancestry from which a number of properties of the population can be inferred.

The process runs backwards in time, and in its simplest form each pair of genes can coalesce with a fixed rate. When two genes coalesce, it models the time where they last shared an ancestor (known as the most recent common ancestor, or MRCA, of the two genes). After a pair of genes have coalesced, they are replaced by a gene representing their MRCA, and the process continues further back in time, now with $k-1$ genes. The process is continued until all genes have coalesced, i.e. when $k=1$. A run of the coalescent process corresponds to a tree, where the order in which different genes coalesce determines the topology and the time in which genes coalesce determines the branch lengths, see Fig. 1 (a).

By treating the process as a continuous time Markov chain, each tree can be assigned a probability, and by placing mutations on the tree we can compute the probability that a given tree gave rise to the observed genes. From this we can get the joint probability of the tree and the observed genes, and use this to make statistical inference. Since the true ancestry of genes is unknown, and in general unknowable, statistical inference based on the coalescence process involves integrating over all trees, either explicitly (for small k) or with statistical Monte Carlo integration (for larger k).

A simple tree relationship for genes, however, is an inaccurate model of species with two genders. Sex cells in species with two genders are constructed as "recombinations" of the genetic material inherited by each parent. In the coalescent process, this is modeled by adding a second type of event. Each gene can undergo recombination, in which case the gene is split in two at a random point, the left and right side of the recombination point. The process is then continues with $k+1$ genes as the left and right part of the recombined gene is assumed to have independent ancestries.

A run of the coalescent process with recombination can no longer be represented as a tree but instead a directed acyclic graph, known as the *ancestral recombination graph* or ARG, see Fig. 1 (b). Scanning from left to right along



(a) A coalescence tree over a sample of size 4

(b) An ancestral recombination graph over a sample of size 4 of sequences of length 2

Fig. 1. A coalescence tree and an ancestral recombination graph. (a) A coalescence tree, where first genes 1 and 2 coalesce into their most recent common ancestor, $\{1,2\}$, then genes 3 and 4 coalesce into their most recent common ancestor $\{3,4\}$ and finally all genes coalesce into the grand most recent common ancestor. (b) An ancestral recombination graph of four sequences of length two. First genes 3 and 4 coalesce, where both their left and right nucleotide find an ancestor at the same time. Then gene 2 recombines, leading to independent genes for its left and right nucleotide. The right nucleotide of gene 2 coalesce with the ancestor of genes 3 and 4 while the left nucleotide of gene 2 coalesce with gene 1, before all genes find their most recent common ancestor. The left and right nucleotide in this example have different genealogies, with the left having topology $((1,2),(3,4))$ and the right having topology $(1,(2,(3,4)))$.

the sampled genes, at each point the ancestry of the genes will be a tree, but the trees can change whenever a recombination point is seen. The tree at each point is known as a local genealogy while the ARG is known as the global genealogy of the genes.

The state space of possible ARGs for a gene sample is generally intractable for all but the smallest samples [30] even for statistical integration, and to deal with large sample sizes or long gene sequences approximations to the process is necessary. One such approximation is assuming that the relationship between genealogies is Markov along the genes [1, 20], an assumption that greatly reduces the complexity of the process. Assuming the Markov property essentially means that we only need to model pairs of nucleotides rather than the full DNA sequence, since the probability of a sequence can be specified through all the pairwise probabilities.

2.2 A Colored Petri Net Model for Pairwise Genealogies

While the coalescence process is difficult to make inference from, the rules for how the process generates genealogies are straightforward and can be expressed as a very simple colored Petri net. The way the coalescence process treats genes as independent items with events that can affect one or two genes maps straightforwardly to a CPN model where genes become tokens and coalescence and

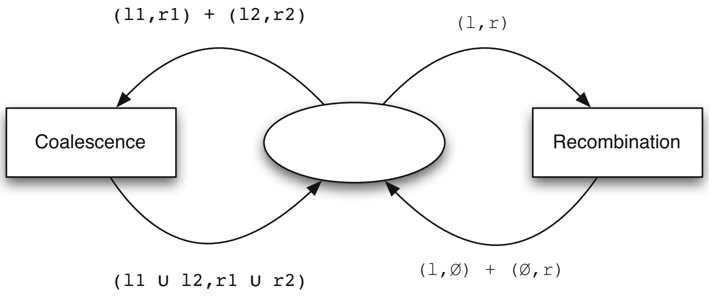


Fig. 2. CPN model of the basic two-nucleotide coalescence. This rather simple colored Petri net can construct all two-nucleotide coalescence runs for any number of samples in a single population. The set of genes in the process are represented as tokens on the single place, where each token contains a pair of sets of sampled genes. The pair represent the left and right nucleotide in the gene, and the sets the genes or most recent common ancestor of a set of genes. A coalescence event combines the left and right sets of the genes, while a recombination breaks up one gene into two: the left and right nucleotide of the original gene.

recombination events become transitions. Such a CPN model is shown in Fig. 2. The CPN model has a single place, containing the genes of the process, and two transitions modeling the two operations Coalescence and Recombination. The tokens on the single place consists of pairs – the left and right nucleotide of the genes – and each nucleotide will contain the set of original sampled genes. The initial marking consists of pairs $(\{i\},\{i\})$ for genes $i = 1,\ldots,k$

A run of this CPN, producing the ARG in Fig. 1 (b), would look like this:

State:	$1`(\{1\},\{1\}) + 1`(\{2\},\{2\}) + 1`(\{3\},\{3\}) + 1`(\{4\},\{4\})$
Binding:	$[\mathsf{Coalescence}; 1`(\{3\},\{3\}) + 1`(\{4\},\{4\})\rangle$
State:	$1`(\{1\},\{1\}) + 1`(\{2\},\{2\}) + 1`(\{3,4\},\{3,4\})$
Binding:	$[\mathsf{Recombination}; 1`(\{2\},\{2\})\rangle$
State:	$1`(\{1\},\{1\}) + 1`(\{2\},\emptyset) + 1`(\emptyset,\{2\}) + 1`(\{3,4\},\{3,4\})$
Binding	$[\mathsf{Coalescence}; 1`(\{3,4\},\{3,4\}) + 1`(\emptyset,\{2\})\rangle$
State	$1`(\{1\},\{1\}) + 1`(\{2\},\emptyset) + 1`(\{3,4\},\{2,3,4\})$
Binding:	$[\mathsf{Coalescence}; 1`(\{1\},\{1\}) + 1`(\{2\},\emptyset)\rangle$
State	$1`(\{1,2\},\{1\}) + 1`(\{3,4\},\{2,3,4\})$
Binding:	$[\mathsf{Coalescence}; 1`(\{1,2\},\{1\}) + 1`(\{3,4\},\{2,3,4\})\rangle$
State:	$1`(\{1,2,3,4\},\{1,2,3,4\})$

When we have different populations or different species, the probability of coalescing genes in different populations/species is zero, and we cannot model genes in this simple way. To model this, we can annotate tokens with populations and only allow Coalescence and Recombination to affect genes within a single population, but instead add a new event that migrates a gene from one population to another, see Fig. 3.

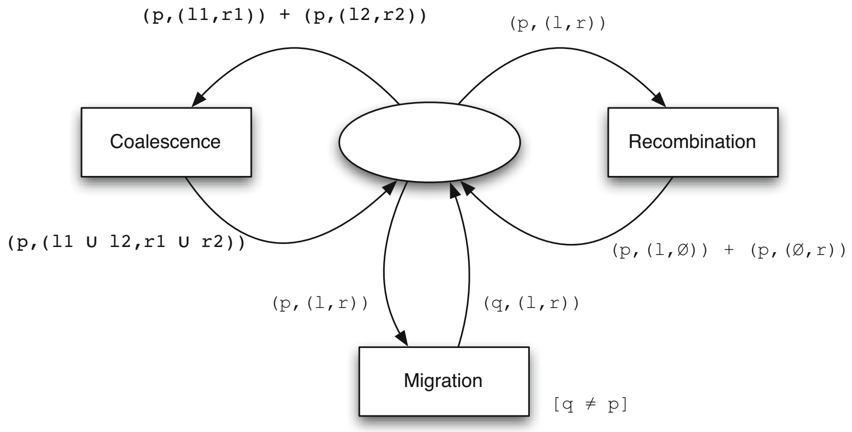


Fig. 3. CPN model with migration. To model different populations, we annotate each token with the population it belongs to. Coalescence events are only possible between lineages in the same population. Recombination, as well, although this only involves a single lineage so the difference is only seen in the arc annotation. To allow lineages to move from one population to another, a new transition is added that moves one lineage from one population to another.

2.3 Building Coalescent CTMCs from the Petri Net Specification

From the CPN specification we can build a state space capturing all possible ancestries of a sample. Our goal is to assign probabilities to all such ancestries. To do this, we consider the process a continuous time Markov chain (CTMC), and build the complete state space graph of the system. This corresponds to a matrix of rates between states where the rate between states is given by the type of transition in the CPN, and is similar to how a Markov chain is constructed from a Stochastic Petri net. We cannot use vanilla stochastic Petri nets, though as the transition rates depend also on the binding of the variables and we need to keep it symbolic for future estimation.

In terms of CTMC theory, what we construct is the instantaneous transition matrix, usually denoted Q and from this we can derive the probability of any run of the system. Obtaining a probabilistic model of the ancestries of a sample thus involves building the complete state space of the CPN model, translating this into a matrix of rates of transitions and considering this a CTMC rate matrix. For samples from a single population, we assign a fixed rate to transitions and recombinations (see Fig. 2), while for a scenario with multiple populations, we allow different coalescence rates for each population and different migration rates between different pairs of populations.

3 Constructing Sequential Markov Coalescent Models

The computational efficiency of CoalHMMs stems from assuming that the probability distribution of genealogies along a genome alignment is Markov: The probability of a local genealogy depends on its immediate neighbor, but not the more distant genealogies [1,20,22]. This way, the probability of a genealogy of the entire alignment can be specified from just the probability distribution of genealogies of two neighboring nucleotides [8,18].

Let $\Pr(\mathcal{G}_L, \mathcal{G}_R)$ denote the joint probability of genealogies, $\mathcal{G}_L$ and $\mathcal{G}_R$ of two nucleotides L and R (left and right). If this probability can be efficiently computed, then the probability of a genealogy over L nucleotides, $\Pr(\mathcal{G}_1, \mathcal{G}_2, \ldots, \mathcal{G}_L)$ can efficiently be computed as $\Pr(\mathcal{G}_1, \mathcal{G}_2, \ldots, \mathcal{G}_L) = \Pr(\mathcal{G}_1) \prod_{i=1}^{L-1} \Pr(\mathcal{G}_{i+1} \mid \mathcal{G}_i)$ where $\Pr(\mathcal{G}_1) = \sum_g \Pr(\mathcal{G}_1, g)$ and $\Pr(\mathcal{G}_{i+1} \mid \mathcal{G}_i) = \Pr(\mathcal{G}_i, \mathcal{G}_{i+1}) / \Pr(\mathcal{G}_i)$.

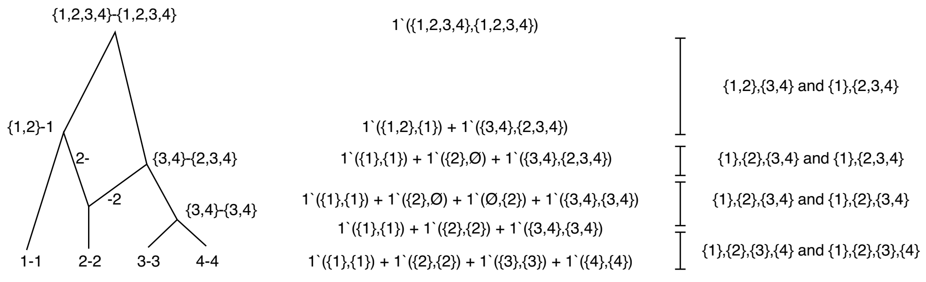
The key idea in Mailund *et al.* [18], that we generalize in this paper, was that these joint probabilities can be computed from a two-nucleotide CTMC. We can explicitly enumerate all possible states and state changes in the ancestry of two neighboring nucleotides, construct the corresponding CTMC, and obtain probabilities from this. Constructing the CTMC manually is feasible for small systems, as in Mailund *et al.* [18], but quickly becomes unmanageable. Below we show how the system can be constructed from a colored Petri net, and how the joint probability of a pair of genealogies can be algorithmically constructed from this.

To fully specify a CoalHMM we need to specify both transition and emission matrices (see Appendix A for the formal specification of a hidden Markov model), but since emission matrices can be computed using standard bioinformatics techniques, we will only focus on the transition matrix here. Constructing the transition matrix for the CoalHMM from the CTMC involves two steps: projecting a run of the CTMC onto the two neighboring genealogies, and discretizing time into time-intervals.

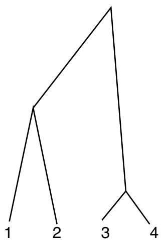
3.1 Projecting Runs of the CPN Model onto Pairs of Genealogies

A run of the CTMC involves coalescence events, recombination events and migration events. Of these, only coalescence events, where two lineages find a MRCA, are observable in the genealogies. All other events are important for computing the probability of the genealogies, but only the times of MRCAs are directly observable as genealogies and all other events should be integrated out when the probabilities of genealogies are computed.

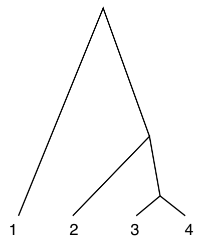
The time points where two lineages find their MRCA corresponds to transitions in the CTMC state space where the system moves from one strongly connected component (SCC) to another since both migration and recombination events are reversible through a migration back or a coalescence of the two genes recombined, respectively. The genealogies of interest thus correspond to the paths in the SCC graph of the CTMC state space. Enumerating all paths in the SCC graph thus gives us all the genealogies to be considered, and there is a one-to-one correspondence between pairs of genealogies and paths in the SCC


{1,2,3,4}-{1,2,3,4}
{1,2}-1
2-
-2
{3,4}-{2,3,4}
{3,4}-{3,4}
1-1
2-2
3-3
4-4
1`({1,2,3,4},{1,2,3,4})
1`({1,2},{1}) + 1`({3,4},{2,3,4})
1`({1},{1}) + 1`({2},Ø) + 1`({3,4},{2,3,4})
1`({1},{1}) + 1`({2},Ø) + 1`(Ø,{2}) + 1`({3,4},{3,4})
1`({1},{1}) + 1`({2},{2}) + 1`({3,4},{3,4})
1`({1},{1}) + 1`({2},{2}) + 1`({3},{3}) + 1`({4},{4})
{1,2},{3,4} and {1},{2,3,4}
{1},{2},{3,4} and {1},{2,3,4}
{1},{2},{3,4} and {1},{2},{3,4}
{1},{2},{3},{4} and {1},{2},{3},{4}

(a) ARG and corresponding states and SCCs



(b) Left genealogy



(c) Right genealogy

Fig. 4. ARG, state space and strongly connected components of a run. (a) On the left the ARG from Fig. 1 (b). In the middle the CPN states corresponding to this ARG. On the right, the strongly connected components corresponding to this run of the CPN. The SCC is represented by the coalesced lineages on the left and right, respectively, and does not change due to the recombination event on the ARG. (b) The left genealogy of the ARG. (c) The right genealogy of the ARG.

graph. Fig. 4 shows a run of the CPN of a coalescent system. Here an ARG (from Fig. 1) is shown together with the states of the CPN that can produce this system, the SCC run of the system and the left and right genealogies of this run.

Paths in the SCC graph corresponds to pairs of genealogies and will be the state transitions in the hidden Markov model we construct. To exploit the efficient algorithms for HMMs we need to project the infinite state space of SCC paths onto a finite state space. We do this by discretizing time into a finite, fixed set of non-overlapping time intervals, $[\tau_0, \tau_1]$, $[\tau_1, \tau_2]$, up to $[\tau_{n-1}, \tau_n]$ with $\tau_0 = 0$ and $\tau_n = \infty$. We obtain finite state spaces by only considering which state the system is in at the time points between these intervals $(\tau_1, \tau_2, \ldots, \tau_{n-1})$.

We combine the discretized time with the valid SCC runs as follows. For any path through the SCC graph, $c_1, c_2, \ldots, c_n$, we assign time points to components $\tau_1 \leftrightarrow c_{i_1}, \tau_2 \leftrightarrow c_{i_2}, \ldots \tau_{n-1} \leftrightarrow c_{i_{n-1}}$ where $i_j \leq i_k$ for $j \leq k$. So, as an example, with three time intervals $[\tau_0 = 0, \tau_1]$, $[\tau_1, \tau_2]$ and $[\tau_2, \tau_3 = \infty]$ and an SCC path with two components, c_1, c_2, we would get the following three timed paths:

Interval	τ_1	Interval	τ_2	Interval
$[\tau_0, \tau_1]$	c_1	$[\tau_1, \tau_2]$	c_1	$[\tau_2, \tau_3]$
$[\tau_0, \tau_1]$	c_1	$[\tau_1, \tau_2]$	c_2	$[\tau_2, \tau_3]$
$[\tau_0, \tau_1]$	c_2	$[\tau_1, \tau_2]$	c_2	$[\tau_2, \tau_3]$

Notice that not all components need to be assigned a time point, and some can be assigned to several. This reflects that the system can move through several components within a single time interval and also stay in one component over several time intervals.

CTMC theory provides us with the mechanism for integrating over all paths leading from one state to another. If Q denotes the instantaneous rate matrix of the CTMC, then the probability of being in state s at time τ_i and state t at time τ_{i+1} is given by $P_{i,j}^{\tau_{i+1}-\tau_i}$ where $P^{\tau_{i+1}-\tau_i} = \exp\left(Q\left[\tau_{i+1}-\tau_i\right]\right)$ (where $\exp\left(M\right)$ denotes matrix exponentiation [23]). The probability of being in SCC c_i at time τ_i and SCC c_j at time τ_{i+1} is then computed by summing over all transitions from a state $s \in c_i$ to a state $t \in c_j$ in the time interval $[\tau_i, \tau_{i+1}]$: $\sum_{s\in c_i}\sum_{t\in c_j} P_{s,t}^{\tau_{i+1}-\tau_i}$. The probability of an entire SCC path assigned to time intervals is obtained by summing across all time intervals in this way (see Fig. 5), e.g. for the example above:

$$\Pr\left([\tau_0,\tau_1]\ c_1\ [\tau_1,\tau_2]\ c_1\ [\tau_2,\tau_3]\right) = \sum_{s\in c_1}\sum_{t\in c_1} P_{\iota,s}^{\tau_1}\cdot P_{s,t}^{\tau_2-\tau_1}$$

$$\Pr\left([\tau_0,\tau_1]\ c_1\ [\tau_1,\tau_2]\ c_2\ [\tau_2,\tau_3]\right) = \sum_{s\in c_1}\sum_{t\in c_2} P_{\iota,s}^{\tau_1}\cdot P_{s,t}^{\tau_2-\tau_1}$$

and

$$\Pr\left([\tau_0,\tau_1]\ c_2\ [\tau_1,\tau_2]\ c_2\ [\tau_2,\tau_3]\right) = \sum_{s\in c_2}\sum_{t\in c_2} P_{\iota,s}^{\tau_1}\cdot P_{s,t}^{\tau_2-\tau_1}$$

(notice changes in subscripts) where we assume that the system always starts in a fixed state ι.

For the general case $[\tau_0,\tau_1]\ c_{i_1}\ [\tau_1,\tau_2]\ c_{i_2}\ldots[\tau_{n-2},\tau_{n-1}]\ c_{i_{n-1}}\ [\tau_{n-1},\tau_n]$ this becomes

$$\sum_{s_1\in c_{i_1}}\sum_{s_2\in c_{i_2}}\cdots\sum_{s_{n-1}\in c_{i_{n-1}}} P_{\iota,s_1}^{\tau_1}\cdot P_{s_{i_1},s_{i_2}}^{\tau_2-\tau_1}\cdots P_{s_{n-2},s_{n-1}}^{\tau_{n-1}-\tau_{n-2}}$$

which is a sum of $|c_{i_1}|\times|c_{i_2}|\times\cdots\times|c_{i_{n-1}}|$ terms, where each term is a product of $n-1$ transition probabilities. To efficiently compute this for all paths, we rewrite this to

$$\sum_{s_1\in c_{i_1}} P_{\iota,s_1}^{\tau_1}\left(\sum_{s_2\in c_{i_2}} P_{s_{i_1},s_{i_2}}^{\tau_2-\tau_1}\left(\cdots\left(\sum_{s_{n-1}\in c_{i_{n-1}}} P_{s_{n-2},s_{n-1}}^{\tau_{n-1}-\tau_{n-2}}\right)\right)\cdots\right)$$

which we can compute inside-out for all paths using dynamic programming.

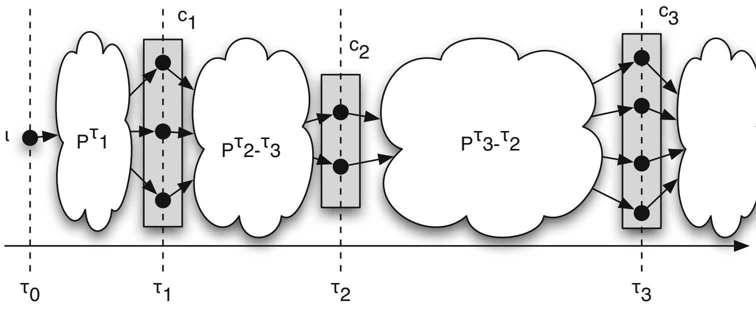


Fig. 5. Computing path probabilities. When computing the probability of the timed path $[\tau_0, \tau_1]\ c_1\ [\tau_1, \tau_2]\ c_2\ [\tau_2, \tau_3]\ c_3\ [\tau_3, \tau_4]$ we implicitly sum over all paths between the time interval breakpoints using CTMC transition probability matrices $P^{\tau_{i+1}-\tau_i}$ and explicitly sum over states in the strongly connected components at the breakpoints c_i.

3.2 Dealing with Different Demographic Epochs

When modeling the history of a set of genomes from different species, we need to consider different time period of their history. Consider for example a model of the ancestry of three different species, e.g. humans, chimpanzees and gorillas. At present, these are three different species that cannot exchange genes, but as we go back in time we first enter a period where humans and chimpanzees share an ancestral species, where they can exchange genes, and further back in time all three species share an ancestor where they exchange genes.

To deal with this we use what we called different "epochs". Each epoch corresponds to separate model in terms of transitions and transition rates, but all epochs for the same analysis can be embedded in the same (often large) state space, enabling us to map states between them. For the human, chimpanzee and gorilla example, we would have three populations/species and one sample from each. So the type used for lineages would have three colors (e.g. H, C and G for human, chimpanzee and gorilla) and the type used for populations also three colors. The space of all possible states would be all states that the CPN could be in. The different epochs would consist of restrictions to this state space, and typically we would never enumerate the full state space but only the sub-state spaces reachable in the different epochs.

A simple human, chimpanzee and gorilla model could have three epochs, one where all three species are isolated, one where humans and chimpanzees have found a common ancestor and one where all three African apes have found a common ancestor. This model will not allow migrations in any epochs. The first epoch will have each species in its own population, the second epoch would have humans and chimpanzees in the same population, and the third epoch would have all three species in the same population.

We construct the model by first constructing the state space of the first epoch, where the populations are H, C and G. We then take all reachable states in this system and maps H and C tokens to the same population, e.g. H, so tokens are mapped $(p,(l,r)) \mapsto (\mathsf{H},(l,r))$ whenever p is H or C. For the second epoch, we compute the state space of all states reachable from these mapped states (but not states from the first epoch where tokens can be in population C). For the third epoch we repeat this, but now mapping G populations to H as well.

When computing the probability of paths in the system, we add this projection of states as well. If the time point τ_i is between two different epochs, we use a matrix $P^{\tau_i - \tau_{i-1}} \cdot I_i$ instead of $P^{\tau_i - \tau_{i-1}}$ where I_i is a projection matrix mapping states from the epoch before τ_i to the epoch after τ_i. For the transition between the first and second epoch in the human, chimpanzee and gorilla example, this projection matrix would have a 1 in all entries where the states are equal exact for all C populations being set to H and 0 in all other entries, and for the projection from the second to the third epoch, the projection matrix would have a 1 in entries where the states are equal except that now G populations are set to H as well. The projection onto left and right genealogies, and the sums used for computing the probabilities of strongly connected components paths is not changed otherwise.

4 Results

The algorithm was implemented in the Python programming language and below we show results for state space construction, model fit and parameter estimation.

4.1 State Space Statistics

We constructed the state space and HMM transition matrix for a number of different configurations, varying the number of populations from one to three and varying the number of chromosomes from one to four. With one population there is a single time epoch, with two populations there are two epochs, one before and after the populations merge, and with three populations there are three time epochs: the first before any populations merge, the second after the first and second population merge, and the last when all three populations have merged.

Table 1 shows the size of the state spaces in the various configurations and epochs and the time it takes to construct the HMM transition matrix. The HMM construction time is split in three components: 1) the time it takes to construct the CTMC (i.e. build the state space of the CPN and translate it into a rate matrix), 2) pre-processing time for the HMM construct, involving building the SCC graph and assign all possible SCC paths to time intervals, and 3) the time it takes to construct the actual transition matrix, involving exponentiating rate matrices and summing over SCC paths. Of these three, the first two needs only be computed once for a given model, while the third needs to be recomputed whenever the parameters of the HMM changes, and must

Table 1. Summaries of the state space sizes, SCCs and construction time for both the state space and the hidden Markov model transition matrix. Configurations $n = i, j, k$ should be read as population one containing i chromosomes, population two containing j chromosomes and population three containing k chromosomes. Construction time is measured in seconds and – indicates that the computation was terminated before finishing.

	1st epoch		2nd epoch		3rd epoch		Construction time		
Configuration	States	SCCs	States	SCCs	States	SCCs	CTMC	Pre.	Trans.
1 population									
n = 1	2	1					0.00	0.00	0.01
n = 2	15	4					0.00	0.01	0.09
n = 3	203	25					0.03	3.44	18.02
n = 4	4 140	225					1.35	–	–
2 populations									
n = 1,1	4	1	15	4			0.00	0.02	0.08
n = 2,1	30	4	203	25			0.03	14.60	24.90
n = 3,1	406	25	203	25			1.71	–	–
n = 2,2	225	16	4 140	225			1.49	–	–
3 populations									
n = 1,1,1	8	1	30	4	203	25	0.11	19.75	21.87
n = 2,1,1	60	4	306	25	4 140	225	1.67	–	–

potentially be computed hundreds of times in a numerical optimization of the HMM likelihood.

The most time consuming part of constructing the HMM is clearly not constructing the state space of the model but rather the alignment of the SCC graph onto time intervals for constructing the HMM states and the exponentiation of rate matrices for computing transition probabilities. The configurations in Table 1 where the construction time is missing were terminated after hours of run-time indicating a very steep exponential growth in running time as the size of the system grows.

4.2 Parameter Estimation

As an evaluation of the model, we consider the so-called *isolation-with-migration* model of speciation, see Fig. 6. In this model, the speciation is initiated by a split in an ancestral species into two groups who evolve independently but exchange genes through limited migration, until at some later point this gene-flow ends and the species evolve completely independent. For most apes, this is believed to be the process in which they separated; the two gorilla species alive today are believed to have split about a million years ago but exchanged genes until recently [29,31], the two orangutan sub-species has a similar story [17], and even humans have exchanged genes with archaic forms of humans, such as Neanderthals [10] and the recently discovered Denisovan humans [27,28].

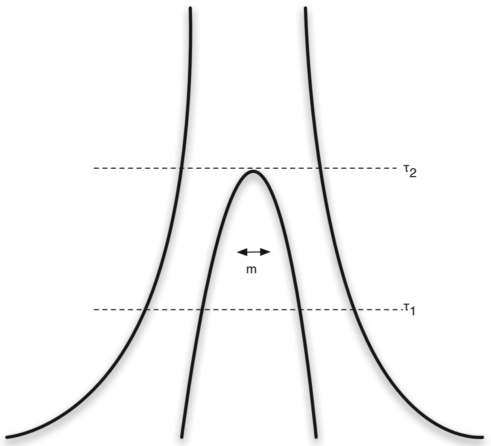


Fig. 6. Isolation-with-migration model. A model of the speciation process, where an ancestral species is initially split into two populations that exchange genes through migration between the groups for a period of time, after which gene flow stops and the species evolve independently.

Relevant parameters in this model include the initial split time, the time when gene-flow ended, the migration rate between the ancestral populations and the coalescence rate in the ancestral species (which measures the genetic variation in the ancestral species). To test our model in this scenario, we simulated data using the coalescence simulator CoaSim [19] and then estimated the parameters with our CoalHMM. Results are shown in Fig. 7.

Although there naturally is some variation in the estimated parameters, we find that the model accurately estimates the parameters of the simulated data.

5 Discussion

We have presented a method for building inference models for complex demographic histories of speciation and genome ancestry. The method 1) employs a colored Petri net to specify the demographic scenario, constructs the state space for the scenario, 2) uses this as a continuous time Markov chain to compute probabilities of genealogies, 3) uses the strongly connected component graph of the state space to compute transition probabilities between local genealogies, and finally 4) uses these transition probabilities to construct a coalescent hidden Markov model for inferring parameters of the scenario.

The approach we have presented fully automates the translation of a demographic scenario to an inference method, making CoalHMMs accessible to biologists with limited computational skills, but the complexity of scenarios that can be explored is still limited by the state space explosion in the model and the computational time needed for enumerating all SCC paths and for exponentiating large rate matrices.

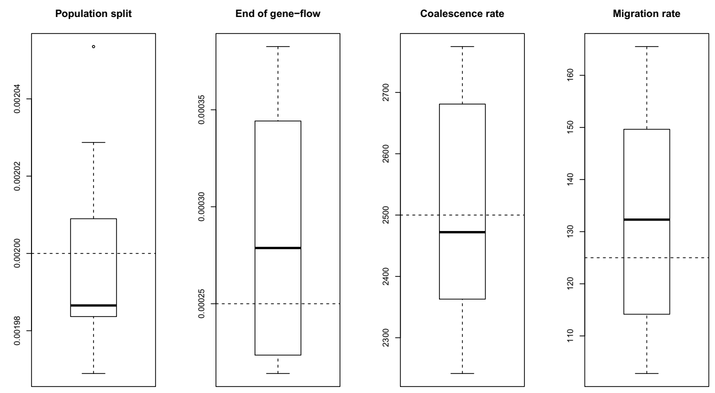


Fig. 7. Parameter estimates on simulated data. In a model with an initial population split followed by a period of migration after which gene-flow stops, we simulated data and estimated parameters. The box plot shows the distribution of estimates in ten simulations. The dashed lines show the simulated parameter.

Different points of attack are possible. The state space explosion can be alleviated using reduction methods. Samples within the same population are by nature symmetric, so symmetry reduction [4,14] is an obvious approach. The symmetry method allows us to consider two states as equal if it is possible to find a permutation of the samples mapping one to the other. In our example, we can allow all permutations of samples. To evaluate the viability of this approach, we built a prototype implementation of this using CPN Tools [26]. Due to the limitation of our prototype, we have implemented this method using a simple hand-crafted mapping mapping each state to a canonical member of its symmetry group. This means that we not necessarily generate the minimal state space but that we have an efficient means of computing symmetric mappings. In the future we want to either improve this or instead use the lower-level formalism of symmetric nets [2], for which symmetries can be computed automatically. In Table 2, we have shown the reduced sizes of the state space for the different cases. We see that the ratio of the size of the reduced state space to the full state space gets much smaller as the number of species grow.

The sweep-line method [3] makes is possible to delete states from memory when they are no longer needed. In our example, we notice that as soon as two species have coalesced, it is impossible for them to split again. Thus, a state can never have arcs to states where fewer species have coalesced. We can define a progress measure which counting the number of coalesced species and process states in a least progress-first-order. This allows us to delete any states from memory with a lower progress value than any state we have yet to process. We can generate a highly efficient progress measure by creating the state space for the model with only coalescence, computing the strongly connected components of this graph, traverse this graph using a breadth-first traversal and use the

Table 2. Summaries of the state space sizes, SCCs and construction time for the full state space and reduced state spaces. The *basic model* are scenarios with all samples in the same population, while *models with gene-flow* assume one population per sample.

	Full		Symmetry		Sweep-line	Construction time	
Size	States	SCCs	States	SCCs	States	Full	Symmetry
Basic model							
n = 1	2	1	2	1	2	0.00	0.00
n = 2	15	4	12	4	12	0.00	0.00
n = 3	203	25	77	11	46	0.02	0.02
n = 4	4 140	225	607	39	363	0.92	0.30
n = 5	115 975	2 704	5 455	215	2 659	47.05	4.45
n = 6	–	–	54 054	1 604	25 518	–	65.10
n = 7	–	–	586 534	–	266 550	–	2 043.62
Model with gene-flow							
n = 2	94	4	79	4	76	0.01	0.02
n = 3	12 351	25	6 065	10	5 017	4.81	4.99
n = 4	3 188 340	–	731 840	–	451 559	26 525.88	5 720.11

breadth-first rank of each state as progress value for the full state space. In Table 2 we show the maximum number of states in memory during processing (when combined with symmetry reduction). The time to construct the state space and the number of SCCs is the same as for constructing the symmetry reduced graph.

We notice that if we sort the states of the (full) state space such that states belonging to the same strongly connected component are kept together, we get a rate which has block corresponding to each of the strongly connected components. If we know the layout of the blocks we can thus compute the rate matrix from the individual blocks. Each block on the diagonal of the rate matrix corresponds to all transitions internal to a SCC and all other blocks correspond to transitions from one SCC to another. A property of the sweep-line method is that it keeps entire SCCs in memory at the same time (assuming that the progress measure is monotone, which it is here). Thus, we can easily compute all blocks on the diagonal of the rate matrix. Furthermore, we can store, for each SCC, all transitions leading out of the SCC and subsequently sort them according to the target SCC. This allows us to also compute all other blocks. If we sort the states according to the progress measure, we get a rate matrix which is almost upper-triangular. The reason is that it is only possible to go from a state with lesser progress to a state with higher progress. The only parts below the diagonal are the blocks corresponding to transitions internal to a SCC. Furthermore, we know that the rate matrix will be very sparse even above the diagonal, as we only have a non-zero block if there is an arc from one SCC to another.

Unfortunately, the rates of transitions are not necessarily the same even for symmetrical transitions, and it is not obvious to us how to construct the HMM transition matrix from the symmetry reduced CPN state space; future work includes combining lumping techniques for Markov chains [2,6] with the construction of the rate matrix outlines using the sweep-line method to our case.

Rather than exponentiating the rate matrix, it is also possible to get good approximations of the probabilities through Monte Carlo simulation where we can simulate thousands or millions of runs of the continuous time Markov chain and obtain the probabilities this way. Future work will concentrate on ways of extending the complexity of scenarios by alleviating the state space explosion problem.

6 Conclusions

Coalescent hidden Markov models (CoalHMMs) [8,12] are a recent invention that has become popular for genome analysis as they are currently the only approach that is computationally efficient enough for analyzing full genome data. Different variants of CoalHMMs have been successfully used in recent great ape genome projects, including an analysis of human population size changes [16], an analysis of the orangutan sub-species [17,18], and estimating speciation times between humans, chimpanzees, bonobos, gorillas and orangutans [13,25,29]. The demographic scenarios explored, however, have been very simple ones because the CoalHMMs have so far been constructed manually. Typically, this involves deriving equations for transition probabilities by approximating the coalescence process, which at best is tedious and in cases can introduce biases in the estimation because of the simplifying assumptions necessary to do this. Computing the transition probabilities using a continuous time Markov chain alleviates this somewhat, but manually constructing the Markov chain is still only possible for simple scenarios.

While the colored Petri net model we present here is very simple, we stress that it is capable of modeling most demographic scenarios. Combined with an algorithm for translating a formal model of demographics like this into the final CoalHMM, complex scenarios can be explored in genome analysis. As the cost of sequencing genomes is steadily decreasing, the bottleneck in future genome projects will be in the mathematical modeling and in constructing analysis methods that both captures the complexity of the genomes and are computationally efficient. We believe that CoalHMMs combined with formal methods such as Petri nets can be a powerful approach in this.

Acknowledgements. The authors wish to thank Lars M. Kristensen for fruitful discussions about the use state-space reduction techniques.

References

1. Chen, G.K., Marjoram, P., Wall, J.D.: Fast and flexible simulation of DNA sequence data. Genome Res. 19(1), 136–142 (2009)

2. Chiola, G., Dutheillet, C., Franceshinis, G., Haddad, S.: Stochastic Well-Formed Colored Nets and Symmetric Modeling Applications. IEEE Trans. Computers 42(11), 1343–1360 (1993)
3. Christensen, S., Kristensen, L.M., Mailund, T.: A Sweep-Line Method for State Space Exploration. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 450–464. Springer, Heidelberg (2001)
4. Clarke, E., Emerson, E., Jha, S., Sistla, A.P.: Symmetry Reductions in Model Checking. In: Vardi, M.Y. (ed.) CAV 1998. LNCS, vol. 1427, pp. 147–158. Springer, Heidelberg (1998)
5. Davison, D., Pritchard, J.K., Coop, G.: An approximate likelihood for genetic data under a model with recombination and population splitting. Theoretical Population Biology 75(4), 331–345 (2009)
6. Derisavi, S., Hermanns, H., Sanders, W.H.: Optimal state-space lumping in markov chains. Inf. Process. Lett. 87(6), 309–315 (2003)
7. Durbin, R., Eddy, S.R., Krogh, A., Mitchison, G.: Biological Sequence Analysis. Probabilistic Models of Proteins and Nucleic Acids. Cambridge Univ. Pr. (February 2005)
8. Dutheil, J.Y., Ganapathy, G., Hobolth, A., Mailund, T., Uyenoyama, M.K., Schierup, M.H.: Ancestral population genomics: the coalescent hidden Markov model approach. Genetics 183(1), 259–274 (2009)
9. Eriksson, A., Mahjani, B., Mehlig, B.: Sequential Markov coalescent algorithms for population models with demographic structure. Theor. Popul. Biol. 76(2), 84–91 (2009)
10. Green, R.E., et al.: A draft sequence of the neandertal genome. Science 328(5979), 710–722 (2010)
11. Hein, J., Schierup, M.H., Wiuf, C.: Gene genealogies, variation and evolution. a primer in coalescent theory. Oxford University Press, USA (2005)
12. Hobolth, A., Christensen, O.F., Mailund, T., Schierup, M.H.: Genomic relationships and speciation times of human, chimpanzee, and gorilla inferred from a coalescent hidden Markov model. PLoS Genet 3(2), e7 (2007)
13. Hobolth, A., Dutheil, J.Y., Hawks, J., Schierup, M.H., Mailund, T.: Incomplete lineage sorting patterns among human, chimpanzee, and orangutan suggest recent orangutan speciation and widespread selection. Genome Res. 21(3), 349–356 (2011)
14. Jensen, K.: Condensed State Spaces for Symmetrical Coloured Petri Nets. Formal Methods in System Design 9(1/2), 7–40 (1996)
15. Jensen, K., Kristensen, L.M.: Coloured Petri Nets. Modeling and Validation of Concurrent Systems. Springer-Verlag New York Inc. (June 2009)
16. Li, H., Durbin, R.: Inference of human population history from individual whole-genome sequences. Nature (July 2011)
17. Locke, D.P., et al.: Comparative and demographic analysis of orang-utan genomes. Nature 469(7331), 529–533 (2011)
18. Mailund, T., Dutheil, J.Y., Hobolth, A., Lunter, G., Schierup, M.H.: Estimating Divergence Time and Ancestral Effective Population Size of Bornean and Sumatran Orangutan Subspecies Using a Coalescent Hidden Markov Model. PLoS Genet. 7(3), e1001319 (2011)
19. Mailund, T., Schierup, M.H., Pedersen, C.N.S., Mechlenborg, P.J.M., Madsen, J.N., Schauser, L.: CoaSim: a flexible environment for simulating genetic data under coalescent models. BMC Bioinformatics 6, 252 (2005)
20. Marjoram, P., Wall, J.D.: Fast "coalescent" simulation. BMC Genetics 7, 16 (2006)
21. Marsan, M.: Stochastic Petri Nets: An Elementary Introduction. In: Rozenberg, G. (ed.) APN 1989. LNCS, vol. 424, pp. 1–29. Springer, Heidelberg (1990)

22. McVean, G.A.T., Cardin, N.J.: Approximating the coalescent with recombination. Philosophical Transactions of the Royal Society of London. Series B, Biological Sciences 360(1459), 1387–1393 (2005)
23. Moler, C., van Loan, C.: Nineteen Dubious Ways to Compute the Exponential of a Matrix, Twenty-Five Years Later. SIAM Review 45(1), 3–49 (2003)
24. Paul, J.S., Steinrucken, M., Song, Y.S.: An Accurate Sequentially Markov Conditional Sampling Distribution for the Coalescent With Recombination. Genetics 187(4), 1115–1128 (2011)
25. Prüfer, K., et al.: The bonobo genome compared with the genomes of chimpanzee and human, under review at Nature
26. Vinter Ratzer, A., 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)
27. Reich, D., et al.: Genetic history of an archaic hominin group from denisova cave in siberia. Nature 468(7327), 1053–1060 (2010)
28. Reich, D., et al.: Denisova admixture and the first modern human dispersals into southeast asia and oceania. Am. J. Hum. Genet. 89(4), 516–528 (2011)
29. Scally, A., et al.: Insights into hominid evolution from the gorilla genome sequence. Nature 483(7388), 169–175 (2012)
30. Song, Y.S., Lyngso, R., Hein, J.: Counting All Possible Ancestral Configurations of Sample Sequences in Population Genetics. IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB) 3(3), 239 (2006)
31. Thalmann, O., Fischer, A., Lankester, F., Pääbo, S., Vigilant, L.: The complex evolutionary history of gorillas: insights from genomic data. Mol. Biol. Evol. 24(1), 146–158 (2007)

A Hidden Markov Models

Hidden Markov models (HMMs) is a framework for modeling sequential data such as DNA sequences. HMMs provide a computational efficient way of analyzing sequential data, that would otherwise be intractable to analyze.

Given a sequence of observation $Z_1, Z_2, \ldots, Z_n$, the Markov assumption states the the probability of the entire sequence $\Pr(Z_1, Z_2, \ldots, Z_n)$ can be stated as a sequence of conditional probabilities

$$\Pr(Z_1, Z_2, \ldots, Z_n) = \Pr(Z_1) \prod_{i=1}^{n-1} \Pr(Z_{i+1} \mid Z_i)$$

which is typically much more efficient to calculate.

In many applications, however, a sequence of observations cannot be justified modeled in this way. The genetic differences between a sample of genes, for instance, is conditional on the local genealogies, but whereas we can model the local genealogies as a Markov process the observed genetic differences do not directly lead to a Markov process.

Hidden Markov models instead assumes that we have a sequence of unseen parameters that *do* follow a Markov process, but that the observations we see depend on those parameters but are not themselves a Markov process. An HMM models a sequence of observations, $X_1, X_2, \ldots, X_n$, by assuming there is an underlying but unobserved sequence of states the process goes through, $Z_1, Z_2, \ldots, Z_n$, that determines the probability of the observations. Each observation depends on one hidden state, $\Pr(X_i \mid X_1, \ldots, X_n, Z_1, \ldots, Z_n) = \Pr(X_i \mid Z_i)$, as the genetic differences between a set of genes would depend on the local genealogies at each position but not neighboring genealogies. Were both the hidden states and the observations known, the joint probability would be

$$\Pr(\mathbf{X}, \mathbf{Z}) = \Pr(Z_1) \Pr(X_1 \mid Z_1) \prod_{i=1}^{n-1} \Pr(Z_{i+1} \mid Z_i) \Pr(X_{i+1} \mid Z_{i+1}) \quad .$$

The Markov process states, Z_i, however are not observed in a *hidden* Markov model, only the sequence $X_1, \ldots, X_n$. Efficient dynamic programming algorithms exist, however to sum over all hidden state paths and thus computing $\Pr(\mathbf{X})$ and from this making maximum likelihood parameter estimations.

An HMM is completely parameterized by specifying the initial state probabilities $\pi = (\Pr(Z_1), \ldots, \Pr(Z_m))$ for possible hidden states $Z_1, \ldots, Z_m$, the *transition* probability matrix $T_{i,j} = \Pr(Z_i \mid Z_j)$ and the *emission* probability matrix $E_{l,m} = \Pr(X_m \mid Z_l)$.

An SMT-Based Discovery Algorithm for C-Nets

Marc Solé[1,2] and Josep Carmona[2]

[1] Computer Architecture Department, UPC
msole@ac.upc.edu
[2] Software Department, UPC
jcarmona@lsi.upc.edu

Abstract. Recently, *Causal nets* have been proposed as a suitable model for process discovery, due to their declarative semantics and the great expressiveness they possess. In this paper we propose an algorithm to discover a causal net from a set of traces. It is based on encoding the problem as a Satisfiability Modulo Theories (SMT) formula, and uses a binary search strategy to optimize the derived model. The method has been implemented in a prototype tool that interacts with an SMT solver. The experimental results obtained witness the capability of the approach to discover complex behavior in limited time.

1 Introduction

Process Mining [1] is a discipline that aims at using the data available in information systems to discover the processes taking place inside an organization, check their compliance and perform predictions. It is an evolving area which, although becoming crucial to support decision making, still needs to settle down in terms of algorithmic and model support. One example of this is the vast amount of algorithms that exist for a wide range of models: Petri nets, Heuristic nets, Event-Driven Process Chains, Fuzzy models, among others [1].

Recently, a formalism called Causal nets (C-nets) [2] has been proposed as a suitable modeling language for process mining. It is a rather compact representation that allows expressing complex behavior that is sometimes difficult to describe using other models. For instance consider the set of traces (*log*) in Fig. 1(a). In Fig. 1(b) we can see two Petri nets that are required to represent all the sequences without adding extra behavior. It is possible to use a single Petri net, instead of two, but then the use of *silent* transitions (or, alternatively, transitions sharing the same label) is needed, as shown in (c). On the other hand the equivalent C-net representation (d) is quite compact. The semantics of that C-net can be informally described as:

Activity a must be executed initially, followed either by b, c, or b and c. Any of these three possibilities is followed by the execution of activity e.

Figure 2 (from [2]) shows a more meaningful example, describing a C-net that models the process of booking resources for travel.

The *discovery problem* refers to obtaining a model (in some suitable formalism) that describes the behavior recorded in a log. To the best of our knowledge, there

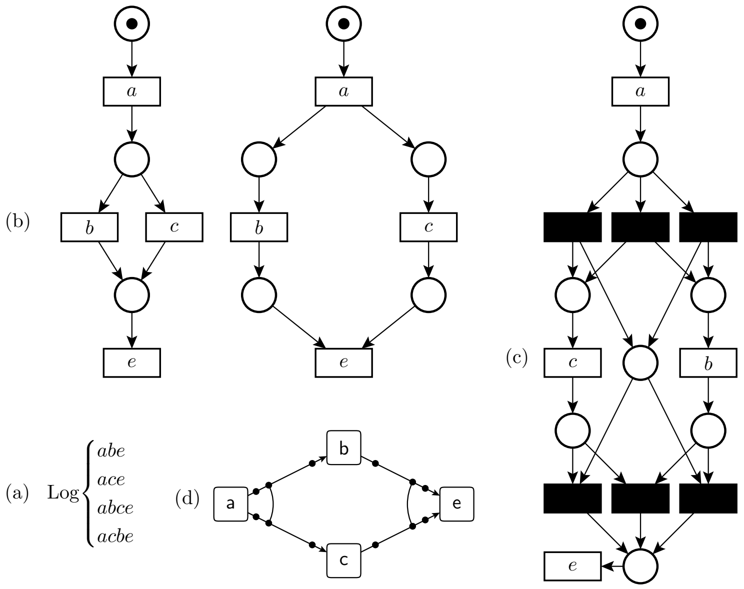


Fig. 1. (a) A log. (b) Set of two Petri nets describing the log. (c) Single Petri net describing the log with the help of silent transitions. (d) Causal net of the same log.

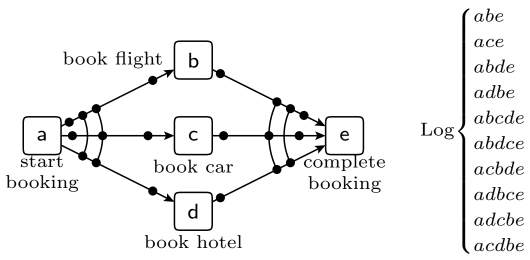


Fig. 2. Causal net C_{travel} from [2]

are few discovery algorithms for C-nets. One indirect method is to first discover a Petri net and then convert it into a C-net as described in [2]. This strategy is very cumbersome since the conversion introduces a silent activity in the C-net for each place in the Petri net, thus increasing significantly the size of the C-net, although a very compact C-net representing the same language may exist. Another possibility is to use discovery algorithms for *flexible heuristic nets* [3], a model closely related to C-nets, or *heuristic nets* [4,5] which can be viewed as a restricted subclass of C-nets. However, these two approaches cannot guarantee that the log is included in the language of the derived model.

This paper presents the first algorithm to discover a C-net from a log that guarantees that i) the language of the C-net includes the log, and ii) among those satisfying the previous property, the C-net has the minimal number of arcs. It is based on encoding the problem as an SMT formula, and using binary search to find a minimal C-net (in terms of number of arcs). A prototype tool which interfaces with an SMT solver is reported, showing promising experimental results.

Organization of the Paper: In Sect. 2 we give the formal definition of C-nets and we introduce some of the mathematical background used in the rest of the paper. Our approach to the discovery of C-nets is explained in Sect. 3 and experimentally tested in Sect. 4. Finally, Sect. 5 presents some future work and concludes this paper.

2 Background

2.1 Mathematical Preliminaries

A multiset (or a bag) is a set in which elements of a set X can appear more than once, formally defined as a function $X \to \mathbb{N}$. We denote as $\mathbb{B}(X)$ the space of all multisets that can be created using the elements of X. Let $M_1, M_2 \in \mathbb{B}(X)$, we consider the following operations on multisets: sum $(M_1 + M_2)(x) = M_1(x) + M_2(x)$, subtraction $(M_1 - M_2)(x) = \max(0, M_1(x) - M_2(x))$ and inclusion $(M_1 \subseteq M_2) \Leftrightarrow \forall x \in X, M_1(x) \leq M_2(x)$. As usual, sets will be considered as bags when necessary.

A log L is a bag of sequences of activities. In this work we restrict the type of sequences that can form a log. In particular, we assume that all the sequences start with the same initial activity and end with the same final activity, and that these two special activities only appear once in every sequence. This assumption is without loss of generality, since any log can be easily converted by using two new activities that are properly inserted in each trace.

Given a finite sequence of elements $\sigma = e_1 e_2 \ldots e_n$, its length is denoted $|\sigma| = n$, and its prefix sequence up to element i, with $i \leq n$, denoted σ_i, is $e_1 \ldots e_i$. We define σ_0 as the empty sequence, denoted ϵ. Conversely, its suffix sequence after i, with $i < n$, denoted $\sigma_{i\to}$, is $e_{i+1} \ldots e_n$. The alphabet of σ, denoted A_σ, is the set of elements in σ. We extend this notation to logs, so that A_L is the alphabet of the log L, *i.e.*, $A_L = \bigcup_{\sigma \in L} A_\sigma$. Finally, we extend the $\in$ notation to sequences, so that $e \in \sigma$ is true if, and only if, e occurs in σ.

2.2 Causal Nets (C-Nets)

In this section we introduce the main model used along this paper.

Definition 1 (Causal net [2]). *A Causal net is a tuple $C = \langle A, a_s, a_e, I, O \rangle$, where A is a finite set of activities, $a_s \in A$ is the start activity, $a_e \in A$ is the end activity, and I (and O) are the set of possible* input *(*output *resp.)* bindings *per activity. Formally, both I and O are functions $A \to S_A$, where $S_A = \{X \subseteq \mathcal{P}(A) \mid X = \{\emptyset\} \vee \emptyset \notin X\}$, and satisfy the following conditions:*

– $\{a_s\} = \{a \mid I(a) = \{\emptyset\}\}$ *and* $\{a_e\} = \{a \mid O(a) = \{\emptyset\}\}$
– *all the activities in the graph* $(A, \text{arcs}(C))$ *are on a path from* a_s *to* a_e*, where* arcs(C) *is the* dependency relation induced by I and O *such that* $\text{arcs}(C) = \{(a_1, a_2) \mid a_1 \in \bigcup_{X \in I(a_2)} X \wedge a_2 \in \bigcup_{Y \in O(a_1)} Y\}$.

Definition 1 slightly differs from the original one from [2], where the set arcs(C) is explicitly defined in the tuple. The C-net of Fig. 1(d) is formally defined as $C = \langle\{a, b, c, e\}, a, e, I, O\rangle$, with $I(a) = \emptyset$, $O(a) = \{\{b\}, \{c\}, \{b, c\}\}$, $I(b) = \{\{a\}\}$, $O(b) = \{\{e\}\}$, $I(c) = \{\{a\}\}$, $O(c) = \{\{e\}\}$, $I(e) = \{\{b\}, \{c\}, \{b, c\}\}$ and $O(e) = \emptyset$. The dependency relation of C, which corresponds graphically to the arcs in the figure, in this case is: $\text{arcs}(C) = \{(a, b), (a, c), (b, e), (c, e)\}$. The activity bindings are denoted in the figure as dots in the arcs, *e.g.*, $\{b\} \in O(a)$ is represented by the dot in the arc (a, b) that is next to activity a, while $\{a\} \in I(a)$ is the dot in arc (a, b) next to b. Non-singleton activity bindings are represented by circular segments connecting the dots: $\{b, c\} \in O(a)$ is represented by the two dots in arcs (a, b), (a, c) that are connected through a circular segment.

Definition 2 (Binding, Binding Sequence, Activity Projection). *Given a C-net* $\langle A, a_s, a_e, I, O\rangle$*, the set* B *of activity bindings is* $\{(a, S^I, S^O) \mid a \in A \wedge S^I \in I(a) \wedge S^O \in O(a)\}$*. A binding sequence* $\beta \in B^*$ *is a sequence of activity bindings. Given a binding sequence* $\beta = (a_1, S_1^I, S_1^O) \ldots (a_n, S_n^I, S_n^O)$*, its activity projection is the activity sequence denoted* $\sigma_\beta = a_1 \ldots a_n$.

Two binding sequences of the C-net in Fig. 1(d) are: $\beta^1 = (a, \emptyset, \{b\})(b, \{a\}, \{e\})(e, \{b\}, \emptyset)$ and $\beta^2 = (a, \emptyset, \{b, c\})(c, \{a\}, \{e\})(e, \{c\}, \emptyset)$. The projection of β^1 is $\sigma_{\beta^1} = abe$.

The semantics of a C-net are achieved by selecting, among all the possible binding sequences, the ones satisfying certain properties. These sequences will form the set of *valid binding sequences* of the C-net, and their corresponding projection (see Def. 2) will define the language of the C-net. The next definition addresses this.

Definition 3 (State, Valid Binding Sequence, Language). *Given a C-net* $C = \langle A, a_s, a_e, I, O\rangle$*, its state space* $S = \mathbb{B}(A \times A)$ *is composed of states that are bags of* obligations *(activity 2-tuples). An obligation* (a, b) *expresses that activity* a *has executed and expects* b *to execute. When this obligation is satisfied, it is removed from the state, thus a state informally represents the bag of pending (*i.e.*, not yet satisfied) obligations. Function* $\psi \in B^* \rightarrow S$ *is defined inductively:* $\psi(\epsilon) = \emptyset$ *and* $\psi(\beta \cdot (a, S^I, S^O)) = \psi(\beta) - (S^I \times \{a\}) + (\{a\} \times S^O)$*. The state* $\psi(\beta)$ *is the state of the C-net after the sequence of bindings* β*. The binding sequence* $\beta = (a_1, S_1^I, S_1^O) \ldots (a_n, S_n^I, S_n^O)$ *is said to be* valid *if:*

1. $a_1 = a_s$, $a_n = a_e$ *and* $\forall k : 1 < k < n, a_k \in A \setminus \{a_s, a_e\}$
2. $\forall k : 1 \leq k \leq n, (S_k^I \times \{a_k\}) \subseteq \psi(\beta_{k-1})$
3. $\psi(\beta) = \emptyset$

The set of all valid binding sequences of C *is denoted as* $V(C)$*. The language of* C*, denoted* $\mathcal{L}(C)$*, is the set of activity sequences that correspond to a valid binding sequence of* C*,* i.e., $\mathcal{L}(C) = \{\sigma_\beta \mid \beta \in V(C)\}$.

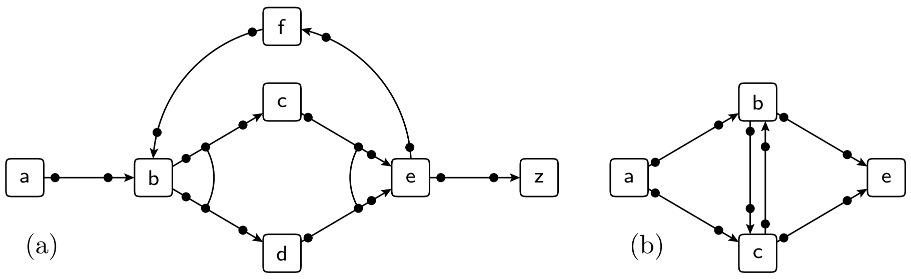


Fig. 3. (a) C-net mixing concurrent and exclusive behavior for activities c and d. (b) Immediately follows C-net of log $L = \{abce, acbe\}$. Its language (using regular expressions) is $ab(cb)^*e \cup a(bc)^+e \cup ac(bc)^*e \cup a(cb)^+e$.

For instance, in Fig. 1(d), β^1 is a valid binding sequence, while β^2 is not, since the final state is not empty (condition 3 is violated). The language of that C-net is $\{abe, ace, abce, acbe\}$. Similarly to *Workflow* Petri nets [6], C-nets have a notion of *soundness* [2]:

Definition 4 (Soundness). *A C-net $C = \langle A, a_s, a_e, I, O\rangle$ is* sound *if (i) $\forall a \in A, \forall S \in I(a), \exists \beta \in V(C), \exists X \subseteq A : (a, S, X) \in \beta$, and (ii) $\forall a \in A, \forall S \in O(a), \exists \beta \in V(C), \exists X \subseteq A : (a, X, S) \in \beta$. That is, every input and output binding defined in C is used in at least one valid sequence.*

Importantly, C-nets can naturally represent behaviors that cannot be easily expressed in the Petri net notation unless unobservable (silent) transitions and/or transitions sharing labels are used. Fig. 3(a) illustrates this point. In the C-net, activities c and d can occur concurrently or exclusively, even in different iterations of the loop created by activity f, *e.g.*, *abcdez* or *abdefbcez*. However, there is still another possibility that arises from combining the output binding $\{c, d\}$ of b and the input bindings $\{c\}$ and $\{d\}$ of e: *abdceefbdfbcez*. Note that in this last trace, activity e could execute twice for a single b (although in the overall trace they execute the same number of times), and that after these two executions there are two (e, f) obligations in the state (*i.e.*, $\psi(abcdee) = 2(e, f)$), which shows the necessity of defining the state as a multiset of obligations.

C-nets, contrary to Petri nets, have an "additive" nature. That is, while adding a place to a Petri net can only restrict behavior, adding an arc (or any other element) to a C-net can only add behavior. The "additive" nature of C-nets is formally defined with the help of Def. 5 and Property 1.

Definition 5. *Given two C-nets $C_1 = \langle A_1, a_s^1, a_e^1, I_1, O_1\rangle$ and $C_2 = \langle A_2, a_s^2, a_e^2, I_2, O_2\rangle$, we say that C_1 is included in C_2, denoted $C_1 \subseteq C_2$, if:*

- $a_s^1 = a_s^2 \wedge a_e^1 = a_e^2$,
- $A_1 \subseteq A_2$, *and*
- $\forall a \in A_1, I_1(a) \subseteq I_2(a) \wedge O_1(a) \subseteq O_2(a)$

Property 1 ([7]). Let C_1 and C_2 be two C-nets. If $C_1 \subseteq C_2$, then $V(C_1) \subseteq V(C_2)$, $\mathcal{L}(C_1) \subseteq \mathcal{L}(C_2)$ and $\text{arcs}(C_1) \subseteq \text{arcs}(C_2)$.

2.3 C-Net Discovery

Given a log L, the problem tackled in this paper is to derive a C-net C such that $\mathcal{L}(C) \supseteq L$ and C contains the minimal number of arcs. In Sect. 3 we present such a method together with heuristics to limit the language of the derived net.

Given the additive nature of C-nets, there is a simple method to generate a C-net that can replay all the traces in L. It is based on the *immediately follows* relation [6] between the activities in L, denoted $<_L$ and defined as $<_L = \{(a,b) \mid \exists \sigma = a_1 \dots a_n \in L, \exists i : 1 \leq i < n \wedge a_i = a \wedge a_{i+1} = b\}$.

Definition 6. *Given a log L, the* immediately follows C-net of L, *denoted $C_{IF}(L)$, is the C-net $\langle A, a_s, a_e, I, O \rangle$ such that: (i) $A = A_L$, (ii) $\forall \sigma = a_1 \dots a_n \in L, a_1 = a_s \wedge a_n = a_e$, (iii) $\forall a \in A, O(a) = \{\{b\} \mid a <_L b\} \wedge I(a) = \{\{b\} \mid b <_L a\}$. Trivially, $\mathcal{L}(C_{IF}(L)) \supseteq L$.*

The immediately follows C-net can be computed in linear time with respect of the size of the log, but allows for many unobserved behavior, thus exhibiting a poor *precision* [8]. For instance consider the following log: $L = \{abce, acbe\}$. Activities b and c interleave, but if we build the immediately follows C-net (Fig. 3(b)) we can see that it allows for loops of arbitrary length involving these two activities, *e.g.*, *abcbce* or *acbce*, since $\mathcal{L}(C_{\text{IF}}(L)) = ab(cb)^*e \cup a(bc)^+e \cup ac(bc)^*e \cup a(cb)^+e$, which significantly differs from L.

3 Discovering Strategies for C-Nets Based on SMT

3.1 Protobinding Sequences of a Log

In Sect. 2.2 we have seen first the definition of a C-net and then the definition of the valid sequences of bindings it can produce. To discover a C-net from a log, we follow the same path but in the opposite direction: we will define sequences of triples representing unrestricted bindings that satisfy some properties, and then we will show that given these sequences, it is easy to obtain a C-net C such that these sequences are actually valid sequences of bindings of C. Consequently, this transforms the discovery problem for C-nets into the problem of deriving these sequences of triples from the sequences in the log. Let us first formalize the concept of *protobinding*:

Definition 7 (Protobinding, Well-Formed Protobinding Sequence). *A triple (a, X, Y) is a* protobinding *if a is an element and both X and Y are sets. A sequence $\beta = (a_1, X_1, Y_1) \dots (a_n, X_n, Y_n)$ of protobindings is* well-formed *if it satisfies the following conditions:*

(W1) $\forall i : 1 < i \leq n, X_i \supset \emptyset \wedge a_i \neq a_1$
(W2) $\forall i : 1 \leq i < n, Y_i \supset \emptyset \wedge a_i \neq a_n$

(W3) $X_1 = Y_n = \emptyset$
(W4) $\forall i : 1 \leq i \leq n, \psi(\beta_{i-1}) \supseteq (X_i \times \{a_i\})$
(W5) $\psi(\beta_n) = \emptyset$

Compared with the definition of binding (Def. 2), this is a weaker definition since it is no longer tied to a particular C-net. Given a set B of sequences of well-formed sequences of protobindings it is possible to characterize (with necessary conditions) all the C-nets such that their set of valid sequences of bindings contain the sequences of protobindings in B, as next lemma states.

Lemma 1. *Given a set of well-formed protobinding sequences B with identical initial and final activities a_s and a_e, respectively, and a C-net $C = \langle A, a_s, a_e, I, O\rangle$. The following conditions:*

(N1) $A \supseteq \{a \mid \exists \beta \in B : (a, X, Y) \in \beta\}$
(N2) $\forall a \in A,\ I(a) \supseteq \{X \mid \exists \beta \in B, \exists Y : (a, X, Y) \in \beta\}$
(N3) $\forall a \in A,\ O(a) \supseteq \{Y \mid \exists \beta \in B, \exists X : (a, X, Y) \in \beta\}$

hold if, and only if, $V(C) \supseteq B$.

Proof. $\boxed{\Rightarrow}$ Take any protobinding sequence $\beta \in B$, we will prove that since it is well-formed it is actually a valid binding sequence of C, thus $V(C) \supseteq B$. By Def. 2, we know that to be a binding sequence (not necessarily valid) of C, every protobinding $(a, X, Y) \in \beta$ must satisfy that $a \in A$, $X \in I(a)$ and $Y \in O(a)$, which is trivially true given our definition of C. Thus, given N1, N2 and N3, β is a binding sequence of C. We now prove that β is in fact also valid. To prove this we use the fact that β is well-formed (thus satisfies W1 to W5). Proving that $\beta = (a_1, X_1, Y_1) \ldots (a_n, X_n, Y_n)$ is a valid binding sequence, we need to prove that $a_1 = a_s$, $a_n = a_e$ and $\forall k : 1 < k < n, a_k \in A \setminus \{a_s, a_e\}$. The first two conditions are satisfied because we require that all the sequences in B start with a_s and end with a_e. The third condition is guaranteed by W1 and W2. The remaining two conditions for a valid sequence are directly W4 and W5.

$\boxed{\Leftarrow}$ If C is a C-net and $\mathcal{L}(C) \supseteq \{\sigma_\beta \mid \beta \in B\}$, then if N1 does not hold it exists some activity that appears in the sequences of B that cannot be executed by C. If N2 does not hold, then there is some sequence that is not possible because some X cannot be used by C, and the same applies for N3 and Y. $\square$

Creating a tuple $\langle A, a_s, a_e, I, O\rangle$ satisfying N1, N2 and N3 does not necessarily yield a C-net (for instance we could add activities to A that violate some of the conditions of Def. 1), since these are necessary but not sufficient conditions. To guarantee that a C-net is generated and it is *sound* (c.f., Def. 4), we restrict the conditions of the previous lemma in the following theorem:

Theorem 1. *Given a set of well-formed protobinding sequences B with identical initial and final activities a_s and a_e, respectively, the tuple $C = \langle A, a_s, a_e, I, O\rangle$ with:*

(T1) $A = \{a \mid \exists \beta \in B : (a, X, Y) \in \beta\}$

(T2) $\forall a \in A,\ I(a) = \{X \mid \exists \beta \in B, \exists Y : (a, X, Y) \in \beta\}$
(T3) $\forall a \in A,\ O(a) = \{Y \mid \exists \beta \in B, \exists X : (a, X, Y) \in \beta\}$

is a sound C-net such that $V(C) \supseteq B$.

Proof. We will prove that C is a C-net, since then we can use Lemma 1 to prove that $V(C) \supseteq B$. Proving that C is sound, once we know it is a C-net, is straightforward since every input and output binding in I and O appears in at least one of the sequences in B, thus in at least one valid binding sequence of C, thus C is sound. So we have only to prove that C is a C-net satisfying Def. 1. First of all we have to prove that a_s is the only activity with empty input binding ($I(a_s) = \emptyset$) and a_e is the only activity with empty output binding ($O(a_e) = \emptyset$). Consider all sequences $\beta = (a_1, X_1, Y_1) \ldots (a_n, X_n, Y_n) \in B$, by W3, we know that $\emptyset \in I(a_s)$ and $\emptyset \in O(a_e)$ (because $a_1 = a_s$ and $a_n = a_e$), and since these two activities are only executed once (due to W1 and W2) there is no other set in $I(a_s)$ and $O(a_e)$. Since the other activities $a \in A \setminus \{a_s, a_e\}$ are not executed at the begining nor the end of β, also by W1 and W2 we know their X_i and Y_i sets are non-empty, thus they cannot have $I(a) = \{\emptyset\}$ nor $O(a) = \{\emptyset\}$. We must also prove that the I and O functions are defined over the powerset of A, *i.e.*, $\forall a \in A, \forall X \in I(a), X \subseteq A$. Because of W4 and W5 the sequence can only consume obligations previously produced, and all the obligations must be consumed. If some X_i in β contains activities not in A, it is impossible to satisfy W4 thus it would not be well-formed, which is a contradiction. Similarly, if some Y_i in β contains activities not in A, then the obligations created can never be consumed, violating W5. Finally, we have to prove that all the activities in the graph $(A, \text{arcs}(C))$ are on a path from a_s to a_e. Again by W4 and W5, since an activity a_i (different than a_s) can only execute when it has at least one obligation (a, a_i) for it in $\psi(\beta_{i-1})$ and $a \in X_i$, thus $(a, a_i) \in \text{arcs}(C)$, and the source of all this chain of obligations is a_s and ends in a_e or otherwise a_i (or some of its successors in the obligation chain) would have produced an obligation that nobody would have consumed, violating W5, then every activity is in a dependency chain between a_s and a_e. $\square$

The theorem allows an easy conversion from protobinding sequences to C-nets, so that the C-net discovery problem from a log L can be reduced to the following problem: given a log L, compute a well-formed protobinding sequence for each sequence in L. Since by definition all our sequences in the log have the same initial and final activities, then all the protobinding sequences will also have, thus we can use Theorem 1 to discover a C-net. Although the theorem does not consider all the C-nets whose valid binding sequences include the protobinding sequences B, it gives always the smallest C-net (in terms of valid binding sequences and also in terms of number of structural elements of the C-net) that can generate the sequences in B as next corollary states.

Corollary 1. *Given a set of well-formed protobinding sequences* B *with identical initial and final activities* a_s *and* a_e*, respectively, a C-net* C *such that* $V(C) \supseteq B$ *and a C-net* $C_\perp$ *satisfying T1, T2 and T3, then* $C_\perp \subseteq C$.

Proof. If C is a C-net and $V(C) \supseteq B$, then by Lemma 1 we know it satisfies N1, N2 and N3. Since $C_\perp$ satisfies T1, T2 and T3, by Def. 5 $C_\perp \subseteq C$, and this entails by Property 1 $V(C_\perp) \subseteq V(C)$. □

In the next section we explain how we can encode as linear constraints the problem of computing the sequences of protobindings.

3.2 Encoding the Problem as Linear Constraints

Given a sequence $\sigma = a_1 \dots a_n$ of a log L, it is trivial to build a protobinding sequence β_σ out of it as $\beta_\sigma = (a_1, X_1, Y_1) \dots (a_n, X_n, Y_n)$. The difficult part is to ensure that β_σ is actually well-formed. We will encode the unknown X_i and Y_i sets using integer variables and then define the linear constraints that will guarantee that β_σ is well-formed. We start by delimiting the values that the X_i and Y_i unknowns can take using the following property:

Property 2. Let $\sigma = a_1 \dots a_n$ be a sequence of activities. Consider the protobinding sequence $\beta_\sigma = (a_1, X_1, Y_1) \dots (a_n, X_n, Y_n)$. If β_σ is well-formed, then $\forall i : 1 \leq i \leq n, X_i \subseteq A_{\sigma_{i-1}} \wedge Y_i \subseteq A_{\sigma_{i\rightarrow}}$.

Proof. If for some i, $X_i \not\subseteq A_{\sigma_{i-1}}$, then let $a \in X_i \setminus A_{\sigma_{i-1}}$. Now activity a_i is waiting for an obligation (a, a_i) that cannot have been produced (since $a \notin A_{\sigma_{i-1}}$). Thus $\psi(\beta_{i-1}) \not\supseteq (X_i \times \{a_i\})$ and β is not well-formed (violates W4), which is a contradiction. Similarly, if $Y_i \not\subseteq A_{\sigma_{i\rightarrow}}$, then it exists an activity a such that $a \in Y_i \setminus A_{\sigma_{i\rightarrow}}$. Now obligation (a_i, a) cannot be consumed (since $a \notin A_{\sigma_{i\rightarrow}}$) so $\psi(\beta_n) \neq \emptyset$ (violating W5), and again β is not well-formed. □

To encode arithmetically the sets X_i and Y_i for each β_σ, we use an integer variable over the domain $\{0, 1\}$ (*i.e.*, a Boolean variable, although we treat it as an integer in this section) to encode the fact that a particular activity belongs to the set. In particular we use a variable $x_{\sigma,i,(a,a_i)}$ to indicate whether activity a belongs to X_i in β_σ or not. As usual when sets are encoded using characteristic functions we use the following semantics:

$$x_{\sigma,i,(a,a_i)} = \begin{cases} 1 & \text{if } a \in X_i \text{ in } \beta_\sigma \\ 0 & \text{otherwise.} \end{cases}$$

Similarly, the variable $y_{\sigma,i,(a_i,a)}$ indicates if a belongs to Y_i in β_σ. Due to Property 2, the activity a for x variables can only be chosen among the alphabet of prefix σ_{i-1}, while in y variables it is restricted to the alphabet of the suffix of σ after a_i, *i.e.*, $A_{\sigma_{i\rightarrow}}$. We denote by $\mathcal{X}$ and $\mathcal{Y}$ the set of all x and all y variables, respectively.

We will now rewrite the conditions (W1,W2,W3,W4 and W5) of Def. 7 for a well-formed protobinding sequence $\beta_\sigma = (a_1, X_1, Y_1) \dots (a_n, X_n, Y_n)$ as inequalities using the $\mathcal{X}$ and $\mathcal{Y}$ variables.

Condition W1. In this case, part of the condition is already guaranteed, since our definition of log already assumes that the initial activity only appears once. Thus the condition simplifies to requiring that every X_i (except X_1) must be non-empty:

$$\forall i : 1 < i \leq n, \sum_{e \in A_{\sigma_{i-1}}} x_{\sigma,i,(e,a_i)} \geq 1 \tag{1}$$

Condition W2. This is the symmetrical case to W1 but with the Y_i sets. Since the uniqueness of the final activity is already guaranteed, we must only enforce that the Y_i sets (except Y_n) are non-empty:

$$\forall i : 1 \leq i < n, \sum_{e \in A_{\sigma_{i\rightarrow}}} y_{\sigma,i,(a_i,e)} \geq 1 \tag{2}$$

Condition W3. This needs no conversion, since we can directly assign the empty set to X_1 and Y_n. Note that the model does not even generate any variable in X or Y to represent these sets, since $A_{\sigma_0} = A_{\sigma_{n\rightarrow}} = \emptyset$.

Condition W4. This condition requires that the state of obligations after executing prefix β_{i-1} (*i.e.*, $\psi(\beta_{i-1})$) contains, at least, the obligations in $(X_i \times \{a_i\})$. This is the same as requiring that the number of obligations of the type (e, a_i) in $\psi(\beta_{i-1})$ is larger or equal than the number of obligations (e, a_i) in $(X_i \times \{a_i\})$. Moreover, if a_i is the last occurrence of that activity, condition W5 applies instead, since there cannot be pending obligations in the final state, so the last occurence of an activity must consume all the obligations for it. The number of such obligations in $\psi(\beta_{i-1})$ can be computed by summing the number of times the obligation has been produced minus the number of times it has been already consumed before the execution of a_i.

$$\forall i : (1 \leq i \leq n \wedge \exists j : (j > i \wedge a_j = a_i)), \forall e \in A_{\sigma_{i-1}},$$
$$\sum_{k: k<i \wedge a_k = e} y_{\sigma,k,(e,a_i)} - \sum_{m: m \leq i \wedge a_m = a_i} x_{\sigma,m,(e,a_i)} \geq 0 \tag{3}$$

Condition W5. To force that the final number of obligations must be zero we require that the number of (e, a_i) obligations is exactly zero after the last execution of a_i in the sequence. Since it is simply a stronger version of (3), it replaces (3) in the last execution of a_i.

$$\forall i : (1 \leq i \leq n \wedge \forall j\, (j > i \Rightarrow a_j \neq a_i)), \forall e \in A_{\sigma_{i-1}},$$
$$\sum_{k: k<i \wedge a_k = e} y_{\sigma,k,(e,a_i)} - \sum_{m: m \leq i \wedge a_m = a_i} x_{\sigma,m,(e,a_i)} = 0 \tag{4}$$

Definition 8 (Structural equations). *The set of* structural equations *for a C-net including the behavior of a log L, denoted* structural_equations(L), *is the set of equations obtained by adding the set of equations (1), (2), (3) and (4) for every $\sigma \in L$.*

Example 1. Consider the sequence $\sigma_\beta = abcbe$, so that $\beta = (a_1, X_1, Y_1)(a_2, X_2, Y_2)(a_3, X_3, Y_3)(a_4, X_4, Y_4)(a_5, X_5, Y_5)$ with $a_1 = a$, $a_2 = b$, $a_3 = c$, $a_4 = b$, $a_5 = e$, and $X_1 = Y_5 = \emptyset$. Table 1 shows the structural equations for each prefix in the sequence.

Table 1. Structural equations for sequence *abcbe*

$i = 1$	$A_{\sigma_0} = \emptyset, A_{\sigma_{1\rightarrow}} = \{b, c, e\}, \sigma_1 = a$
(1)	–
(2)	$y_{\sigma,1,(a,b)} + y_{\sigma,1,(a,c)} + y_{\sigma,1,(a,e)} \geq 1$
(3)	–
$i = 2$	$A_{\sigma_1} = \{a\}, A_{\sigma_{2\rightarrow}} = \{b, c, e\}, \sigma_2 = ab$
(1)	$x_{\sigma,2,(a,b)} \geq 1$
(2)	$y_{\sigma,2,(b,b)} + y_{\sigma,2,(b,c)} + y_{\sigma,2,(b,e)} \geq 1$
(3)	$y_{\sigma,1,(a,b)} - x_{\sigma,2,(a,b)} \geq 0$
$i = 3$	$A_{\sigma_2} = \{a, b\}, A_{\sigma_{3\rightarrow}} = \{b, e\}, \sigma_3 = abc$
(1)	$x_{\sigma,3,(a,c)} + x_{\sigma,3,(b,c)} \geq 1$
(2)	$y_{\sigma,3,(c,b)} + y_{\sigma,3,(c,e)} \geq 1$
(4)	$y_{\sigma,1,(a,c)} - x_{\sigma,3,(a,c)} = 0$ and $y_{\sigma,2,(b,c)} - x_{\sigma,3,(b,c)} = 0$
$i = 4$	$A_{\sigma_3} = \{a, b, c\}, A_{\sigma_{4\rightarrow}} = \{e\}, \sigma_4 = abcb$
(1)	$x_{\sigma,4,(a,b)} + x_{\sigma,4,(b,b)} + x_{\sigma,4,(c,b)} \geq 1$
(2)	$y_{\sigma,4,(b,e)} \geq 1$
(4)	$y_{\sigma,1,(a,b)} - x_{\sigma,2,(a,b)} - x_{\sigma,4,(a,b)} = 0$, $y_{\sigma,2,(b,b)} - x_{\sigma,4,(b,b)} = 0$ and $y_{\sigma,3,(c,b)} - x_{\sigma,4,(c,b)} = 0$
$i = 5$	$A_{\sigma_4} = \{a, b, c\}, A_{\sigma_{5\rightarrow}} = \emptyset, \sigma_5 = abcbe$
(1)	$x_{\sigma,5,(a,e)} + x_{\sigma,5,(b,e)} + x_{\sigma,5,(c,e)} \geq 1$
(2)	–
(4)	$y_{\sigma,1,(a,e)} - x_{\sigma,5,(a,e)} = 0$, $y_{\sigma,2,(b,e)} - y_{\sigma,4,(b,e)} - x_{\sigma,5,(b,e)} = 0$ and $y_{\sigma,3,(c,e)} - x_{\sigma,5,(c,e)} = 0$

Note that in this table some of the equations for $i = 1$ are empty since $A_{\sigma_0} = \emptyset$, a similar case to that of $i = 5$ and (2), because $A_{\sigma_{5\rightarrow}} = \emptyset$. Moreover, (4) is used instead of (3) for $i \in \{3, 4, 5\}$ because these are the last executions of activities c, b and e, respectively.

In summary, by finding the satisfying assignments to the $\mathcal{X}$ and $\mathcal{Y}$ variables in the equations arising from a log, one can derive a C-net that includes the language of the log. In terms of complexity, the number of variables that each activity occurrence generates is $|A|$, thus for a sequence σ, the total number of variables generated is $|A| \cdot |\sigma|$. Hence, the total number of variables for a log L is $|A| \cdot \sum_{\sigma \in L} |\sigma|$, which is $\mathcal{O}\left(|L| \cdot |A| \cdot \max_{\sigma \in L}\left(|\sigma|\right)\right)$. The number of equations for an activity $a_i \in \sigma$ is two ((1) and (2)) plus $|A_{\sigma_{i-1}}|$ for (3) and (4). Thus, for a sequence σ, the maximum number of equations is $\mathcal{O}(|\sigma| \cdot |A|)$ so for the whole log L is $\mathcal{O}(|L| \cdot |A| \cdot \max_{\sigma \in L}(|\sigma|))$, the same as the number of variables. Next sections illustrate how to algorithmically solve the discovery problem described in this section.

3.3 Solving Linear Constraints Using SMT

The equations presented in the previous section can be represented in different domains. In the algebraic domain, one option is to model equations (1)–(4) in a Integer Linear Programming (ILP) model (but with binary variables), and use one of the available solvers. However, such an option has an important drawback: the cost function used to minimize the solution to the problem must be linear. A possibility is to minimize the sum of all the $\mathcal{X}$ and $\mathcal{Y}$ variables. However, this will promote solutions like the immediately follows C-net, since in that C-net every activity (except the initial and final ones) always consumes one obligation and produces one obligation, thus it is not possible to have a C-net producing less obligations. Ideally we would like to express that we seek for a C-net as simple as possible and, as we will see in Sect. 3.4, we can restrict its number of arcs. However, this requires an expression involving logical disjunctions. Although these type of constraints can be encoded as linear combinations[1], they require the introduction of auxiliary variables and additional constraints.

An alternative will be to solve the problem in the Boolean domain. SMT solvers for the theory of quantifier-free bit-vector arithmetic [9], as we will see in this section, can also model equations (1)–(4) and have the advantage that they can also encode more easily the bound on the number of arcs in the C-net, as well as some other constraints (see Sect. 3.6). Since SMT solvers provide a higher degree of flexibility and our tests showed that in terms of running time ILP solvers and SMT solvers had a similar performance in our benchmarks, we have decided to use SMT to encode the problem.

Variables in $\mathcal{X}$ and $\mathcal{Y}$ are all Boolean, so obtaining a Boolean formula that represents the model is possible. Now let us show how (1), (2), (3) and (4) can be encoded as Boolean formulas. Equations (1) and (2) are trivial, since they correspond to a disjunction. For instance, the inequality (1): $\sum_{e \in A_{\sigma_{i-1}}} x_{\sigma,i,(e,a_i)} \geq 1$ can be rewritten as $\bigvee_{e \in A_{\sigma_{i-1}}} x_{\sigma,i,(e,a_i)} = 1$. Equations (3) and (4), are very similar, so we can simply focus on one of them. The key idea is to compute Boolean expressions that represent the individual bits of the sum or the subtraction of these Boolean variables. For (3) we then have to check the sign of the result (must be positive, so the most significant bit in two's complement must be zero) while for (4) the result must be zero (so all the bits of the result have to be zero).

The translation process is quite involved (this is why automated tools like [10] are used for this task), but it can be quite straightforward for the most simple cases. For instance, going back to Example 1, consider the equation

[1] For instance $z = x \vee y$ is equivalent to $z \geq x$, $z \geq y$ and $z \leq x + y$.

$y_{\sigma,1,(a,b)} - x_{\sigma,2,(a,b)} \geq 0$ ((3) for $i = 2$). Translated to a Boolean expression this is simply $y_{\sigma,1,(a,b)} \vee \overline{x_{\sigma,2,(a,b)}}$.

Once all formulas have been translated to the Boolean domain, they can be converted into CNF formulas and fed into a SAT solver. In this work (Sect. 4) we have used the STP solver [10] to convert our linear equations to CNF formulas.

3.4 Adding a Cost Function

Due to the additive nature of C-nets, reducing the number of arcs tends to restrict the language of the net. Fortunately, it is possible to encode as an SMT formula an expression that bounds the number of arcs in the derived C-net. To accomplish this we can use any of the sets of Boolean variables. Without loss of generality, we use set $\mathcal{X}$. For readability we introduce an auxiliary notation to denote all the variables in $\mathcal{X}$ that correspond to a given binding (a, b) in the sequences of a log L. Namely, $\mathcal{X}_{(a,b)}(L) = \{x_{\sigma,i,(a,b)} \mid \exists \sigma = a_1 \ldots a_{|\sigma|} \in L : a_i = b \wedge a \in A_{\sigma_{i-1}}\}$. We can now compute the number of arcs in the C-net obtained through T1, T2 and T3 (Theorem 1) using the following expression:

$$\text{number\_of\_arcs}(L) \stackrel{\text{def}}{=} \sum_{a \in A_L} \sum_{b \in A_L} \bigvee_{x \in \mathcal{X}_{(a,b)}(L)} x$$

Then, the equation bounding the number of arcs is:

$$\text{bound\_arcs}(L, l) \stackrel{\text{def}}{=} \text{number\_of\_arcs}(L) \leq l \tag{5}$$

In Sect. 3.5 we will use this equation to find the C-net whose language includes the log L and has the minimum number of arcs. Since we will explore the solution space using a binary search strategy, we need to derive lower and upper bounds on the number of arcs that the C-net can have.

An upper bound can be obtained by computing the immediately follows C-net and counting its arcs. A possible lower bound can be the maximum between $|A_L| - 1$, which is the minimum number of arcs to guarantee that all the activities in the log are connected, and the bound obtained in the following lemma:

Lemma 2. *Let A_s be a set containing all the activities that appear in second position in some sequence of log L. Similarly, let A_e be a set containing the activities that appear in previous to the last position in some sequence. Any C-net C such that $L \subseteq \mathcal{L}(C)$ satisfies:*

$$\text{arcs}(C) \geq |A_s| + |A_e| + |A_L \setminus (A_s \cup A_e)| - 2 + \max(|A_s \setminus A_e|, |A_e \setminus A_s|)$$

Proof. C has an arc from initial activity a_s to all the activities in A_s. Similarly it has an arc from every activity in A_e to the final activity a_e, otherwise there is a sequence in L that does not belong to $\mathcal{L}(C)$. This means we have already $|A_s| + |A_e|$ arcs in C. Now for activities in $A_s \cap A_e$ no further arc is mandatory, however for activities not in the intersection there must be a path from a_s to a_e (by C-net definition). Since one activity in $A_s \setminus A_e$ can be connected to another activity in $A_e \setminus A_s$, the difference set with maximum number of elements determines the number of additional arcs that have to be added (thus we must add $\max(|A_s \setminus A_e|, |A_e \setminus A_s|)$ to the number of arcs). Finally the activities not in A_s, A_e nor in $\{a_s, a_e\}$ can be arbitrarily placed, however the structure that yields the least number of additional arcs is to put them in a sequence in an already existing path from a_s to a_e, in which case one arc is added for each activity in this set. Hence $|A_L \setminus (A_s \cup A_e \cup \{a_s, a_e\})| = |A_L \setminus (A_s \cup A_e)| - 2$ arcs are added. $\square$

Depending on the characteristics of the log (mainly the sizes of A_s and A_e), this bound might be more restrictive than using simply connectedness arguments (*i.e.*, the bound $|A_L| - 1$) thus using the largest of both values potentially decreases the number of SMT problems that have to be solved.

3.5 The Algorithm

In Algorithm 1 we give the pseudocode of the proposed approach. The main idea is to build the structural equations mandatory to any C-net whose language includes a given log L, and then bound the number of arcs allowed in the solution. Following the outcome of the SMT solver, the bound is changed, so that we minimize the number of arcs using a binary search strategy. To obtain reasonable initial bounds, we use Lemma 2 for a lower bound (line 5) and the number of arcs in the immediately follows C-net for the upper bound (line 6).

Algorithm 1. Discover minimal C-net

```
 1: procedure DISCOVERMINCNET($L$)
 2:     $C = \langle A, a_s, a_e, I, O \rangle \leftarrow C_{\mathrm{IF}}(L)$                ▷ See Sect. 2.3
 3:     $A_s \leftarrow \{a \mid (a_s, a) \in \mathrm{arcs}(C)\}$
 4:     $A_e \leftarrow \{a \mid (a, a_e) \in \mathrm{arcs}(C)\}$
 5:     $min \leftarrow \max(|A|, |A_s| + |A_e| + |A \setminus (A_s \cup A_e)| - 2 + \max(|A_s \setminus A_e|, |A_e \setminus A_s|))$
 6:     $max \leftarrow |\mathrm{arcs}(C)| - 1$
 7:     $E_s \leftarrow \mathrm{structural\_equations}(L)$
 8:     while $min \leq max$ do
 9:         $avg \leftarrow \lfloor (min + max)/2 \rfloor$
10:         $E \leftarrow E_s \cup \{\mathrm{bound\_arcs}(L, avg)\}$                ▷ Add (5)
11:         $feasible, solutions \leftarrow \mathrm{solve}(E)$                ▷ Call SMT solver
12:         if $feasible$ then
13:             $C \leftarrow \mathrm{extract\_cnet}(solutions)$                ▷ Model feasible
14:             $max \leftarrow |\mathrm{arcs}(C)| - 1$                ▷ Since $|\mathrm{arcs}(C)| \leq avg$
15:         else
16:             $min \leftarrow avg + 1$                ▷ Model unfeasible
17:         end if
18:     end while
19:     return $C$
20: end procedure
```

Note that, although the minimum number of arcs to guarantee that all activities are connected is $|A| - 1$, the minimum bound in the algorithm is set to $|A|$. This is because there is a single model that has $|A| - 1$ arcs, which corresponds to a sequence of activities. If this model is feasible, then it should have been already found in C_{IF}, thus $|\mathrm{arcs}(C)| = |A| - 1$ and the algorithm would never enter the loop and return C_{IF}. On the other hand, if $|\mathrm{arcs}(C)| > |A| - 1$, then there is no feasible model with just $|A| - 1$ arcs, thus the minimum search bound can be set to $|A|$.

The algorithm contains two calls to functions not yet introduced. One is function `solve`(E) which simply calls the SMT solver on the set of equations E and returns two values: *feasible* that is a Boolean value indicating whether the solver found a solution to the equations in E and *solutions* that contains the values of the $\mathcal{X}$ and $\mathcal{Y}$ variables in case the problem was feasible. The other function, `extract_cnet`$(solutions)$, simply builds a C-net out of the values of the variables in sets $\mathcal{X}$ and $\mathcal{Y}$ using the principles explained in Theorem 1.

Theorem 2. *Let C be the C-net returned by Algorithm 1 executed on a log L. The language of C includes L and there is no other C-net including L that has less arcs than C.*

Proof. Since the equations E_s represent all possible well-formed protobinding sequences of L, any valid solution is a set B of well-formed protobinding sequences of L. Using Theorem 1 on B (the `extract_cnet` function) we obtain the C-net C whose set of valid binding sequences includes B (thus its language includes L) and has the smallest number of arcs (Corollary 1). Since we simply add a restriction (`bound_arcs`(L, avg)) on the maximum number of arcs that the sequences in B induce on C, by performing a binary search we guarantee that no other C-net whose language includes L can have fewer arcs than C. $\square$

3.6 Encoding Other Types of Constraints

The approach of Sect. 3.5 does not give any guarantee on the amount of additional behavior that the generated C-net might exhibit. For instance, consider the log $\{abcdez, abdcez\}$. The C-net whose language contains only this log is shown in Fig. 4(a). However there are other C-nets with six arcs that also contain the log (as well as some other sequences), like the one in (b). Ideally we would like to obtain the simplest C-net that adds the least amount of additional behavior. While restricting the language accepted by a Petri net is straightforward, the same operation in C-nets is much more difficult given their additive nature and the fact that their language is not prefix closed.

The basic problem is that in C-nets we could only exclude complete sequences (notice that there may be infinitely many sequences starting with the initial activity and ending with the final activity), rather than prefixes (known as *wrong continuations* [11] or *faulty words* [12]) as in Petri nets. Since it is not possible to exclude an infinite number of complete sequences, we have to resort to some heuristics to favor the selection of C-nets with a more particular language.

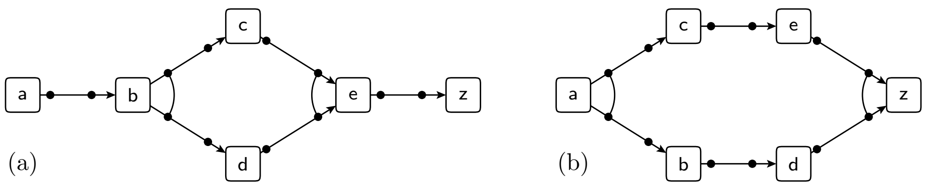


Fig. 4. Two possible C-nets with the minimum amount of arcs for log $\{abcdez, abdcez\}$

We will use three approaches: the first one penalizes the activities whose input binding set does not contain activities that are near enough in a sequence, the second limits the amount of different input and/or output bindings per activity, and the third restricts the set of activities for which an activity can generate or consume obligations. This latter approach is fundamental to tackle some of our largest benchmarks, since it can greatly reduce the amount of variables in the model to solve. Note that, using the following heuristics, the guarantee that no other C-net whose language includes the log can have fewer arcs is lost.

Restricting the First Occurrence of Activities. To check whether two activities are nearby in a sequence, we take into account the other sequences in the log and the position that the activity occupies in all other sequences that share a prefix with the current sequence. For instance in the log used in Fig. 4, $\{abcdez, abdcez\}$, activity d is executed in the fourth position in the first sequence, but there is another sequence sharing a prefix ab in which it appears in the third position. In this case, we consider that the last activity in the prefix, namely activity b in second position in both sequences, is the first point in which an obligation for d can be generated, otherwise d could potentially be executed prior to this point (*e.g.*, for instance just after the execution of a). Consequently, we would penalize activity d if its input binding does not contain at least one of the activities after the first position in any of the two sequences, namely activities b, c, d, e and z. That is, if its input binding is simply $\{a\}$ we would add one to the penalty function (but, for instance, we would not penalize $\{a, b\}$ or $\{c\}$). Summing all the penalties for the first occurrence of every activity in each sequence, we obtain an expression that can be bounded, similarly to (5), and added into the SMT problem. In our example, the C-net of 4(b) would have a penalty of one (because of activity c), while the C-net in (a) has a zero penalty.

Limiting the Input/Output Bindings per Activity. The second approach involves limiting the number of input and/or output bindings per activity. Auxiliary variables are used for this task. We illustrate this point using variables $\mathcal{X}$, since the strategy for the $\mathcal{Y}$ variables is identical. Assume that two input bindings are allowed for each activity. For each variable $x_{\sigma,i,(a,b)}$ involving an obligation (a, b) we generate one variable $i_{k,(a,b)}$ for each one of the k input bindings we allow. Since in this case $k = 2$, this means that we would have $i_{1,(a,b)}$ and $i_{2,(a,b)}$. Now for every input binding set in position i of sequence σ, we will enforce that $\bigvee_{1 \leq j \leq k} \left(\bigwedge_{a \in A_{\sigma_i}} x_{\sigma,i,(a,a_i)} = i_{j,(a,a_i)} \wedge \bigwedge_{a \notin A_{\sigma_i}} x_{\sigma,i,(a,a_i)} = 0 \right)$.

Limiting the Obligation Alphabet. We have seen in Property 1 that the set of input bindings for an activity a_i in sequence σ is a subset of A_{σ_i}. However this allows for very long causal dependencies, that are rather unfrequent in most of the logs. We can simplify the C-net discovery problem by bounding this set using a window: we will only allow activity a_i to consume obligations generated by activities that are at most at distance w of any occurrence of a_i in any of the sequences of the log. Using a window size of one ($w = 1$) the number of variables can be dramatically reduced (the same principles are used to restrict the output binding sets), allowing the discovery of larger benchmarks. However, when using this approach complex causalities between activities can be missed.

4 Experiments

First of all, Table 2 describes some of the examples used in our experiments. We have used logs obtained by simulation of C-nets coming from [2], or that we have created to represent a variety of non-trivial behavior. Other benchmarks are logs introduced in [13], for which we have manually created a C-net (to set a target C-net to achieve) from a Petri net generated by a discovery tool using the theory of regions [12]. The table also includes the following information: $|L|$ is the number of distinct sequences in the log, $|\sigma_m|$ is the length of the largest sequence and $|A|$ is the size of the alphabet of activities.

We have implemented Algorithm 1 in a prototype tool that uses the `STP` solver [10] as the underlying SMT solver[2]. We have compared it with the *Flexible Heuristic Miner* (`FHM`) [3] which is able to discover a formalism similar to C-nets (called *flexible heuristic nets* from a log [3]) . Table 3 shows the results on some small log examples with the following information: *arcs* is the number of arcs of the final C-net, T is the elapsed time (in seconds) required to complete the discovery process, *id* indicates if the obtained C-net was identical to the original one, in the case where the log originated from a C-net, or has the same language as a Petri net found using the theory of regions, *cf* is the *cost-based fitness per case* metric of [14] where 1.0 indicates that all sequences in the log belong to the language of the C-net, and the smaller the value is the less sequences are reproducible by the C-net, $|\mathcal{X} \cup \mathcal{Y}|$ is the number of Boolean variables used to encode the SMT problem, $|E|$ is the number of equations that the SMT problem contains, *bounds* is the initial range in the number of arcs where the binary search must take place, *it* is the number of iterations to obtain this C-net, and column *heur* indicates if some of the heuristics in Sect. 3.6 was used, where f refers to restricting the first occurrence of activities and i (o) to limiting the number of input (output) bindings per activity. In this first set of experiments we did not limit the obligation alphabet.

The results on these small benchmarks show that the approach is, in general, able to derive valuable C-nets. In fact the quality of the discovered nets is much better than the ones derived using `FHM`. For instance, the latter generates four

[2] `STP` translates the SMT formula to a SAT formula and then uses the miniSAT solver, but any other SAT (or incremental SAT) tool could have been used as backend.

Table 2. Logs used in the experiments

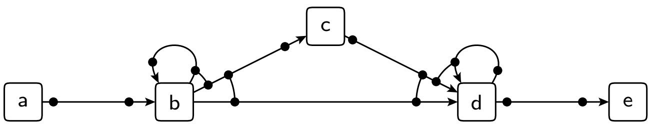
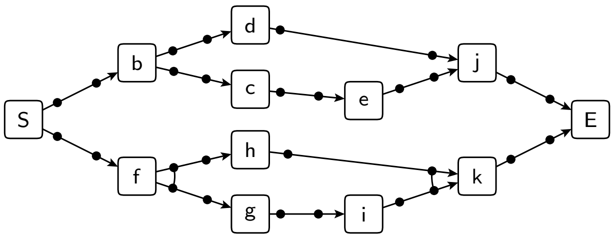
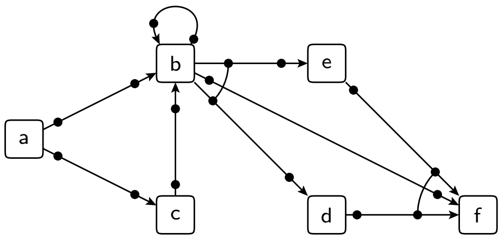
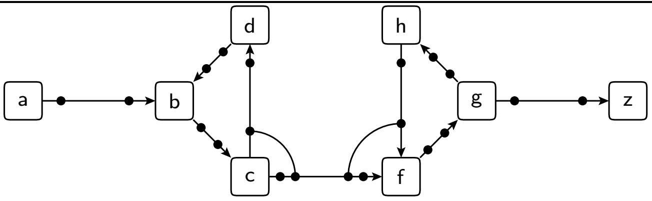
aalst2b [2]	$\lvert L\rvert = 10,\ \lvert\sigma_m\rvert = 5,\ \lvert A\rvert = 5$
abcde abcbcdde abbbcccddde abbcdcde abcbdcde abcbbdccdde abbccdde abbcbdcdcde	
mixedXorAnd	$\lvert L\rvert = 8,\ \lvert\sigma_m\rvert = 11,\ \lvert A\rvert = 5$
abcdez abdefbcez abdceefbdfbcez	See Fig. 3(a)
a12f0n00_5 [13]	$\lvert L\rvert = 5,\ \lvert\sigma_m\rvert = 7,\ \lvert A\rvert = 12$
SbcejE SbdjE SfghikE SfgihkE SfhgikE	
optional1	$\lvert L\rvert = 11,\ \lvert\sigma_m\rvert = 8,\ \lvert A\rvert = 6$
abf acbdef abbedf abbbbedf acbedf acbbbedf acbbf abbbbdef abbdef acbbbdef acbf	
cycles	$\lvert L\rvert = 7,\ \lvert\sigma_m\rvert = 18,\ \lvert A\rvert = 8$
abcfgz abcdbcfghfgz abcfghdbcfgz abcfgdhbcfgz abcfdgbhcfgz abcfdghbcfgz abcdbcdbcfghfghfgz	

Table 3. Results of discovery algorithm on small examples

	FHM				Algorithm 1								
Benchmark	arcs	T	id	cf	$\|\mathcal{X} \cup \mathcal{Y}\|$	$\|E\|$	bounds	arcs	it	T	heur	id	cf
aalst1 (Fig. 2)	6	0.1	n	0.71	156	147	$[6, 11]$	6	2	0.3	–	y	1.0
aalst2b	7	0.0	n	0.24	156	147	$[5, 9]$	6	3	0.2	–	n	1.0
mixedXorAnd	8	0.0	n	0.12	219	162	$[7, 11]$	8	3	0.2	–	y	1.0
a12f0n00_5	14	0.2	n	0.87	176	143	$[12, 17]$	14	3	0.1	f	y	1.0
optional1	7	0.0	n	0.27	413	264	$[6, 10]$	9	2	0.1	f,o	y	1.0
cycles	9	0.1	n	0.09	839	542	$[8, 17]$	9	3	1.3	f,i	y	1.0
a22f0n00_1	34	0.5	n	0.37	28898	18942	$[22, 166]$	≤ 39	≥ 4	> 1h	–	–	–

C-nets with empty language. In contrast, Algorithm 1 always generates C-nets whose language contains the given log (fitness=1.0), not only this but also it rediscovers the original C-nets in most of the cases. However two logs are not successfully discovered: for the `aalst2b` benchmark, we obtain a C-net equal to the one in Table 2, but without the arc between b and d; on the other hand, the largest benchmark in this table (`a22f0n00_1` from [13]) could not be discovered in the one hour limit used in our experiments. Although it seems reasonable to assume that the large number of variables and equations is the responsible of this fact, a more careful evaluation shows that this is not the main factor. For instance, in Fig. 5, we can see the time used by the solver to solve the set of equations, as the bound in the number of arcs allowed is reduced. Initially, the solver finds quickly a solution, but as the bound approaches the lower limit, the time needed grows exponentially. The reason is simple: the set of valid solutions diminishes as the bound also reduces, and the solver has to spend more time searching. Note that the set of equations to solve in all these cases is exactly the same (same variables, same equations) with the only exception that the constant used to bound the number of arcs is different.

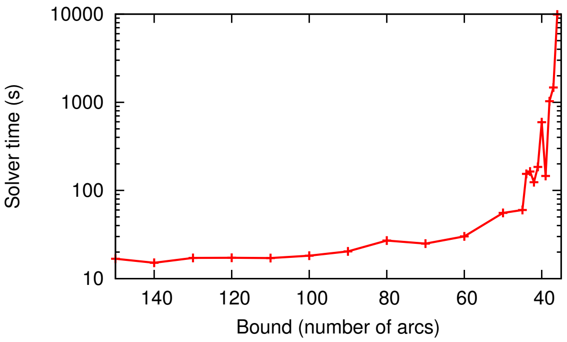


Fig. 5. `stp` solver times for the a22f0n00_1 log

To be able to process larger benchmarks we have to resort to our last heuristic (limiting the obligation alphabet). Table 4 shows the results for our previous benchmarks as well as some larger examples also from [13]. In this case we have not used any other heuristic. Despite the fact that the original models were discovered only in three of the benchmarks, in each case the model found was actually the original but in which some additional input and output binding sets were present. This strongly suggests that a

Table 4. Results of discovery algorithm when heuristics to limit the number of variables are used (activity window of size 1)

Benchmark	$\lvert L\rvert$	$\lvert\sigma_m\rvert$	$\lvert A\rvert$	$\lvert\mathcal{X}\cup\mathcal{Y}\rvert$	$\lvert E\rvert$	bounds	arcs	it	T	id	cf
aalst1	10	5	5	136	137	$[6, 11]$	6	2	0.0	y	1.0
aalst2b	8	11	5	240	246	$[5, 9]$	6	3	0.1	n	1.0
mixedXorAnd	3	14	7	89	98	$[7, 11]$	8	3	0.0	y	1.0
a12f0n00_5	5	7	12	72	91	$[12, 17]$	14	3	0.0	y	1.0
optional1	11	8	6	229	220	$[6, 10]$	9	2	0.0	n	1.0
cycles	7	18	8	265	288	$[8, 17]$	9	3	0.1	y	1.0
a22f0n00_1	99	46	22	12827	10369	$[22, 166]$	34	7	9.3	n	1.0
a22f0n00_5	836	76	22	121281	97429	$[22, 183]$	34	7	264.9	n	1.0
a32f0n00_1	100	73	32	26378	19049	$[32, 362]$	46	8	35.4	n	1.0
a42f0n00_1	100	58	42	48432	31815	$[42, 735]$	63	9	248.4	n	1.0

strategy that minimizes the number of such sets would greatly improve the results, as well as the overall readability of the derived C-nets.

5 Conclusion and Future Work

This paper has presented an algorithm to derive a C-net from a set of traces, which guarantees minimality in the number of arcs. As future work, we plan to incorporate in the algorithm new ideas on how to bound the language of the C-net obtained. Also, high-level strategies that can make the approach able to handle industrial examples will be considered in the future.

References

1. van der Aalst, W.M.P.: Process Mining - Discovery, Conformance and Enhancement of Business Processes. Springer (2011)
2. van der Aalst, W.M.P., Adriansyah, A., van Dongen, B.: Causal Nets: A Modeling Language Tailored towards Process Discovery. In: Katoen, J.-P., König, B. (eds.) CONCUR 2011 – Concurrency Theory. LNCS, vol. 6901, pp. 28–42. Springer, Heidelberg (2011)
3. Weijters, A.J.M.M., Ribeiro, J.T.S.: Flexible heuristics miner (FHM). In: IEEE CIDM, pp. 310–317 (2011)
4. van der Aalst, W.M.P., de Medeiros, A.K.A., Weijters, A.J.M.M.T.: Genetic Process Mining. In: Ciardo, G., Darondeau, P. (eds.) ICATPN 2005. LNCS, vol. 3536, pp. 48–69. Springer, Heidelberg (2005)
5. Weijters, A., van der Aalst, W.M.P., de Medeiros, A.A.: Process mining with the heuristics miner-algorithm. Technical Report WP 166, BETA Working Paper Series, Eindhoven University of Technology (2006)
6. van der Aalst, W.M.P., Weijters, T., Maruster, L.: Workflow mining: Discovering process models from event logs. IEEE TKDE 16, 1128–1142 (2004)
7. Solé, M., Carmona, J.: A high-level strategy for C-net discovery. In: ACSD 2012 (in press, 2012)

8. Muñoz-Gama, J., Carmona, J.: A Fresh Look at Precision in Process Conformance. In: Hull, R., Mendling, J., Tai, S. (eds.) BPM 2010. LNCS, vol. 6336, pp. 211–226. Springer, Heidelberg (2010)
9. Jha, S., Limaye, R., Seshia, S.A.: Beaver: Engineering an Efficient SMT Solver for Bit-Vector Arithmetic. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 668–674. Springer, Heidelberg (2009)
10. Ganesh, V., Dill, D.L.: A Decision Procedure for Bit-Vectors and Arrays. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 519–531. Springer, Heidelberg (2007)
11. 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)
12. 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. 364–383. Springer, Heidelberg (1995)
13. van der Werf, J.M.E.M., van Dongen, B.F., Hurkens, C.A.J., Serebrenik, A.: Process Discovery Using Integer Linear Programming. In: van Hee, K.M., Valk, R. (eds.) PETRI NETS 2008. LNCS, vol. 5062, pp. 368–387. Springer, Heidelberg (2008)
14. Adriansyah, A., van Dongen, B., van der Aalst, W.M.P.: Conformance checking using cost-based fitness analysis. In: Enterprise Distributed Object Computing Conference (EDOC), pp. 55–64 (2011)

Decomposing Process Mining Problems Using Passages

Wil M.P. van der Aalst

Department of Mathematics and Computer Science,
Technische Universiteit Eindhoven, The Netherlands
www.vdaalst.com

Abstract. Process discovery—discovering a process model from example behavior recorded in an event log—is one of the most challenging tasks in process mining. Discovery approaches need to deal with competing quality criteria such as *fitness*, *simplicity*, *precision*, and *generalization*. Moreover, event logs may contain low frequent behavior and tend to be far from complete (i.e., typically only a fraction of the possible behavior is recorded). At the same time, models need to have formal semantics in order to reason about their quality. These complications explain why dozens of process discovery approaches have been proposed in recent years. Most of these approaches are time-consuming and/or produce poor quality models. In fact, simply checking the quality of a model is already computationally challenging.

This paper shows that process mining problems can be *decomposed* into a set of smaller problems after determining the so-called *causal structure*. Given a causal structure, we partition the activities over a collection of *passages*. Conformance checking and discovery can be done *per passage*. The decomposition of the process mining problems has two advantages. First of all, the problem can be distributed over a network of computers. Second, due to the exponential nature of most process mining algorithms, decomposition can significantly reduce computation time (even on a single computer). As a result, conformance checking and process discovery can be done much more efficiently.

Keywords: process mining, conformance checking, process discovery, distributed computing, business process management.

1 Introduction

A recent report by the McKinsey Global Institute (MGI) called “Big Data: The Next Frontier for Innovation, Competition, and Productivity” describes the spectacular growth of data and the potential economic value of such data in different industry sectors [28]. MGI estimates that enterprises globally stored more than 7 exabytes of new data on disk drives in 2010, while consumers stored more than 6 exabytes of new data on devices such as PCs and notebooks. Despite the growth of storage space, it is impossible to store all event data. The global capacity to store data has been estimated in various studies. For example, a recent study in *Science* suggests that the total global storage capacity increased from 2.6 exabytes in 1986 to 295 exabytes in 2007 [25].

The incredible growth of event data provides new opportunities for process analysis. As more and more actions of people, organizations, and devices are recorded, there are ample opportunities to analyze processes based on the footprints they leave in event logs. In fact, the analysis of *hand-made* process models will become less important given the omnipresence of event data. This is the reason why *process mining* is one of the “hot” topics in Business Process Management (BPM). Process mining aims to *discover, monitor and improve real processes by extracting knowledge from event logs* readily available in today’s information systems [2].

Starting point for process mining is an *event log*. Each event in such a log refers to an *activity* (i.e., a well-defined step in some process) and is related to a particular *case* (i.e., a *process instance*). The events belonging to a case are *ordered* and can be seen as one “run” of the process. It is important to note that an event log contains only example behavior, i.e., we cannot assume that all possible runs have been observed. In fact, an event log often contains only a fraction of the possible behavior [2].

The growing interest in process mining is illustrated by the *Process Mining Manifesto* [26] recently released by the *IEEE Task Force on Process Mining*. This manifesto is supported by 53 organizations and 77 process mining experts contributed to it. The active contributions from end-users, tool vendors, consultants, analysts, and researchers illustrate the significance of process mining as a bridge between data mining and business process modeling.

Petri nets are often used in the context of process mining. Various algorithms employ Petri nets as the internal representation used for process mining. Examples are the region-based process discovery techniques [6, 13, 19, 33, 36], the α algorithm [7], and various conformance checking techniques [8, 30–32]. Other techniques use alternative internal representations (C-nets, heuristic nets, etc.) that can easily be converted to (labeled) Petri nets [2].

In this paper, we focus on the following two main process mining problems:

– *Process discovery problem*: Given an event log consisting of a collection of traces (i.e., sequences of events), construct a Petri net that “adequately” describes the observed behavior.
– *Conformance checking problem*: Given an event log and a Petri net, diagnose the differences between the observed behavior (i.e., traces in the event log) and the modeled behavior (i.e., firing sequences of the Petri net).

Both problems are formulated in terms of Petri nets. However, other process notations could be used, e.g., BPMN models, BPEL specifications, UML activity diagrams, Statecharts, C-nets, heuristic nets, etc. In fact, also different types of Petri nets can be employed, e.g., safe Petri nets, labeled Petri nets, free-choice Petri nets, etc.

Process mining problems tend to be very challenging. There are obvious challenges that also apply to many other data mining and machine learning problems, e.g., dealing with noise, concept drift, and the need to explore a large and complex search space. For example, event logs may contain millions of events. Moreover, there are also some specific problems that make process discovery even more challenging:

– there are *no negative examples* (i.e., a log shows what has happened but does not show what could not happen);
– due to concurrency, loops, and choices the *search space has a complex structure* and the log typically contains only a *fraction* of all possible behaviors;

- there is *no clear relation between the size of a model and its behavior* (i.e., a smaller model may generate more or less behavior although classical analysis and evaluation methods typically assume some monotonicity property); and
- there is a need to balance between four (often) *competing quality criteria* (see Section 3): (1) *fitness* (be able to generate the observed behavior), (2) *simplicity* (avoid large and complex models), (3) *precision* (avoid "underfitting"), and (4) *generalization* (avoid "overfitting").

Process discovery and conformance checking are related problems. This becomes evident when considering *genetic* process discovery techniques [15, 29]. In each generation of models generated by the genetic algorithm, the conformance of every individual model in the population needs to be assessed (the so-called fitness evaluation). Models that fit well with the event log are used to create the next generation of candidate models. Poorly fitting models are discarded. The performance of genetic process discovery techniques will only be acceptable if dozens of conformance checks can be done per second (on the whole event log). This illustrates the need for efficient process mining techniques.

Dozens of process discovery [2, 6, 7, 11, 13, 18, 19, 21, 24, 29, 33, 35, 36] and conformance checking [3, 8–10, 16, 22, 24, 30–32, 34] approaches have been proposed in literature. Despite the growing maturity of these approaches, the quality and efficiency of existing techniques leave much to be desired. State-of-the-art techniques still have problems dealing with large and/or complex event logs and process models. Therefore, we proposed a *divide and conquer approach for process mining*. This approach uses a new concept: *passages*. A passage is a pair of two sets of activity nodes (X, Y) such that $X\bullet = Y$ (i.e., the activity nodes in X influence the enabling of the activity nodes in Y) and $X = \bullet Y$ (i.e., the activity nodes in Y are influenced by the activity nodes in X). The notion of passages will be formalized in terms of graphs and labeled Petri nets. Passages can be used to *decompose process discovery and conformance checking problems into smaller problems*. By localizing process mining techniques to passages, more refined techniques can be used. Assuming that the event log and process model can be decomposed into many passages, substantial speedups are possible. Moreover, passages can also be used to *distribute* process mining problems over a network of computers (e.g., a grid or cloud infrastructure).

This paper focuses on the theoretical foundations of process mining based on passages. Section 2 introduces various preliminaries, including the new notion of passages on graphs, event logs, and Petri nets. Section 3 discusses quality criteria for process mining, e.g., the fitness notion is introduced. The notion of passages is used in Section 4 to decompose the overall conformance checking problem into a set of local conformance checking problems. Section 5 shows how the same ideas can be used for process discovery, i.e., after determining the causal structure and related passages, the overall process discovery problem can be decomposed into a set of local process discovery problems. Related work is discussed in Section 6. Section 7 concludes the paper.

2 Preliminaries

This section introduces basic concepts related to Petri nets, WF-nets, and event logs. Moreover, we introduce the notation of *passages* on arbitrary graphs. This notion will be used to decompose process mining problems into a set of smaller problems.

2.1 Graphs, Passages, and Paths

First, we introduce basic graphs notations. We will use graphs to represent process models (i.e., Petri nets) and the causal structure (also referred to as skeleton) of processes.

Definition 1 (Graph). *A graph is a pair $G = (N, E)$ comprising a set N of nodes and a set $E \subseteq N \times N$ of edges.*

For a graph $G = (N, E)$ and $n \in N$, we define preset $\overset{G}{\bullet} n = \{n' \in N \mid (n', n) \in E\}$ (direct predecessors) and postset $n \overset{G}{\bullet} = \{n' \in N \mid (n, n') \in E\}$ (direct successors). This can be generalized to sets, i.e., for $X \subseteq N$: $\overset{G}{\bullet} X = \cup_{n \in X} \overset{G}{\bullet} n$ and $X \overset{G}{\bullet} = \cup_{n \in X} n \overset{G}{\bullet}$. The superscript G can be omitted if the graph is clear from the context.

To decompose process mining problems into smaller problems, we partition process models using the notion *passages* introduced in this paper. A passage is a pair of non-empty sets of nodes (X, Y) such that the set of direct successors of X is Y and the set of direct predecessors of Y is X.

Definition 2 (Passage). *Let $G = (N, E)$ be a graph. $P = (X, Y)$ is a passage if and only if $\emptyset \neq X \subseteq N$, $\emptyset \neq Y \subseteq N$, $X \overset{G}{\bullet} = Y$, and $X = \overset{G}{\bullet} Y$. $pas(G)$ is the set of all passages of G.*

Consider the sets $X = \{b, c, d\}$ and $Y = \{d, e, f\}$ in Fig. 1 (for the moment ignore the numbers in the graph). $X\bullet = \{b, c, d\}\bullet = \{d, e, f\} = Y$ and $X = \{b, c, d\} = \bullet\{d, e, f\} = \bullet Y$, so (X, Y) is indeed a passage.

A *weak passage* is a pair (X, Y) such that $\emptyset \neq X \cup Y \subseteq N$, $X \overset{G}{\bullet} \subseteq Y$, and $\overset{G}{\bullet} Y \subseteq X$, i.e., X may contain nodes without predecessors and Y may contain nodes without successors. Note that any passage is also a weak passage but not vice versa. In the remainder, we only consider passages.

Definition 3 (Operations on Passages). *Let $P_1 = (X_1, Y_1)$ and $P_2 = (X_2, Y_2)$ be two passages.*

- *$P_1 \leq P_2$ if and only if $X_1 \subseteq X_2$ and $Y_1 \subseteq Y_2$,*
- *$P_1 < P_2$ if and only if $P_1 \leq P_2$ and $P_1 \neq P_2$,*
- *$P_1 \cup P_2 = (X_1 \cup X_2, Y_1 \cup Y_2)$,*
- *$P_1 \setminus P_2 = (X_1 \setminus X_2, Y_1 \setminus Y_2)$.*

The union of two passages $P_1 \cup P_2$ is again a passage. The difference of two passages $P_1 \setminus P_2$ is a passage if $P_2 < P_1$.

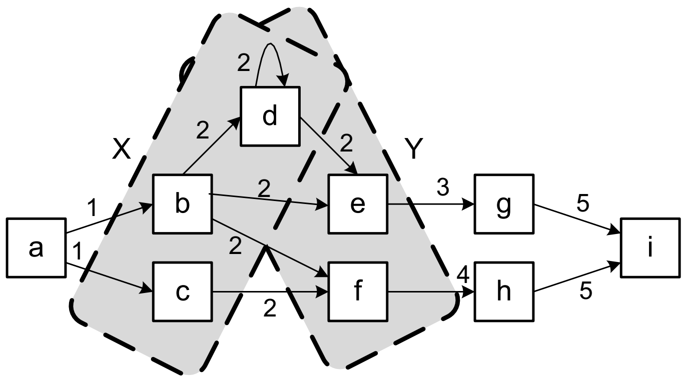


Fig. 1. A graph with five minimal passages: $P_1 = (\{a\}, \{b, c\})$, $P_2 = (\{b, c, d\}, \{d, e, f\})$, $P_3 = (\{e\}, \{g\})$, $P_4 = (\{f\}, \{h\})$, and $P_5 = (\{g, h\}, \{i\})$. Passage P_2 is highlighted and edges carry numbers to refer to the minimal passage they belong to.

Lemma 1 (Properties of Passages). *Let $G = (N, E)$ be a graph with passages $P_1, P_2 \in pas(G)$.*

– *$P_1 \cup P_2$ is a passage.*
– *If $P_2 < P_1$, then $P_1 \setminus P_2$ is a passage.*

Proof. Let $P_1 = (X_1, Y_1)$ and $P_2 = (X_2, Y_2)$ be two passages.

For $P_3 = (X_3, Y_3) = P_1 \cup P_2$ we need to prove: $\emptyset \neq X_3 \subseteq N$, $\emptyset \neq Y_3 \subseteq N$, $X_3\bullet = Y_3$, and $X_3 = \bullet Y_3$. This trivially holds because $X_3\bullet = (X_1 \cup X_2)\bullet = X_1 \bullet \cup X_2\bullet = Y_1 \cup Y_2 = Y_3$ and $\bullet Y_3 = \bullet(Y_1 \cup Y_2) = \bullet Y_1 \cup \bullet Y_2 = X_1 \cup X_2 = X_3$.

Assume that $P_2 < P_1$ and $P_3 = (X_3, Y_3) = P_1 \setminus P_2$. Again we need to prove that $\emptyset \neq X_3 \subseteq N$, $\emptyset \neq Y_3 \subseteq N$, $X_3\bullet = Y_3$, and $X_3 = \bullet Y_3$. There is a $(x, y) \in E$ with $x \in X_3$ and $y \in Y_3$. Otherwise, $P_2 \not< P_1$. Hence, $X_3 \neq \emptyset$ and $Y_3 \neq \emptyset$. Observe that $X_2 \bullet \cap X_3\bullet = \emptyset$ and $\bullet Y_2 \cap \bullet Y_3 = \emptyset$ because P_2 is a passage. Moreover, $X_3\bullet \subseteq Y_1$ and $\bullet Y_3 \subseteq X_1$. Hence, $X_3\bullet = (X_1 \setminus X_2)\bullet = X_1 \bullet \setminus X_2\bullet = Y_1 \setminus Y_2 = Y_3$. $\bullet Y_3 = \bullet(Y_1 \setminus Y_2) = \bullet Y_1 \setminus \bullet Y_2 = X_1 \setminus X_2 = X_3$. Therefore, P_3 is indeed a passage. $\square$

Since the union of two passages is again a passage, it is interesting to consider *minimal passages*. A passage is *minimal* if it does not "contain" a smaller passage.

Definition 4 (Minimal Passage). *Let $G = (N, E)$ be a graph with passages $pas(G)$. $P \in pas(G)$ is* minimal *if there is no $P' \in pas(G)$ such that $P' < P$. $pas_{min}(G)$ is the set of minimal passages.*

Figure 1 contains five minimal passages. The sets X and Y highlight minimal passage $P_2 = (\{b, c, d\}, \{d, e, f\})$. The edges in Fig. 1 have numbers corresponding to the passage they belong to, e.g., edges (a, b) and (a, c) have a label "1" showing that they belong to passage $P_1 = (\{a\}, \{b, c\})$. Here we already use the property that an edge belongs to precisely one minimal passage. In fact, a minimal passage is uniquely identified by any of its elements as is shown next.

Lemma 2. *Let $G = (N, E)$ be a graph and $(x, y) \in E$. There is precisely one minimal passage $P_{(x,y)} = (X, Y) \in pas_{min}(G)$ such that $x \in X$ and $y \in Y$.*

Proof. Construct $P_{(x,y)} = (X, Y)$ as follows. Initially: $X := \{x\}$ and $Y := \{y\}$. Then repeat $X := X \cup \bullet Y$ and $Y := Y \cup X\bullet$ until X and Y do not change anymore. The algorithm will end because there are finitely many nodes. When it ends $X = \bullet Y$ and $Y = X\bullet$. Hence, $P_{(x,y)} = (X, Y)$ is passage. No unnecessary elements are added to X and Y, so (X, Y) is minimal and there is precisely one such mininal passage for $(x, y) \in E$. □

Passages define an equivalence relation on the edges in a graph: $(x_1, y_1) \sim (x_2, y_2)$ if and only if $P_{(x_1,y_1)} = P_{(x_2,y_2)}$. It is easy to see that $\sim$ is reflexive (i.e., $(x, y) \sim (x, y)$) , symmetric (i.e., $(x_1, y_1) \sim (x_2, y_2)$ if and only if $(x_2, y_2) \sim (x_1, y_1)$), and transitive (i.e., $(x_1, y_1) \sim (x_2, y_2)$ and $(x_2, y_2) \sim (x_3, y_3)$ implies $(x_1, y_1) \sim (x_3, y_3)$). In Fig. 1 $(b, d) \sim (b, e) \sim (b, f) \sim (c, f) \sim (d, d) \sim (d, e)$, i.e., the arcs having label "2" form an equivalence class.

For any $\{(x, y), (x', y), (x, y')\} \subseteq E$: $P_{(x,y)} = P_{(x',y)} = P_{(x,y')}$, i.e., $P_{(x,y)}$ is uniquely determined by x and $P_{(x,y)}$ is also uniquely determined by y. Moreover, $pas_{min}(G) = \{P_{(x,y)} \mid (x, y) \in E\}$.

We use the notation $x \overset{\sigma:E\#Q}{\rightsquigarrow} y$ to state that there is a non-empty path σ from node x to node y in the graph $G = (N, E)$ where the set of intermediate nodes visited by path σ does not include any nodes in Q.

Definition 5 (Path). *Let $G = (N, E)$ be a graph with $x, y \in N$ and $Q \subseteq N$. $x \overset{\sigma:E\#Q}{\rightsquigarrow} y$ if and only if there is a sequence $\sigma = \langle n_1, n_2, \ldots n_k\rangle$ with $k > 1$ such that $x = n_1$, $y = n_k$, for all $1 \leq i < k$: $(n_i, n_{i+1}) \in E$, and for all $1 < i < k$: $n_i \notin Q$. Derived notations:*

- *$x \overset{E\#Q}{\rightsquigarrow} y$ if and only if there exists a path σ such that $x \overset{\sigma:E\#Q}{\rightsquigarrow} y$,*
- *$x \overset{\sigma:E}{\rightsquigarrow} y$ is a shorthand for $x \overset{\sigma:E\#Q}{\rightsquigarrow} y$ with $Q = \emptyset$,*
- *$nodes(x \overset{E\#Q}{\rightsquigarrow} y) = \{n \in \sigma \mid \exists_{\sigma \in N^*}\, x \overset{\sigma:E\#Q}{\rightsquigarrow} y\}$, and*
- *for $X, Y \subseteq N$: $nodes(X \overset{E\#Q}{\rightsquigarrow} Y) = \cup_{(x,y)\in X\times Y}\, nodes(x \overset{E\#Q}{\rightsquigarrow} y)$.*

Consider the graph $G = (N, E)$ in Fig. 1 to illustrate these notions. $a \overset{E\#Q}{\rightsquigarrow} i$ holds for $Q = \{b, d, e, g\}$ because of the path $\sigma = \langle a, c, f, h, i\rangle$. $a \overset{E\#Q}{\rightsquigarrow} i$ does not hold if $Q = \{g, h\}$ because all paths connecting a to i need to visit g or h. If $Q = \{d, e, g\}$, then $nodes(a \overset{E\#Q}{\rightsquigarrow} i) = \{a, b, c, f, h, i\}$ because of the two paths connecting a to i not visiting any of the nodes in Q.

2.2 Multisets

Multisets are used to represent the state of a Petri net and to describe event logs where the same trace may appear multiple times.

$\mathcal{B}(A)$ is the set of all multisets over some set A. For some multiset $b \in \mathcal{B}(A)$, $b(a)$ denotes the number of times element $a \in A$ appears in b. Some examples: $b_1 = [\,]$,

$b_2 = [x, x, y]$, $b_3 = [x, y, z]$, $b_4 = [x, x, y, x, y, z]$, $b_5 = [x^3, y^2, z]$ are multisets over $A = \{x, y, z\}$. b_1 is the empty multiset, b_2 and b_3 both consist of three elements, and $b_4 = b_5$, i.e., the ordering of elements is irrelevant and a more compact notation may be used for repeating elements.

The standard set operators can be extended to multisets, e.g., $x \in b_2$, $b_2 \uplus b_3 = b_4$, $b_5 \setminus b_2 = b_3$, $|b_5| = 6$, etc. $\{a \in b\}$ denotes the set with all elements a for which $b(a) \geq 1$. $[f(a) \mid a \in b]$ denotes the multiset where element $f(a)$ appears $\sum_{x \in b | f(x)=f(a)} b(x)$ times.

2.3 Petri Nets

Most of the results presented in the paper, can be adapted for various process modeling notations. However, we use Petri nets to formalize the main ideas and to prove their correctness.

Definition 6 (Petri Net). *A Petri net is tuple $PN = (P, T, F)$ with P the set of places, T the set of transitions, and $F \subseteq (P \times T) \cup (T \times P)$ the flow relation.*

Figure 2 shows an example Petri net $PN = (P, T, F)$ with $P = \{\mathit{start}, c1, \ldots, c5, \mathit{end}\}$, $T = \{a, b, \ldots, h\}$, and $F = \{(\mathit{start}, a), (a, c1), (a, c2), \ldots, (h, \mathit{end})\}$. The state of a Petri net, called *marking*, is a multiset of places indicating how many *tokens* each place contains. $[\mathit{start}]$ is the initial marking shown in Fig. 2. Another potential marking is $[c1^{10}, c2^5, c4^5]$. This is the state with ten tokens in $c1$, five tokens in $c2$, and five tokens in $c4$.

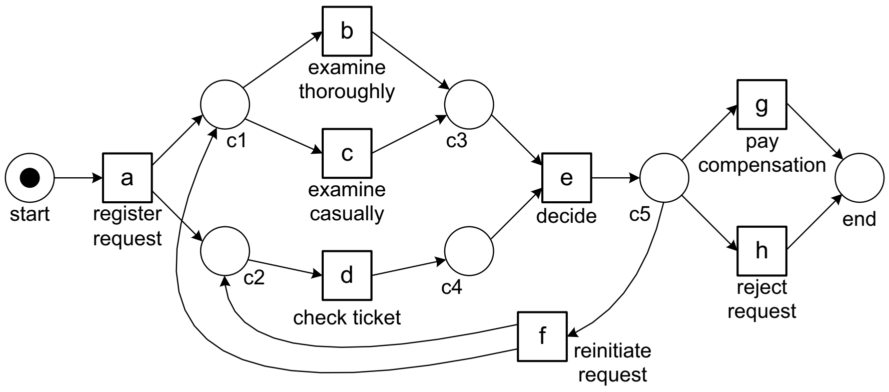


Fig. 2. A Petri net

Definition 7 (Marking). *Let $PN = (P, T, F)$ be Petri net. A marking M is a multiset of places, i.e., $M \in \mathcal{B}(P)$.*

Like for graphs we define the preset and postset of a node. For any $x \in P \cup T$, $\overset{PN}{\bullet} x = \{y \mid (y, x) \in F\}$ (input nodes) and $x \overset{PN}{\bullet} = \{y \mid (x, y) \in F\}$ (output nodes). We drop the superscript PN if it is clear from the context.

A transition $t \in T$ is *enabled* in marking M, denoted as $M[t\rangle$, if each of its input places $\bullet t$ contains at least one token. Consider the Petri net in Fig. 2 with $M = [c3, c4]$: $M[e\rangle$ because both input places are marked.

An enabled transition t may *fire*, i.e., one token is removed from each of the input places $\bullet t$ and one token is produced for each of the output places $t\bullet$. Formally: $M' = (M \setminus \bullet t) \uplus t\bullet$ is the marking resulting from firing enabled transition t in marking M. $M[t\rangle M'$ denotes that t is enabled in M and firing t results in marking M'. For example, $[start][a\rangle[c1, c2]$ and $[c3, c4][e\rangle[c5]$ for the net in Fig. 2.

Let $\sigma = \langle t_1, t_2, \ldots, t_n\rangle \in T^*$ be a sequence of transitions. $M[\sigma\rangle M'$ denotes that there is a set of markings $M_0, M_1, \ldots, M_n$ such that $M_0 = M$, $M_n = M'$, and $M_i[t_{i+1}\rangle M_{i+1}$ for $0 \leq i < n$. A marking M' is *reachable* from M if there exists a σ such that $M[\sigma\rangle M'$. For example, $[start][\sigma\rangle[end]$ for $\sigma = \langle a, b, d, e, g\rangle$.

Definition 8 (Labeled Petri Net). *A labeled Petri net $PN = (P, T, F, T_v)$ is a Petri net (P, T, F) with visible labels $T_v \subseteq T$. Let $\sigma_v = \langle t_1, t_2, \ldots, t_n\rangle \in T_v^*$ be a sequence of visible transitions. $M[\sigma_v \rhd M'$ if and only if there is a sequence $\sigma \in T^*$ such that $M[\sigma\rangle M'$ and the projection of σ on T_v yields σ_v (i.e., $\sigma_v = \sigma\upharpoonright_{T_v}$).*

If we assume $T_v = \{a, e, g, h\}$ for the Petri net in Fig. 2, then $[start][\sigma_v \rhd [end]$ for $\sigma_v = \langle a, e, e, e, e, g\rangle$ (i.e., b, c, d, and f are invisible).

In the context of process mining, we always consider processes that start in an initial state and end in a well-defined end state. For example, given the net in Fig. 2 we are interested in firing sequences starting in $M_i = [start]$ and ending in $M_o = [end]$. Therefore, we define the notion of a *system net*.

Definition 9 (System Net). *A system net is a triplet $SN = (PN, M_i, M_o)$ where $PN = (P, T, F, T_v)$ is a Petri net with visible labels T_v, $M_i \in \mathcal{B}(P)$ is the initial marking, and $M_o \in \mathcal{B}(P)$ is the final marking.*

Given a system net, $\tau(SN)$ is the set of all possible visible full traces, i.e., firing sequences starting in M_i and ending in M_o projected onto the set of visible transitions.

Definition 10 (Traces). *Let $SN = (PN, M_i, M_o)$ be a system net. $\tau(SN) = \{\sigma_v \mid M_i[\sigma_v \rhd M_o\}$ is the set of visible traces starting in M_i and ending in M_o.*

If we assume $T_v = \{a, e, f, g, h\}$ for the Petri net in Fig. 2, then $\tau(SN) = \{\langle a, e, g\rangle, \langle a, e, h\rangle, \langle a, e, f, e, g\rangle, \langle a, e, f, e, h\rangle, \ldots\}$.

2.4 WF-Net

The Petri net in Fig. 2 has a designated source place ($start$), a designated source place (end), and all nodes are on a path from $start$ to end. Such nets are called *WF-nets* [1, 4].

Definition 11 (WF-net). *$WF = (PN, in, T_i, out, T_o)$ is a workflow net (WF-net) if*

- *$PN = (P, T, F, T_v)$ is a labeled Petri net,*
- *$in \in P$ is a source place such that $\bullet in = \emptyset$ and $in\bullet = T_i$,*

- *$out \in P$ is a sink place such that $out\bullet = \emptyset$ and $\bullet out = T_o$,*
- *$T_i \subseteq T_v$ is the set of initial transitions and $\bullet T_i = \{in\}$,*
- *$T_o \subseteq T_v$ is the set of final transitions and $T_o\bullet = \{out\}$, and*
- *$nodes(in \overset{F}{\rightsquigarrow} out) = P \cup T$, i.e., all nodes are on some path from source place in to sink place out.*

WF-nets are often used in the context of business process modeling and process mining. Compared to the standard definition of WF-nets [1, 4] we added the requirement that the initial and final transitions need to be visible.

A WF-net $WF = (PN, in, T_i, out, T_o)$ defines the system $SN = (PN, M_i, M_o)$ with $M_i = [in]$ and $M_o = [out]$. Ideally WF-nets are also *sound*, i.e., free of deadlocks, livelocks, and other anomalies [1, 4]. Formally, this means that for any state reachable from M_i it is possible to reach M_o.

Process models discovered using existing process mining techniques may be unsound. Therefore, we cannot assume/require all WF-nets to be sound.

2.5 Event Log

As indicated earlier, *event logs* serve as the starting point for process mining. An event log is a multiset of *traces*. Each trace describes the life-cycle of a particular *case* (i.e., a *process instance*) in terms of the *activities* executed.

Definition 12 (Trace, Event Log). *Let A be a set of activities. A trace $\sigma \in A^*$ is a sequence of activities. $L \in \mathcal{B}(A^*)$ is an event log, i.e., a multiset of traces.*

An event log is a *multiset* of traces because there can be multiple cases having the same trace. In this simple definition of an event log, an event refers to just an *activity*. Often event logs may store additional information about events. For example, many process mining techniques use extra information such as the *resource* (i.e., person or device) executing or initiating the activity, the *timestamp* of the event, or *data elements* recorded with the event (e.g., the size of an order). In this paper, we abstract from such information. However, the results presented in this paper can easily be extended to event logs with more information.

An example log is $L_1 = [\langle a, e, g\rangle^{10}, \langle a, e, h\rangle^5, \langle a, e, f, e, g\rangle^3, \langle a, e, f, e, h\rangle^2]$. L_1 contains information about 20 cases, e.g., 10 cases followed trace $\langle a, e, g\rangle$. There are $10 \times 3 + 5 \times 3 + 3 \times 5 + 2 \times 5 = 70$ events in total.

Definition 13 (Projection). *Let A be a set and $X \subseteq A$ a subset. $\upharpoonright_X \in A^* \to X^*$ is a projection function and is defined recursively: (1) $\langle\ \rangle\upharpoonright_X = \langle\ \rangle$ and (2) for $\sigma \in A^*$ and $a \in A$:*

$$(\sigma; \langle a\rangle)\upharpoonright_X = \begin{cases} \sigma\upharpoonright_X & \text{if } a \notin X \\ \sigma\upharpoonright_X; \langle a\rangle & \text{if } a \in X \end{cases}$$

The projection function is generalized to event logs, i.e., for some event log $L \in \mathcal{B}(A^*)$ and set $X \subseteq A$: $L\upharpoonright_X = [\sigma\upharpoonright_X \mid \sigma \in L]$.

For the event log L_1: $L_1\upharpoonright_{\{a,g,h\}} = [\langle a, g\rangle^{13}, \langle a, h\rangle^7]$. Note that all e and f events have been removed.

3 Conformance Checking

Conformance checking techniques investigate how well an event log $L \in \mathcal{B}(A^*)$ and a system net $SN = (PN, M_i, M_o)$ fit together. Note that the process model SN may have been discovered through process mining or may have been made by hand. In any case, it is interesting to compare the observed example behavior in L and the potential behavior of SN.

Conformance checking can be done for various reasons. First of all, it may be used to audit processes to see whether reality conforms to some normative or descriptive model [5]. Deviations may point to fraud, inefficiencies, and poorly designed or outdated procedures. Second, conformance checking can be used to evaluate the results of a process discovery techniques. In fact, genetic process mining algorithms use conformance checking to select the candidate models used to create the next generation of models [29].

There are four quality dimensions for comparing model and log: (1) *fitness*, (2) *simplicity*, (3) *precision*, and (4) *generalization* [2]. A model with good *fitness* allows for most of 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. The *simplest* model that can explain the behavior seen in the log is the best model. This principle is known as Occam's Razor. Fitness and simplicity alone are not sufficient to judge the quality of a discovered process model. For example, it is very easy to construct an extremely simple Petri net ("flower model") that is able to replay all traces in an event log (but also any other event log referring to the same set of activities). Similarly, it is undesirable to have a model that only allows for the exact behavior seen in the event log. Remember that the log contains only example behavior and that many traces that are possible may not have been seen yet. 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). At the same time, the model should generalize and not restrict behavior to just the examples seen in the log. A model that does not *generalize* is "overfitting". 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 there is a high probability that the model is unable to explain the next batch of cases).

In the remainder, we will focus on fitness. However, the ideas are applicable to the other quality dimensions.

Definition 14 (Perfectly Fitting Log). *Let $L \in \mathcal{B}(A^*)$ be an event log and let $SN = (PN, M_i, M_o)$ be a system net. L is perfectly fitting SN if and only if $\{\sigma \in L\} \subseteq \tau(SN)$.*

Consider two event logs $L_1 = [\langle a, e, g\rangle^{10}, \langle a, e, h\rangle^5, \langle a, e, f, e, g\rangle^3, \langle a, e, f, e, h\rangle^2]$ and $L_2 = [\langle a, e, g\rangle^{10}, \langle a, e, h\rangle^5, \langle a, g\rangle^3, \langle a, a, g, e, h\rangle^2]$ and the system net SN of the WF-net depicted in Fig. 2 with $T_v = \{a, e, f, g, h\}$. Clearly, L_1 is perfectly fitting SN and L_2 is not. There are various ways to quantify fitness [2, 3, 8, 24, 29–32], typically on a scale from 0 to 1 where 1 means perfect fitness. To measure fitness, one needs to *align* traces in the event log to traces of the process model. Some example alignments for L_2 and SN:

$$\gamma_1 = \begin{array}{|c|c|c|} a & e & g \\ \hline a & e & g \end{array} \quad \gamma_2 = \begin{array}{|c|c|c|} a & e & h \\ \hline a & e & h \end{array} \quad \gamma_3 = \begin{array}{|c|c|c|} a & \gg & g \\ \hline a & e & g \end{array} \quad \gamma_4 = \begin{array}{|c|c|c|c|c|} a & a & g & e & h \\ \hline a & \gg & \gg & e & h \end{array} \quad \gamma_5 = \begin{array}{|c|c|c|c|c|c|} a & a & \gg & g & e & h \\ \hline a & \gg & e & g & \gg & \gg \end{array}$$

The top row of each alignment corresponds to "moves in the log" and the bottom row corresponds to "moves in the model". If a move in the log cannot be mimicked by a move in the model, then a "$\gg$" ("no move") appears in the bottom row. For example, in γ_4 the model is unable to do the second a move and is unable to do g before e. If a move in the model cannot be mimicked by a move in the log, then a "$\gg$" ("no move") appears in the top row. For example, in γ_3 the log did not do an e move whereas the model has to make this move to enable g and reach the end. Given a trace in the event log there may be many possible alignments. The goal is to find the alignment with the least number of $\gg$ elements, e.g., γ_4 is clearly better than γ_5. The number of $\gg$ elements can be used to quantify fitness. Moreover, once an optimal alignment has been established for every trace in the event log, these alignments can be used as a basis to quantify precision and generalization [3].

4 Distributed Conformance Checking

Conformance checking techniques can be time consuming as potentially many different traces need to be aligned with a model that may allow for an exponential (or even infinite) number of traces. Event logs may contain millions of events. Finding the best alignment may require solving many optimization problems [8] or repeated state-space explorations [32]. When using genetic process mining, one needs to check the fitness of every individual model in every generation [29]. As a result, thousands or even millions of conformance checks need to be done. For each conformance check, the whole event log needs to be traversed. Given these challenges, we are interested in reducing the time needed for conformance checking.

In this section, we show that it is possible to decompose and distribute conformance checking problems using the notion of *passages* defined in Section 2.1. In order to do this we focus on the visible transitions and create the so-called *skeleton* of the process model.

Definition 15 (Skeleton). *Let $PN = (P, T, F, T_v)$ be a labeled Petri net. The* skeleton *of PN is the graph $skel(PN) = (N, E)$ with $N = T_v$ and $E = \{(x, y) \in T_v \times T_v \mid x \overset{F\#T_v}{\rightsquigarrow} y\}$.*

Figure 3 shows the skeleton of the WF-net in Fig. 2 assuming that $T_v = \{a, e, f, g, h\}$. The resulting graph has two minimal minimal passages.

Note that only the visible transitions T_v appear in the skeleton. For example, if we assume that $T_v = \{a, g, h\}$ in Fig. 2, then the skeleton is $(\{a, g, h\}, \{(a, g), (a, h)\})$ and there is only one passage $(\{a\}, \{g, h\})$.

If there are multiple minimal passages in the skeleton, we can decompose conformance checking problems into smaller problems *by partitioning the Petri net into net*

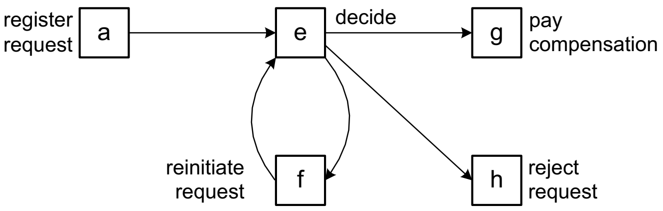


Fig. 3. The skeleton of the labeled Petri net in Fig. 2 (assuming that $T_v = \{a, e, f, g, h\}$). There are two minimal minimal passages: $(\{a, f\}, \{e\})$ and $(\{e\}, \{f, g, h\})$.

fragments and the event log into sublogs. Each passage (X, Y) defines one net fragment $PN^{(X,Y)}$ and one sublog $L\restriction_{X \cup Y}$. We will show that conformance can be checked per passage.

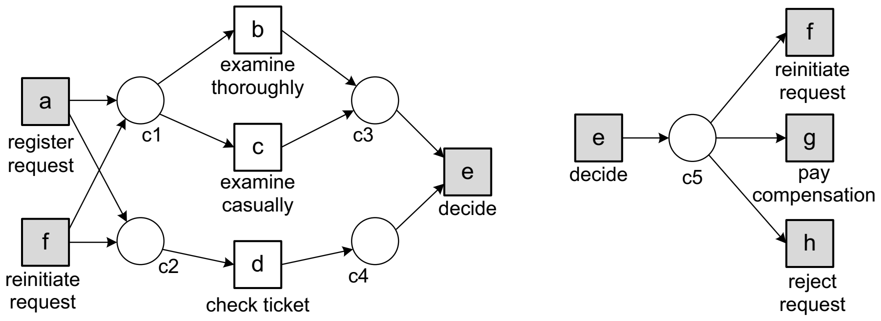


Fig. 4. Two net fragments corresponding to the two passages of the skeleton in Fig. 3: $PN_1 = PN^{(\{a,f\},\{e\})}$ (left) and $PN_2 = PN^{(\{e\},\{f,g,h\})}$ (right). The visible transitions $T_v = \{a, e, f, g, h\}$ that form the boundaries of the fragments are highlighted.

Consider event log $L = [\langle a, e, g\rangle^{10}, \langle a, e, h\rangle^5, \langle a, e, f, e, g\rangle^3, \langle a, e, f, e, h\rangle^2]$, the WF-net PN shown in Fig. 2 with $T_v = \{a, e, f, g, h\}$, and the skeleton shown in Fig. 3. There are two passages: $P_1 = (\{a, f\}, \{e\})$ and $P_2 = (\{e\}, \{f, g, h\})$. Based on this we define two net fragments PN_1 and PN_2 as shown in Fig. 4. Moreover, we define two sublogs: $L_1 = [\langle a, e\rangle^{15}, \langle a, e, f, e\rangle^5]$ and $L_2 = [\langle e, g\rangle^{10}, \langle e, h\rangle^5, \langle e, f, e, g\rangle^3, \langle e, f, e, h\rangle^2]$. To check the conformance of the overall event log on the overall model, we check the conformance of L_1 on PN_1 and L_2 on PN_2. Since L_1 is perfectly fitting PN_1 and L_2 is perfectly fitting PN_2, we can conclude that L is perfectly fitting PN. This illustrates that conformance checking can be decomposed.

In order to prove this, we first define the notion of a net fragment.

Definition 16 (Net Fragment). *Let $PN = (P, T, F, T_v)$ be a labeled Petri net. For any two sets of transitions $X, Y \subseteq T_v$, we define the net fragment $PN^{(X,Y)} = (P', T', F', T'_v)$ with:*

- $Z = nodes(X \overset{F\#T_v}{\rightsquigarrow} Y) \setminus (X \cup Y)$ *are the internal nodes of the fragment,*
- $P' = P \cap Z$,
- $T' = (T \cap Z) \cup X \cup Y$,
- $F' = F \cap ((P' \times T') \cup (T' \times P'))$, *and*
- $T'_v = X \cup Y$.

Note that $PN_1 = PN^{(\{a,f\},\{e\})}$ in Fig. 4 has $Z = \{b, c, d, c1, c2, c3, c4\}$ as internal nodes.

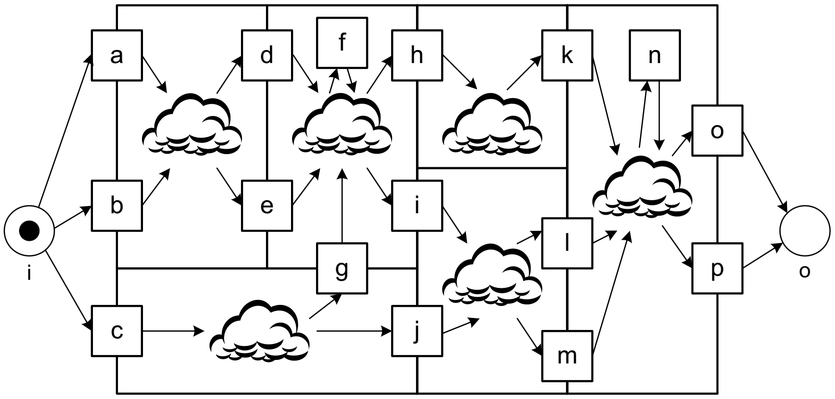


Fig. 5. WF-net WF is decomposed in subnets $PN^{(X,Y)}$. The "clouds" model the internal structure of these subnets (places but possibly also hidden transitions). Due to the decomposition based on passages, one cloud can only influence another cloud through the visible interface transitions X and Y. Since the visible interface transitions are "controlled" by the event log, it is possible to check fitness locally per subnet.

Now we can prove the main result of this paper. Figure 5 illustrates our decomposition approach. A larger model can be decomposed into net fragments corresponding to minimal passages. The event log can be decomposed in a similar manner and conformance checking can be done per passage.

Theorem 1 (Main Theorem). *Let* $L \in \mathcal{B}(A^*)$ *be an event log and let* $WF = (PN, in, T_i, out, T_o)$ *be a WF-net with* $PN = (P, T, F, T_v)$.

L is perfectly fitting system net $SN = (PN, [in], [out])$ *if and only if*

- *for any* $\langle a_1, a_2, \dots a_k \rangle \in L$: $a_1 \in T_i$ *and* $a_k \in T_o$, *and*
- *for any* $(X, Y) \in pas_{min}(skel(PN))$: $L\upharpoonright_{X \cup Y}$ *is perfectly fitting* $SN^{(X,Y)} = (PN^{(X,Y)}, [\,], [\,])$.

Proof. (⇒) Let $\sigma_v = \langle a_1, a_2, \dots a_k \rangle \in L$ such that there is a $\sigma \in T^*$ with $[in][\sigma\rangle[out]$ and $\sigma\upharpoonright_{T_v} = \sigma_v$ (i.e., σ_v fits into the overall WF-net). We need to prove the two properties listed above:

- $a_1 \in T_i$ and $a_k \in T_o$ because only transitions in T_i are enabled in the initial marking and only transitions in T_o can produce tokens for out. Moreover, when σ puts a token in place out all other places should be empty; otherwise σ cannot result in $[out]$ (property of WF-nets). Note that $T_i \subseteq T_v$ and $T_o \subseteq T_v$, so the first and last transition need to be visible.
- For any $(X, Y) \in pas_{min}(skel(PN))$: we define $PN^{(X,Y)} = (P', T', F', T'_v)$ and $\sigma' = \sigma \upharpoonright_{T'}$. We need to prove that $[\,][\sigma'\rangle[\,]$ in $PN^{(X,Y)}$. This follows trivially because $SN^{(X,Y)}$ can mimic any move of SN with respect to transitions T'.

($\Leftarrow$) Let $\sigma_v = \langle a_1, a_2, \ldots a_k\rangle \in L$ such that $a_1 \in T_i$, $a_k \in T_o$, and assume that for any $(X, Y) \in pas_{min}(skel(PN))$ there is a sequence $\sigma_{(X,Y)}$ such that $[\,][\sigma_{(X,Y)}\rangle[\,]$ in $PN^{(X,Y)} = (P', T', F', T'_v)$ with $\sigma_{(X,Y)} \upharpoonright_{X \cup Y} = \sigma_v \upharpoonright_{X \cup Y}$. We need to prove that there is a $\sigma \in T^*$ such that $[in][\sigma\rangle[out]$ in PN with $\sigma \upharpoonright_{T_v} = \sigma_v$. The different $\sigma_{(X,Y)}$ sequences can be stitched together into an overall σ because the different subnets only interface via visible transitions. Transitions in one subnet can only influence other subnets through visible transitions and these can only move synchronously as defined by $\sigma_v \in L$. $\square$

Although the theorem only addresses the notion of perfect fitness, other conformance notions can be decomposed in a similar manner. Metrics can be computed per passage and then aggregated into an overall metric.

Assuming a process model with many passages, the time needed for conformance checking can be reduced significantly. There are two reasons for this. First of all, as Theorem 1 shows, larger problems can be decomposed into a set of independent smaller problems. Therefore, conformance checking can be distributed over multiple computers. Second, due to the exponential nature of most conformance checking techniques, the time needed to solve "many smaller problems" is less than the time needed to solve "one big problem". Existing approaches use state-space analysis (e.g., in [32] the shortest path enabling a transition is computed) or optimization over all possible alignments (e.g., in [8] the A^* algorithm is used to find the best alignment). These techniques do *not* scale linearly in the number of activities. *Therefore, decomposition is useful even if the checks per passage are done on a single computer.*

5 Process Discovery: Divide and Conquer

As explained before, conformance checking and process discovery are closely related. Therefore, we can use the approach used in Theorem 1 for process discovery provided that some coarse *causal structure* (comparable to the skeleton in Section 4) is known. Based on the passages in the causal structure, multiple smaller discovery problems are formulated. This result in one net fragment per passage. These fragments can be folded into an overall model.

More concretely, we propose the following discovery approach:

1. Input is an event log $L_{raw} \in \mathcal{B}(A^*_{raw})$ over a set of activities A_{raw}.
2. Extend each trace in the event log with an artificial start event $\top$ and an artificial end event $\bot$ ($\{\top, \bot\} \cap A_{raw} = \emptyset$). $L_{ext} = [\langle\top\rangle; \sigma; \langle\bot\rangle \mid \sigma \in L_{raw}]$ is the resulting log over $A_{ext} = \{\top, \bot\} \cup A_{raw}$.

3. Discover the causal structure, i.e., we assume that there is an algorithm γ_c such that $\gamma_c(L_{ext}) = (A, C)$ with $\{\top, \bot\} \subseteq A \subseteq A_{ext}$ and $C \subseteq A \times A$. The causal structure may be inspected and modified by a domain expert.
4. Filter the event log using the selected set of activities A: $L = L_{ext}\upharpoonright_A$.
5. Compute the set of passages on the graph $G = (A, C)$: $PS = pas_{min}(G) = \{(X_1, Y_1), (X_2, Y_2), \ldots, (X_k, Y_k)\}$. We assume that there is an algorithm γ_p, such that $\gamma_p(L\upharpoonright_{X_i \cup Y_i}, X_i, Y_i) = PN_i = (P_i, T_i, F_i, X_i \cup Y_i)$ returns a Petri net with visible transitions $X_i \cup Y_i$. The discovered Petri nets only overlap with respect to visible transitions, i.e., for $1 \leq i < j \leq k$: $((P_i \cup T_i) \setminus (X_i \cup Y_i)) \cap ((P_j \cup T_j) \setminus (X_j \cup Y_j)) = \emptyset$. Moreover, each PN_i should respect the causal structure, i.e., visible transition $x \in X_i$ is connected to visible transition $y \in Y_i$ in PN_i if and only if $(x, y) \in C$.
6. Merge the individual subsets into one overall system net $SN = (PN, M_i, M_o)$ with $PN = (P, T, F, T_v)$ such that:
   - $P = \{in, out\} \cup \bigcup_{1 \leq i \leq k} P_i$,
   - $T = \bigcup_{1 \leq i \leq k} T_i$,
   - $F = \{(in, \top), (\bot, out)\} \cup (\bigcup_{1 \leq i \leq k} F_i)$,
   - $T_v = A$,
   - $M_i = [in]$, and
   - $M_o = [out]$.

The discovery process is *parameterized* by γ_c (the algorithm to find causal structure) and γ_p (the algorithm to find a local, transition bordered process model). Any combination of γ_c and γ_p can be used as the two main steps are decoupled by the causal structure. γ_c can also be used to filter out infrequent activities, noise, etc. Moreover, the user is able to edit the causal structure using domain knowledge or particular preferences. Experience shows that user feedback is vital to balance between overfitting and underfitting.

The log is extended by adding an artificial start event $\top$ and an artificial end event $\bot$ to every trace, This is just a technicality to ensure that there is a clearly defined start and end. Note that passages can be activated multiple times, e.g., in case of loops. Therefore, we add transitions $\top$ and $\bot$ and places in and out. If there is a unique start (end) event, then there is no need to add transition $\top$ ($\bot$). Ideally, the causal structure created in Step 3 has one source node $\top$, one sink node $\bot$, and all other nodes are on a path from $\top$ to $\bot$ (like in a WF-net).

To illustrate the divide and conquer approach based on passages, consider the event log $L_{raw} = [\langle a, b, c, d\rangle^{40}, \langle b, a, c, d\rangle^{35}, \langle a, b, c, e\rangle^{30}, \langle b, a, c, e\rangle^{25}, \langle a, b, x, d\rangle^{1}, \langle a, b, e\rangle^{1}]$. The log describes 132 cases. We first add the artificial start and events (Step 2): $L_{ext} = [\langle \top, a, b, c, d, \bot\rangle^{40}, \langle \top, b, a, c, d, \bot\rangle^{35}, \langle \top, a, b, c, e, \bot\rangle^{30}, \langle \top, b, a, c, e, \bot\rangle^{25}, \langle \top, a, b, x, d, \bot\rangle^{1}, \langle \top, a, b, e, \bot\rangle^{1}]$. Then we compute the causal structure using γ_c (Step 3). Assume that the causal structure shown in Fig. 6 is computed. Since x occurs only once whereas the other activities occur more than 50 times, x is excluded. The same holds for the dependency between b and e. L is the log where x is removed (Step 4).

The causal structure has four minimal passages: $P_1 = (\{\top\}, \{a, b\})$, $P_2 = (\{a, b\}, \{c\})$, $P_3 = (\{c\}, \{d, e\})$, and $P_4 = (\{d, e\}, \{\bot\})$. Based on these passages we create four corresponding sublogs: $L_1 = [\langle \top, a, b\rangle^{72}, \langle \top, b, a\rangle^{60}]$, $L_2 = [\langle a, b, c\rangle^{70}, \langle b, a, c\rangle^{60}$,

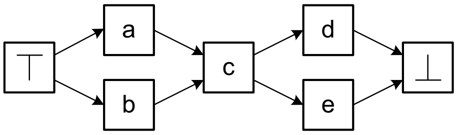


Fig. 6. Causal structure $\gamma_c(L_{ext})$ discovered for the extended event log having four minimal passages

$\langle a, b\rangle^2]$, $L_3 = [\langle c, d\rangle^{75}, \langle c, e\rangle^{55}, \langle d\rangle^1, \langle e\rangle^1]$, and $L_4 = [\langle d, \perp\rangle^{76}, \langle e, \perp\rangle^{56}]$. One transition-bordered Petri net is discovered per sublog using γ_p (Step 5). Figure 7 shows the resulting net fragments. Note that infrequent behavior has been discarded, i.e., trace $\langle a, b\rangle$ in L_2 is not possible in PN_2, and traces $\langle d\rangle$ and $\langle e\rangle$ in L_3 are not possible in PN_3. What behavior is included and what not depends on γ_p.

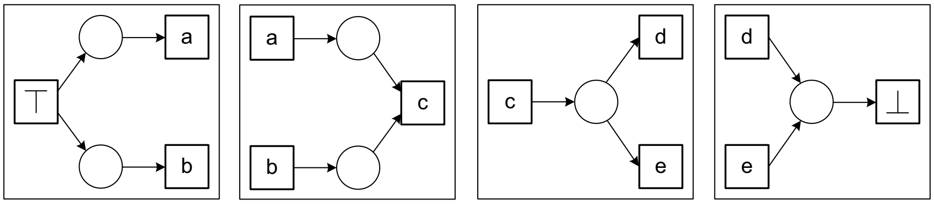


Fig. 7. The Petri net fragments discovered for the four passages: PN_1, PN_1, PN_3, and PN_4

In the last step of the approach, the four net fragments of Fig. 7 are merged into the overall model shown in Figure 8 (Step 6). Note that this model is indeed able to replay all frequent behavior. Two of the 132 cases cannot be replayed because they were treated as noise by γ_c and γ_p.

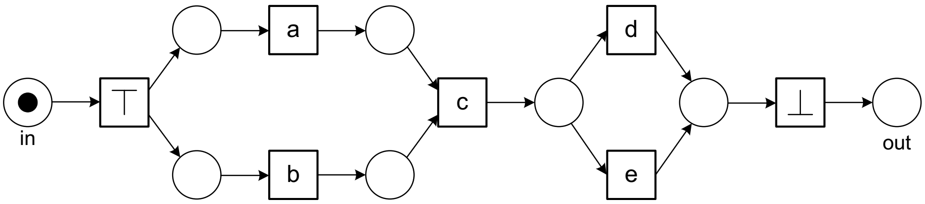


Fig. 8. The WF-net obtained by merging the individual subsets

The small example shows that we can use a divide and conquer approach when discovering process models. We deliberately did not select concrete algorithms for γ_c and γ_p. The approach is generic and can be combined with existing process discovery techniques [2, 6, 7, 11, 13, 18, 19, 21, 24, 29, 33, 35, 36]. Moreover, the user can modify the causal structure (i.e., the result of γ_c) to guide the discovery process.

By decomposing the overall discovery problem into a collection of smaller discovery problems, it is possible to do a more refined analysis and achieve significant speed-ups. The discovery algorithm γ_p is applied to an event log consisting of just the activities involved in the passage under investigation. Hence, process discovery tasks can be distributed over a network of computers (assuming there are multiple passages). Moreover, most discovery algorithms are exponential in the number of activities. *Therefore, the sequential discovery of all individual passages on one computer is often still faster than solving one big discovery problem.* If there are more passages than computers, one can merge minimal passages into aggregate passages and use these for discovery and conformance checking (one passage per computer). However, in most situations, it will be more efficient to analyze the minimal passages sequentially.

6 Related Work

For an introduction to process mining we refer to [2]. For an overview of best practices and challenges, we refer to the Process Mining Manifesto [26]. The goal of this paper is to decompose challenging process discovery and conformance checking problems into smaller problems. Therefore, we first review some of the techniques available for process discovery and conformance checking.

Process discovery, i.e., discovering a process model from a multiset of example traces, is a very challenging problem and various discovery techniques have been proposed [6, 7, 11, 13, 18, 19, 21, 24, 29, 33, 35, 36]. Many of these techniques use Petri nets during the discovery process and/or to represent the discovered model. It is impossible to provide an complete overview of all techniques here. Very different approaches are used, e.g., heuristics [21, 35], inductive logic programming [24], state-based regions [6, 19, 33], language-based regions [13, 36], and genetic algorithms [29]. Classical synthesis techniques based on regions [23] cannot be applied directly because the event log contains only example behavior. For state-based regions one first needs to create an automaton as described in [6]. Moreover, when constructing the regions, one should avoid overfitting. Language-based regions seem good candidates for discovering transition-bordered Petri nets for passages [13, 36]. Unfortunately, these techniques still have problems dealing with infrequent/incomplete behavior.

As described in [2], there are four competing quality criteria when comparing modeled behavior and recorded behavior: fitness, simplicity, precision, and generalization. In this paper, we focused on fitness, but also precision and generalization can also be investigated per passage. Various conformance checking techniques have been proposed in recent years [3, 8–10, 16, 22, 24, 30–32, 34]. Conformance checking can be used to evaluate the quality of discovered processes but can also be used for auditing purposes [5]. Most of the techniques mentioned can be applied to passages. The most challenging part is to aggregate the metrics per passage into metrics for the overall model and log. We consider the approach described in [8] to be most promising as it constructs an optimal alignment given an arbitrary cost function. This alignment can be used for computing precision and generalization [3, 31]. However, the approach can be rather time consuming. Therefore, the efficiency gains can be considerable for larger processes with many activities and passages.

Little work has been done on the decomposition and distribution of process mining problems. In [15] an approach is described to distribute genetic process mining over multiple computers. In this approach candidate models are distributed and in a similar fashion also the log can be distributed. However, individual models are not partitioned over multiple nodes. Therefore, the approach in this paper is complementary. Moreover, unlike [15], the decomposition approach based on passages is not restricted to genetic process mining.

Most related are the divide-and-conquer techniques presented in [20]. In [20] it is shown that region-based synthesis can be done at the level of synchronized State Machine Components (SMCs). Also a heuristic is given to partition the causal dependency graph into overlapping sets of events that are used to construct sets of SMCs. Passages provide a different (more local) partitioning of the problem and, unlike [20] which focuses on state-based region mining, we decouple the decomposition approach from the actual conformance checking and process discovery approaches.

Several approaches have been proposed to distribute the verification of Petri net properties, e.g., by partitioning the state space using a hash function [14] or by modularizing the state space using localized strongly connected components [27]. These techniques do not consider event logs and cannot be applied to process mining.

Most data mining techniques can be distributed [17], e.g., distributed classification, distributed clustering, and distributed association rule mining [12]. These techniques often partition the input data and cannot be used for the discovery of Petri nets.

7 Conclusion

Computationally challenging process mining problems can be decomposed in smaller problems using the new notion of *passages*. This paper shows that the fitness of the overall model can be analyzed per passage. The approach is independent of the particular conformance checking technique used. Moreover, the same idea can be applied to other conformance notions. The paper also presents a discovery approach where the discovery problem can be decomposed after determining the causal structure. The refined behavior can be discovered per passage and, subsequently, the discovered net fragments can be merged into an overall process model. Conformance checking and process discovery can be done much more efficiently using such decompositions. Moreover, the approach can be distributed over a network of computers.

This paper presents the idea of passages and provides a formal correctness proof showing that a log is perfectly fitting the overall model if and only if the property holds per passage. Future work will focus on large scale experiments demonstrating the performance gains on a variety of process mining problems. We anticipate that the actual speedup heavily depends on the number of passages. Therefore, it is important investigate this using real-life logs and models.

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., 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)
4. van der Aalst, W.M.P., van Hee, K.M., ter Hofstede, A.H.M., Sidorova, N., Verbeek, H.M.W., Voorhoeve, M., Wynn, M.T.: Soundness of Workflow Nets: Classification, Decidability, and Analysis. Formal Aspects of Computing 23(3), 333–363 (2011)
5. van der Aalst, W.M.P., van Hee, K.M., van der Werf, J.M., Verdonk, M.: Auditing 2.0: Using Process Mining to Support Tomorrow's Auditor. IEEE Computer 43(3), 90–93 (2010)
6. 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)
7. van der Aalst, W.M.P., Weijters, A.J.M.M., Maruster, L.: Workflow Mining: Discovering Process Models from Event Logs. IEEE Transactions on Knowledge and Data Engineering 16(9), 1128–1142 (2004)
8. Adriansyah, A., van Dongen, B., van der Aalst, W.M.P.: Conformance Checking using Cost-Based Fitness Analysis. In: Chi, C.H., Johnson, P. (eds.) IEEE International Enterprise Computing Conference (EDOC 2011), pp. 55–64. IEEE Computer Society (2011)
9. Adriansyah, A., van Dongen, B.F., van der Aalst, W.M.P.: Towards Robust Conformance Checking. In: zur Muehlen, M., Su, J. (eds.) BPM 2010 Workshops. LNBIP, vol. 66, pp. 122–133. Springer, Heidelberg (2011)
10. Adriansyah, A., Sidorova, N., van Dongen, B.F.: Cost-based Fitness in Conformance Checking. In: International Conference on Application of Concurrency to System Design (ACSD 2011), pp. 57–66. IEEE Computer Society (2011)
11. 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)
12. Agrawal, R., Shafer, J.C.: Parallel Mining of Association Rules. IEEE Transactions on Knowledge and Data Engineering 8(6), 962–969 (1996)
13. 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)
14. Boukala, M.C., Petrucci, L.: Towards Distributed Verification of Petri Nets properties. In: Proceedings of the International Workshop on Verification and Evaluation of Computer and Communication Systems (VECOS 2007), pp. 15–26. British Computer Society (2007)
15. 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)
16. Calders, T., Guenther, C., Pechenizkiy, M., Rozinat, A.: Using Minimum Description Length for Process Mining. In: ACM Symposium on Applied Computing (SAC 2009), pp. 1451–1455. ACM Press (2009)
17. Cannataro, M., Congiusta, A., Pugliese, A., Talia, D., Trunfio, P.: Distributed Data Mining on Grids: Services, Tools, and Applications. IEEE Transactions on Systems, Man, and Cybernetics, Part B 34(6), 2451–2465 (2004)
18. Carmona, J., Cortadella, J.: Process Mining Meets Abstract Interpretation. In: Balcázar, J.L., Bonchi, F., Gionis, A., Sebag, M. (eds.) ECML PKDD 2010. LNCS, vol. 6321, pp. 184–199. Springer, Heidelberg (2010)
19. Carmona, J.A., 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)

20. Carmona, J., Cortadella, J., Kishinevsky, M.: Divide-and-Conquer Strategies for Process Mining. In: Dayal, U., Eder, J., Koehler, J., Reijers, H.A. (eds.) BPM 2009. LNCS, vol. 5701, pp. 327–343. Springer, Heidelberg (2009)
21. 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)
22. Cook, J.E., Wolf, A.L.: Software Process Validation: Quantitatively Measuring the Correspondence of a Process to a Model. ACM Transactions on Software Engineering and Methodology 8(2), 147–176 (1999)
23. Darondeau, P.: Unbounded Petri Net Synthesis. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) ACPN 2004. LNCS, vol. 3098, pp. 413–438. Springer, Heidelberg (2004)
24. Goedertier, S., Martens, D., Vanthienen, J., Baesens, B.: Robust Process Discovery with Artificial Negative Events. Journal of Machine Learning Research 10, 1305–1340 (2009)
25. Hilbert, M., Lopez, P.: The World's Technological Capacity to Store, Communicate, and Compute Information. Science 332(6025), 60–65 (2011)
26. IEEE Task Force on Process Mining. Process Mining Manifesto. In: Business Process Management Workshops. LNBIP, vol. 99, pp. 169–194. Springer, Berlin (2012)
27. Lakos, C., Petrucci, L.: Modular Analysis of Systems Composed of Semiautonomous Subsystems. In: Application of Concurrency to System Design (ACSD 2004), pp. 185–194. IEEE Computer Society (2004)
28. 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)
29. Alves de Medeiros, A.K., Weijters, A.J.M.M., van der Aalst, W.M.P.: Genetic Process Mining: An Experimental Evaluation. Data Mining and Knowledge Discovery 14(2), 245–304 (2007)
30. Muñoz-Gama, J., Carmona, J.: A Fresh Look at Precision in Process Conformance. In: Hull, R., Mendling, J., Tai, S. (eds.) BPM 2010. LNCS, vol. 6336, pp. 211–226. Springer, Heidelberg (2010)
31. Munoz-Gama, J., Carmona, J.: Enhancing Precision in Process Conformance: Stability, Confidence and Severity. In: IEEE Symposium on Computational Intelligence and Data Mining (CIDM 2011), Paris, France. IEEE (April 2011)
32. Rozinat, A., van der Aalst, W.M.P.: Conformance Checking of Processes Based on Monitoring Real Behavior. Information Systems 33(1), 64–95 (2008)
33. 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)
34. De Weerdt, J., De Backer, M., Vanthienen, J., Baesens, B.: A Robust F-measure for Evaluating Discovered Process Models. In: IEEE Symposium on Computational Intelligence and Data Mining (CIDM 2011), Paris, France, pp. 148–155. IEEE (April 2011)
35. Weijters, A., 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)
36. 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)

Complexity of the Soundness Problem of Bounded Workflow Nets

Guan Jun Liu[1], Jun Sun[1], Yang Liu[2], and Jin Song Dong[3]

[1] ISTD, Singapore University of Technology and Design, Singapore 138682
{guanjun_liu,sunjun}@sutd.edu.sg
[2] Temasek Lab, National University of Singapore, Singapore 117417
tslliuya@nus.edu.sg
[3] School of Computing, National University of Singapore, Singapore 117417
dongjs@comp.nus.edu.sg

Abstract. Classical workflow nets (WF-nets) are an important class of Petri nets that are widely used to model and analyze workflow systems. Soundness is a crucial property that guarantees these systems are deadlock-free and bounded. Aalst *et al.* proved that the soundness problem is decidable, and proposed (but not proved) that the soundness problem is EXPSPACE-hard. In this paper, we show that the satisfiability problem of Boolean expression is polynomial time reducible to the liveness problem of bounded WF-nets, and soundness and liveness are equivalent for bounded WF-nets. As a result, the soundness problem of bounded WF-nets is co-NP-hard.

Workflow nets with reset arcs (reWF-nets) are an extension to WF-nets, which enhance the expressiveness of WF-nets. Aalst *et al.* proved that the soundness problem of reWF-nets is undecidable. In this paper, we show that for bounded reWF-nets, the soundness problem is decidable and equivalent to the liveness problem. Furthermore, a bounded reWF-net can be constructed in polynomial time for every linear bounded automaton (LBA) with an input string, and we prove that the LBA accepts the input string if and only if the constructed reWF-net is live. As a result, the soundness problem of bounded reWF-nets is PSPACE-hard.

Keywords: Petri nets, workflow nets, workflow nets with reset arcs, soundness, co-NP-hardness, PSPACE-hardness.

1 Introduction

In the recent decade, workflow nets (WF-nets) have been widely applied to (inter-organizational) workflow management systems and business process management systems to model and analyze their operational processes [1]-[4], [11]-[14], [16]. WF-nets can well characterize their system features such as concurrency, choices, and synchronous/asynchronous communication. To enhance the expressive power, some extensions to WF-nets, such as workflow nets with reset or inhibitor arcs (reWF-nets and inWF-nets, respectively) [17,18], are proposed which can express priority, preemption, or cancelation.

Soundness [1]-[4] is an important property of these systems which, informally speaking, reflects whether the designed systems are correct. For instance, the soundness of

WF-nets guarantees that the designed systems are deadlock-free and bounded. Aalst *et al.* [4] defined eight notions of soundness. This work considers the classical soundness [4] that means any run of the designed systems is always finished correctly and each action has a right/chance to be executed.

It has been proven that the soundness of WF-nets is decidable [3,11]. Aalst proposed (but not proved) in [3] that the soundness problem is EXPSPACE-hard, based on the work in [5], which shows that the problems of reachability, liveness, and deadlock of Petri nets are all EXPSPACE-hard. In addition, Aalst proved in [3] that the soundness problem of a workflow net can be decided in polynomial time if the workflow net is a free-choice one [6].

Generally, enhancing the models' expressive power increases the complexity of deciding their properties. Aalst *et al.* [4] proved that the soundness problems of reWF- and inWF-nets are both undecidable.

Our contribution. In this paper, we show that the satisfiability problem of Boolean expression (SAT problem) is polynomial time reducible to the liveness problem of bounded WF-nets, and the soundness and liveness are equivalent for bounded WF-nets, thereby proving that the soundness problem of bounded WF-nets is co-NP-hard. Based on the Linear Bounded Automaton Acceptance problem (LBA Acceptance problem), we show that the soundness problem of bounded reWF-nets is decidable but PSPACE-hard. To the best of our knowledge, it is the first time to propose and prove these conclusions for WF- and reWF-nets.

Organization. The remainder of the paper is organized as follows. Section 2 reviews the definitions of Petri nets, WF- and reWF-nets, and the SAT and LBA Acceptance problems. Section 3 proves the co-NP-hardness of the soundness of bounded WF-nets and Section 4 proves the PSPACE-hardness of the soundness of bounded reWF-nets. Section 5 concludes this paper.

2 Preliminary

In this section, we review the definitions of Petri nets, WF-nets, reWF-nets, SAT problem, and LBA Acceptance problem. For more details, please refer to [4] and [9].

2.1 Petri Nets

Let $\mathbb{N} = \{0,\ 1,\ 2, \cdots\}$ be the set of nonnegative integers, Given $m \in \mathbb{N}$ and $m > 0$, let $\mathbb{N}_m = \{1,\ 2, \cdots, m\}$ be the set of integers from 1 to m.

Definition 1. *A net is a 3-tuple $N = (P,\ T,\ F)$ where P is a set of places, T is a set of transitions, $F \subseteq (P \times T) \cup (T \times P)$ is a set of arcs, $P \cup T \neq \varnothing$, and $P \cap T = \varnothing$.*

A transition t is called an *input transition* of a place p and p is called an *output place* of a transition t if $(t,\ p) \in F$. *Input place* and *output transition* can be defined similarly. Given a net $N = (P,\ T,\ F)$ and a node $x \in P \cup T$, ${}^\bullet x = \{y \in P \cup T \mid (y,\ x) \in F\}$ and $x^\bullet = \{y \in P \cup T \mid (x,\ y) \in F\}$ are called the *pre-set* and *post-set* of x, respectively.

A *marking* of $N = (P, T, F)$ is a mapping M: $P \to \mathbb{N}$. $p \in P$ is *marked* at M if $M(p) > 0$. A marking may be viewed as a $|P|$-dimensional non-negative integer vector in which every element represents the number of tokens in corresponding place at this marking, e.g., $M = (1, 0, 6, 0)$ over $P = \{p_1, p_2, p_3, p_4\}$ represents that at M, p_1, p_2, p_3, and p_4 have 1, 0, 6, and 0 tokens, respectively. Notice, we assume a total order on the set of places P so that the i-th entry in the vector corresponds to the i-th place in the ordered set. When the number of places is very large and the distribution of tokens is sparse, the above two kinds of presentation of a marking are relatively complex. For convenience, M is denoted as $M = \sum_{p \in P} M(p) \cdot p$ in this paper. For the above example, it is written as $M = p_1 + 6p_3$.

If $\forall p \in {}^{\bullet}t$: $M(p) > 0$, then t is said to be *enabled* at M, which is denoted as $M[t\rangle$. Firing an enabled transition t produces a new marking M', which is denoted as $M[t\rangle M'$, such that $M'(p) = M(p) - 1$ if $p \in {}^{\bullet}t \setminus t^{\bullet}$; $M'(p) = M(p) + 1$ if $p \in t^{\bullet} \setminus {}^{\bullet}t$; and $M'(p) = M(p)$ otherwise.

A marking M_k is said to be reachable from a marking M if there exists a firing sequence $\sigma = t_1 t_2 \cdots t_k$ such that $M[t_1\rangle M_1[t_2\rangle \cdots \rangle M_{k-1}[t_k\rangle M_k$. $M[\sigma\rangle M_k$ represents that M reaches M_k after firing sequence σ. The set of all markings reachable from M in a net N is denoted as $R(N, M)$.

A net N with an *initial marking* M_0 is called a *Petri net*, and denoted as (N, M_0).

Definition 2. *Given a Petri net $(N, M_0) = (P, T, F, M_0)$, $t \in T$ is called live if $\forall M \in R(N, M_0)$, $\exists M' \in R(N, M)$: $M'[t\rangle$. A Petri net (N, M_0) is called live if every transition is live. It is called bounded if $\forall p \in P$, $\exists k \in \mathbb{N}$, $\forall M \in R(N, M_0)$: $M(p) \le k$.*

2.2 WF-Nets

WF-nets are an important subclass of Petri nets and widely studied and applied in academic and industrial systems. Each WF-net has a source place representing the beginning of a task and a sink place representing the ending of the task.

Definition 3. *A net $N = (P, T, F)$ is a WF-net if*

1. *N has two special places i and o where $i \in P$ is called source place such that ${}^{\bullet}i = \emptyset$ and $o \in P$ is called sink place such that $o^{\bullet} = \emptyset$; and*
2. *$N^E = (P, T \cup \{b\}, F \cup \{(b, i), (o, b)\})$ is strongly connected.*

For instance, Fig. 1(a) is a WF-net in which i and o are its source and sink places, respectively. This WF-net may be seen as a composition of three subsystems. The left and right subsystems produce parts and the middle assembles them.

Definition 4. *Let $N = (P, T, F)$ be a WF-net, $M_0 = i$, and $M_d = o$. N is sound if the following requirements hold:*

1. *$\forall M \in R(N, M_0)$: $M_d \in R(N, M)$; and*
2. *$\forall t \in T$, $\exists M \in R(N, M_0)$: $M[t\rangle$.*

Aalst [3] proves that the soundness is equivalent to the liveness and boundedness for WF-nets.

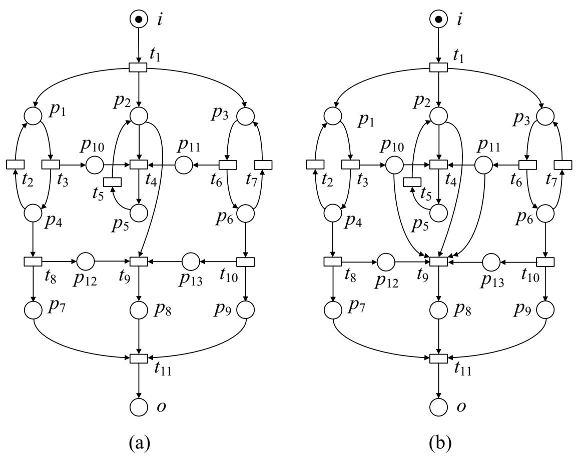


Fig. 1. (a) A WF-net; and (b) an reWF-net

Theorem 1. *Let $N = (P, T, F)$ be a WF-net, $N^E = (P, T \cup \{b\}, F \cup \{(b, i), (o, b)\})$, and $M_0 = i$. Then, N is sound if and only if (N^E, M_0) is live and bounded.*

Therefore, the following conclusion is obvious.

Corollary 1. *Let $N = (P, T, F)$ be a WF-net such that $(N^E, M_0) = (P, T \cup \{b\}, F \cup \{(b, i), (o, b)\}, i)$ is bounded. Then, N is sound if and only if (N^E, M_0) is live.*

2.3 reWF-Nets

reWF-nets are an extension to WF-nets in which some reset arcs are added. A reset arc can delete all tokens from related places and then the next computing can be started.

Definition 5. *A 4-tuple $N = (P, T, F, R)$ is an reWF-net if*

1. *$N = (P, T, F)$ is a WF-net; and*
2. *$R \subseteq [P \setminus \{o\} \times T]$ is the set of reset arcs.*

A reset arc is represented by a double-headed arrow in an reWF-net chart, and denoted as $[p, t]$ formally in order to differ from (p, t) which is the notation of an arc in general Petri nets. We denote $^\circ t = \{p \in P \mid [p, t] \in R\}$, $\forall t \in T$, as the set of places that connect with t by reset arcs. For example, Fig. 1(b) is an reWF-net that has two reset arcs $[p_{10}, t_9]$ and $[p_{11}, t_9]$. $^\circ t_9 = \{p_{10}, p_{11}\}$.

Rules of enabling and firing a transition are defined as follows:

Given an reWF-net $N = (P, T, F, R)$ and a marking M, if $\forall p \in {}^\bullet t$: $M(p) > 0$, then t is said to be *enabled* at M, which is denoted as $M[t\rangle$. Firing an enabled transition t produces a new marking M', which is denoted as $M[t\rangle M'$, such that $M'(p) = 0$ if $p \in {}^\circ t$; $M'(p) = M(p) - 1$ if $p \notin {}^\circ t \wedge p \in {}^\bullet t \setminus t^\bullet$; $M'(p) = M(p) + 1$ if $p \notin {}^\circ t \wedge p \in t^\bullet \setminus {}^\bullet t$; and $M'(p) = M(p)$ otherwise.

Obviously, the enabling rule of transitions of reWF-nets identifies with that of general Petri nets, but, after firing a transition, the tokens in those places that connect with the transition by reset arcs are all removed.

Other notions of reWF-nets, such as liveness, boundedness, and soundness, are the same as those of Petri nets and WF-nets, and are omitted here.

2.4 SAT Problem

The SAT problem, which is NP-complete [9], is used in this paper. Assume that there are n Boolean variables x_1, x_2,$\cdots$, and x_n. A literal l is a variable x or its negation $\neg x$. An expression G of conjunctive normal form (CNF) is a conjunction of m different terms and each term is a disjunction of different literals not containing a complementary pair x and $\neg x$. An expression H of disjunctive normal form (DNF) is a disjunction of m different terms and each term is a conjunction of different literals not containing a complementary pair x and $\neg x$.

Our proof is based on the 3SAT problem, i.e., each term has exactly three literals.

3SAT Problem: For a CNF expression G in which each term has exactly three literals, is there an assignment of variables such that $G = 1$?

For convenience, an equivalent problem, which is constructed by negating the CNF expression G, is used instead of the above problem. That is, DNF expressions are considered. This problem is denoted by $\overline{\text{3SAT}}$ [15].

$\overline{\textbf{3SAT}}$ Problem: For a DNF expression H in which each term has exactly three literals, is there an assignment of variables such that $H = 0$?

Without loss of generality, it is assumed that $m > 3$ (notice, m is the number of terms in the formula) and each variable and its negation are both in H. Additionally, we need the following assumption.

◇ There is no variable x such that it or its negation $\neg x$ occurs in all terms.

This assumption is reasonable. If there exists a variable such that it or its negation occurs in all terms, we may produce two expressions H' and H'' by assigning this variable the value 1 and 0, respectively. Then, $H = 0$ if and only if $H' = 0 \vee H'' = 0$, while the problem of $H' = 0 \vee H'' = 0$ belongs to 2SAT problem that can be decided in polynomial time [9].

2.5 LBA Acceptance Problem

An LBA is a Turing machine that has a finite tape containing initially a test string with a pair of bound symbols on either side.

Definition 6. *An 8-tuple $\Omega = (Q,\ \Gamma,\ \Sigma,\ \Delta,\ q_0,\ q_f,\ \#,\ \$)$ is an LBA if*

1. $Q = \{q_0,\ q_1, \cdots,\ q_m,\ q_f\}$, $m \geq 0$, *is a set of control states where* q_0 *is the initial state and* q_f *is the accept state;*
2. $\Gamma = \{a_1,\ a_2, \cdots,\ a_n\}$, $n > 0$, *is a tape alphabet;*
3. $\Sigma \subseteq \Gamma$ *is an input alphabet;*
4. $\Delta \subseteq Q \times \Gamma \times \{R,\ L\} \times Q \times \Gamma$ *is a set of transitions where R and L represent respectively that the read/write head moves right or left by one cell; and*
5. $\#$ *and* $\$$ *are two bound symbols that are next to the left and right sides of an input string, respectively.*

We assume that there is no transition from the accept state q_f because the computation is finished correctly once q_f is reached. We also assume that there is a transition sequence $(q_0, \_, \_, q', \_)$, $(q', \_, \_, q'', \_)$, $\cdots$, $(q^{(k)}, \_, \_, q_f, \_)$ in an LBA. Otherwise, the accept state is never reached.

If an LBA is at state p with the read/write head scanning a cell in which symbol a is stored, and there is a transition $\delta = (p,\ a,\ R,\ q,\ b) \in \Delta$, then firing δ causes: 1) the read/write head erase a from the cell, write b in the cell, and move right by one cell; and 2) the LBA be at state q.

LBA Acceptance Problem: For an LBA with a test string, does it accept the string?

This problem is PSPACE-complete even if the LBA is deterministic [9].

3 co-NP-Hardness of the Soundness of Bounded WF-Nets

In this section, we prove that the soundness problem of bounded WF-nets is co-NP-hard based on the $\overline{3\mathrm{SAT}}$ problem.

Let $x_1, x_2, \cdots$, and x_n be n variables and $H = D_1 \vee D_2 \vee \cdots \vee D_m = (l_{1,1} \wedge l_{1,2} \wedge l_{1,3}) \vee (l_{2,1} \wedge l_{2,2} \wedge l_{2,3}) \vee \cdots \vee (l_{m,1} \wedge l_{m,2} \wedge l_{m,3})$ be a DNF expression. The $\overline{3\mathrm{SAT}}$ problem is reducible in polynomial time to the liveness problem of bounded WF-nets, thereby proving that the soundness problem of bounded WF-nets is co-NP-hard by Corollary 1.

For each term D_k, $k \in \mathbb{N}_m$, let $\Psi(D_k)$ denote the set of subscripts of three variables in D_k. For example, if $D = \neg x_1 \wedge x_3 \wedge x_6$, then $\Psi(D) = \{1,\ 3,\ 6\}$.

For each DNF expression H, a WF-net can be constructed by the following method.

Construction 1

– $P = \{i,\ o,\ p_0\}$
  $\cup \{p_k,\ p'_k,\ v_k,\ v'_k,\ c_k,\ c'_k \mid k \in \mathbb{N}_n\}$
– $T = \{b,\ t_0,\ t'_0\}$
  $\cup \{d_j,\ d'_j \mid j \in \mathbb{N}_m\}$
  $\cup \{t_k,\ t'_k,\ e_k,\ e'_k \mid k \in \mathbb{N}_n\}$
– $F = \{(i,\ t_0), (t'_0,\ p_0), (o,\ b), (b,\ i)\}$
  $\cup \{(p_0,\ d_j),\ (d'_j,\ o) \mid j \in \mathbb{N}_m\}$
  $\cup \{(t_0,\ p_k),\ (p'_k,\ t'_0) \mid k \in \mathbb{N}_n\}$
  $\cup \{(d_j,\ v_k),\ (v'_k,\ d'_j) \mid k \in \mathbb{N}_n \setminus \Psi(D_j),\ j \in \mathbb{N}_m\}$
  $\cup \{(p_k,\ t_k),\ (p_k,\ t'_k),\ (t_k,\ p'_k),\ (t'_k,\ p'_k) \mid k \in \mathbb{N}_n\}$

$\cup\{(t_k,\ c_k),\ (t'_k,\ c'_k),\ (c_k,\ e_k),\ (c'_k,\ e'_k) \mid k \in \mathbb{N}_n\}$
$\cup\{(v_k,\ e_k),\ (v_k,\ e'_k),\ (e_k,\ v'_k),\ (e'_k,\ v'_k) \mid k \in \mathbb{N}_n\}$
$\cup\{(c_k,\ d_j) \mid l_{j,1} = x_k \vee l_{j,2} = x_k \vee l_{j,3} = x_k,\ k \in \mathbb{N}_n,\ j \in \mathbb{N}_m\}$
$\cup\{(c'_k,\ d_j) \mid l_{j,1} = \neg x_k \vee l_{j,2} = \neg x_k \vee l_{j,3} = \neg x_k,\ k \in \mathbb{N}_n,\ j \in \mathbb{N}_m\}$

– $M_0 = i$

For example, given a DNF expression $H_0 = (\neg x_3 \wedge x_4 \wedge x_5) \vee (x_1 \wedge \neg x_2 \wedge x_3) \vee (\neg x_1 \wedge x_2 \wedge x_4) \vee (\neg x_3 \wedge \neg x_4 \wedge \neg x_5)$, the constructed Petri net is shown as Fig. 2. Notice that, strictly speaking, the Petri net constructed by Construction 1 is not a WF-net but a trivial extension of a WF-net, i.e., if transition b, as a bridge connecting source and sink places, is deleted, then the resulting net is a WF-net. Sometimes, we do not distinguish between a WF-net and its trivial extension if no ambiguity is produced.

Intuitively, the constructed Petri net can be viewed as the composition of two partners. One (e.g., the left section of Fig. 2, i.e., the subnet generated by $\{i,\ t_0,\ t'_0\} \cup \{t_k,\ t'_k,\ p_k,\ p'_k,\ c_k,\ c'_k \mid k \in \mathbb{N}_n\}$) is to set an assignment for variables, and another (e.g., the right section of Fig. 2, i.e., the subnet generated by $\{o,\ p_0\} \cup \{e_k,\ e'_k,\ v_k,\ v'_k,\ c_k,\ c'_k \mid k \in \mathbb{N}_n\} \cup \{d_j,\ d'_j \mid j \in \mathbb{N}_m\}$) is to decide whether H is 0 under the assignment. Variables x_k and $\neg x_k$, $k \in \mathbb{N}_n$, are represented by places c_k and c'_k, respectively. When c_k (resp. c'_k) has a token, it means $x_k = 1$ (resp. $\neg x_k = 1$).

At the initial marking $M_0 = i$, only t_0 is enabled. After firing t_0, only t_1, t'_1, t_2, t'_2, $\cdots$, t_n, and t'_n are enabled, but for the pair t_k and t'_k, firing one will disable another since p_k has only one token. Firing t_k (resp. t'_k) means assigning the value 1 to x_k (resp. $\neg x_k$) since a token is moved into c_k (resp. c'_k). Obviously, x_k and $\neg x_k$ are not assigned true at the same time. In a word, this partner is to set an assignment for variables. Only after each variable is assigned a value, transition t'_0 is enabled. Only after firing t'_0, another partner can start to decide whether the expression H equals to 0 under the corresponding assignment.

Transitions d_1, d_2, $\cdots$, and d_m represent terms D_1, D_2, $\cdots$, and D_m respectively because c_k or c'_k ($k \in \mathbb{N}_n$) is an input place of d_j ($j \in \mathbb{N}_m$) if and only if x_k or $\neg x_k$ occurs in D_j. If none of d_1, d_2, $\cdots$, and d_m is enabled under an assignment, then it means $H = 0$ under this assignment. Notice that, this case ($H = 0$) implies that the Petri net is not live. If $H = 1$ under certain assignment, i.e., some terms are true under this assignment, then the corresponding transitions in $\{d_1,\ d_2, \cdots,\ d_m\}$ are enabled, but only one of these enabled transitions can be fired since p_0 has only one token. Let d_j be an enabled transition under this assignment. Then, after firing d_j, we have that for each $k \in \mathbb{N}_n \setminus \Psi(D_j)$, a token is put into v_k, the token assigned to c_k or c'_k is still retained, and tokens in ${}^\bullet d_j$ are removed. For removing the token from c_k or c'_k where $k \in \mathbb{N}_n \setminus \Psi(D_j)$, e_k or e'_k is competent, i.e., if c_k is marked, then only e_k is enabled, otherwise, only e'_k is enabled. After firing e_k or e'_k, $\forall k \in \mathbb{N}_n \setminus \Psi(D_j)$, d'_j can be fired. Notice that, after firing d'_j, only o has one token, and other places have no tokens. This decision is ended. Finally, firing b means that the initial marking is returned, i.e., a new decision may be started.

In what follows, it is proven that there is an assignment of variables such that $H = 0$ if and only if the constructed Petri nets is not live.

Lemma 1. *There is an assignment of variables such that $H = 0$ if and only if Petri net $(P,\ T,\ F,\ M_0)$ constructed by Construction 1 is not live.*

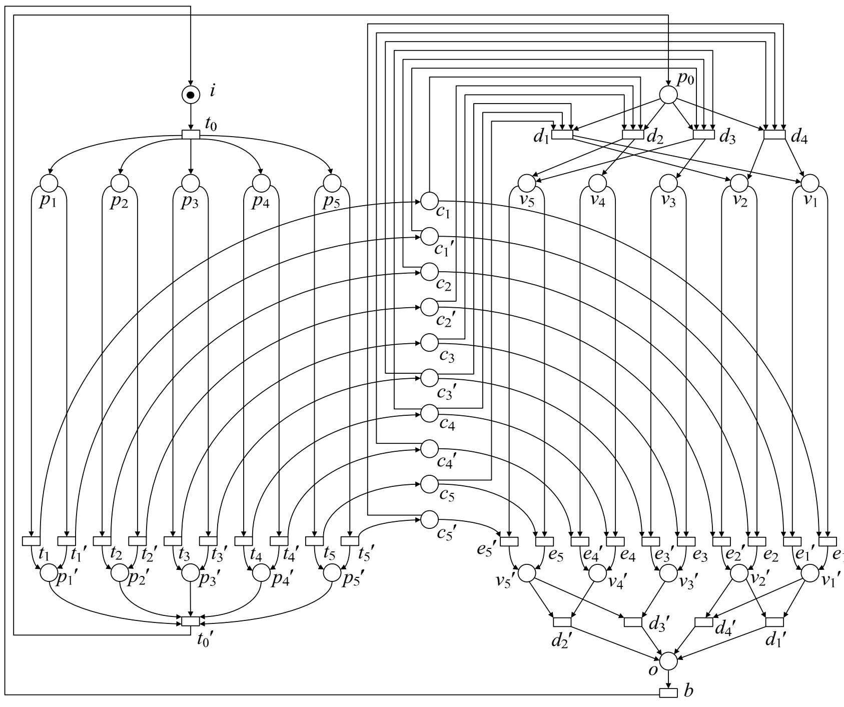


Fig. 2. The WF-net corresponding to $H_0 = (\neg x_3 \wedge x_4 \wedge x_5) \vee (x_1 \wedge \neg x_2 \wedge x_3) \vee (\neg x_1 \wedge x_2 \wedge x_4) \vee (\neg x_3 \wedge \neg x_4 \wedge \neg x_5)$ where x_k (resp. $\neg x_k$), $k \in \mathbb{N}_5$, corresponds to c_k (resp. c_k')

Proof: (***only if***) Let $(\kappa_1,\ \kappa_2, \cdots,\ \kappa_n)$ be an assignment of $(x_1,\ x_2, \cdots,\ x_n)$ such that $H = 0$, where $\kappa_k = 1$ or $\kappa_k = 0, \forall\, k \in \mathbb{N}_n$. Then, after firing the transition sequence $t_0\tau_1\tau_2 \cdots \tau_n t_0'$, where $\tau_i = t_i$ if $\kappa_i = 1$ or $\tau_i = t_i'$ if $\kappa_i = 0$, there is no enabled transition. This is because: if there is still an enabled transition after firing $t_0\tau_1\tau_2 \cdots \tau_n t_0'$, this transition must be the one in $\{d_1,\ d_2, \cdots,\ d_m\}$, which means that there is a term whose value is 1 under the assignment, thereby making $H = 1$. A contradiction is produced.

(***if***) (by contradiction) Assume that no assignment fulfills $H = 0$, i.e., for each assignment $(\kappa_1,\ \kappa_2, \cdots,\ \kappa_n)$, there always exists a term that is true. Through the analysis in the paragraph above this lemma, we know that for each assignment, $(P,\ T,\ F,\ M_0)$ always returns to its initial marking. Therefore, b, t_0, t_0', t_1, t_1', $\cdots$, t_n, and t_n' are obviously live. For each d_j, we can fire the corresponding transitions in $\{t_1,\ t_1', \cdots,\ t_n,\ t_n'\}$ to move tokens into the input places of d_j, thereby ensuring that d_j can be fired. That is, d_j and d_j' are also live. What remains is to show that e_1, e_1',$\cdots$, e_n, and e_n' are live. Obviously, if the pre-set of v_k is not empty, then e_k and e_k' are live, $\forall\, k \in \mathbb{N}_n$. By the previous assumption (i.e., there is no variable x such that it or its negation $\neg x$ occurs in each term), we know that for each v_k, its pre-set is not empty, because there always is a term D_j such that $k \in \mathbb{N}_n \setminus \Psi(D_j)$, i.e., x_k and $\neg x_k$ are not in D_j. Hence, e_k and e_k' are also live, $\forall\, k \in \mathbb{N}_n$. $\square$

Notice that, by the above conclusion we know that Petri net $(P,\ T,\ F,\ M_0)$ constructed by Construction 1 is live if and only if for each assignment of variables there is $H = 1$. The problem on deciding whether there is always $H = 1$ for each assignment of variables is co-NP-complete [9].

Corollary 2. *Petri net $(P,\ T,\ F,\ M_0)$ constructed by Construction 1 is live if and only if $H = 1$ for each assignment of variables.*

Lemma 2. *Let $(P,\ T,\ F,\ M_0)$ be the Petri net constructed for a DNF H by Construction 1. Then, $(P,\ T \setminus \{b\},\ F \setminus \{(o,\ b),(b,\ i)\},\ M_0)$ is a bounded WF-net.*

Proof: It is obvious that for each transition $t \in T \setminus \{b\}$ (resp. each place $p \in P$) in the net $(P,\ T \setminus \{b\},\ F \setminus \{(o,\ b),(b,\ i)\})$, there is a directed path from i to o such that t (resp. p) occurs in it. Therefore, $(P,\ T,\ F)$ is strongly connected. Therefore, $(P,\ T \setminus \{b\},\ F \setminus \{(o,\ b),(b,\ i)\})$ is a WF-net.

Obviously, $i,\ o,\ p_0,\ p_1,\ p'_1,\cdots,\ p_n,\ p'_n,\ v_1,\ v'_1,\ \cdots,\ v_n$, and v'_n are all bounded in $(P,\ T,\ F,\ M_0)$ because the two subnets generated by them represent the state transition of the two partners and are easily shown to be bounded. We only need to observe places $v_1,\ v'_1,\ \cdots,\ v_n$, and v'_n. In the case that $(P,\ T,\ F,\ M_0)$ is not live: all deadlock states satisfy that p_0 has a token, each pair c_k and c'_k has a token, and others have no token. In the case that $(P,\ T,\ F,\ M_0)$ is live: before o is marked, each pair c_k and c'_k has at most one token, and when o is marked, all tokens in $c_1,\ c'_1,\ \cdots,\ c_n$, and c'_n are removed. Hence, $(P,\ T,\ F,\ M_0)$ is bounded, thereby $(P,\ T \setminus \{b\},\ F \setminus \{(o,\ b),(b,\ i)\},\ M_0)$ bounded. □

Theorem 2. *The soundness problem for bounded WF-nets is co-NP-hard.*

Proof: For each DNF expression in which there are n variables and m terms and each term has 3 literals, it is easy to compute that the constructed WF-net has $6n + 3$ places, $4n + 2m + 3$ transitions, and $2mn + 14n - m + 4$ arcs. Therefore, the WF-net can be constructed in polynomial time (i.e., $O(2mn + 24n + m + 10)$). Therefore, it is known by Lemma 2 and Corollaries 1 and 2 that the soundness problem of bounded WF-net is co-NP-hard. □

4 PSPACE-Hardness of the Soundness of Bounded reWF-Nets

Obviously, the reachability, liveness, and soundness are all decidable for bounded reWF-nets since we can construct their reachability graph by which these properties can be decided. However, we are to show that they are PSPACE-hard. First, we prove that the liveness and soundness are equivalent for bounded reWF-nets.

Lemma 3. *Let $N = (P,\ T,\ F,\ R)$ be an reWF-net such that $(N^E,\ M_0) = (P,\ T \cup \{b\},\ F \cup \{(b,\ i),\ (o,\ b)\},\ R,\ i)$ is bounded. Then, N is sound if and only if $(N^E,\ M_0)$ is live.*

Proof: (***only if***) Please see Lemma 5.1 in [4].

(***if***) First, let $M_d = o$. Because $(N^E,\ M_0)$ is live, b is live. Hence, $\exists M \in R(N^E,\ M_0)$: $M[b\rangle$. Hence, $M \geq M_d$. Let $M_0[\sigma\rangle M$ and $M[b\rangle M'$. Next, we use the contradiction method to prove $M = M_d$. Assume that there is place $p \in P$ such that $M(p) > M_d(p) = 0$. Then, $M'(p) > M_d(p) = 0$ and $M'(i) = 1 = M_0(i)$ since there is no reset arc between b and p, i.e., firing b does not empty p. Hence, σb can fire infinitely, thereby making place p unbounded. This contradicts the boundedness of (N^E, M_0). Hence, for each marking $M \in R(N^E, M_0)$, if $M \geq M_d$, then $M = M_d = o$. Hence, $R(N^E, M_0) = R(N, M_0)$. Hence, by using the liveness of (N^E, M_0), we can easily prove that 1) $\forall M \in R(N, M_0)$: $M_d \in R(N, M)$; and 2) $\forall t \in T$, $\exists M \in R(N, M_0)$: $M[t\rangle$. Hence, N is sound. □

Next, we prove that the soundness problem is PSPACE-hard for bounded reWF-nets.

Given an LBA $\Omega = (Q, \Gamma, \Sigma, \Delta, q_0, q_f, \#, \$)$ with an input string S, an reWF-net can be constructed. The construction below is refered as **Construction 2**. We first assume that the length of S is l, $l \geq 0$, and the i-th element of S is denoted as S_i. $Q = \{q_0, q_1, \cdots, q_m, q_f\}$, $m \geq 0$. $\Gamma = \{a_1, a_2, \cdots, a_n\}$, $n > 0$. Cells storing $\#S\$$ are labeled $0, 1, \cdots, l$, and $l+1$, respectively.

– $P = \{i, o, p_0, p'_0\}$
  $\cup\{A_{0,\#}, A_{l+1,\$}, A_{i,j} \mid i \in \mathbb{N}_l, j \in \mathbb{N}_n\}$
  $\cup\{B_{i,j} \mid i \in \{0, 1, \cdots, l+1\}, j \in \{0, 1, \cdots, m\}\}$

A token in $A_{0,\#}$ (resp. $A_{l+1,\$}$) means that the tape cell 0 (resp. $l+1$) stores $\#$ (resp. $\$$). Therefore, once the computation starts, $A_{0,\#}$ (resp. $A_{l+1,\$}$) has a token until the computation ends since the two special symbols are not allowed to be replaced by other symbols. A token in $A_{i,j}$ means that the symbol in the cell i is a_j. A token in $B_{i,j}$ means that the read/write head is on the cell i and the machine is at state q_j. Notice that, in the computing process, the LBA is only at one state at any time, thus only one place in $\{B_{i,j} \mid i \in \{0, 1, \cdots, l+1\}, j \in \{0, 1, \cdots, m\}\}$ is marked by one token in the modeling process. Once the computation finishes, i.e., the LBA accepts the input string S, the token in $\{B_{i,j} \mid i \in \{0, 1, \cdots, l+1\}, j \in \{0, 1, \cdots, m\}\}$ is moved into p_0. The purpose of having p'_0 will be discussed later.

The sets of transitions, arcs, and reset arcs of the reWF-net are constructed as follows.

– b is a transition such that ${}^\bullet b = \{o\}$ and $b^\bullet = \{i\}$.
– t_s is a transition such that ${}^\bullet t_s = \{i\}$ and $t_s^\bullet = \{A_{0,\#}, A_{l+1,\$}, B_{0,0}\} \cup \{A_{i,j} \mid S_i = a_j, i \in \mathbb{N}_l, j \in \mathbb{N}_n\}$. In fact, $t_s^\bullet$ corresponds to the initial configuration of the LBA, i.e., tokens in $\{A_{0,\#}, A_{l+1,\$}, A_{i,j} \mid S_i = a_j, i \in \mathbb{N}_l, j \in \mathbb{N}_n\}$ represent the input string with the two bound symbols, and the token in $B_{0,0}$ represents that the machine is at the initial state q_0 and the read/write head is on the leftmost cell.
– t'_s is a transition such that ${}^\bullet t'_s = \{p'_0\}$ and $t'^\bullet_s = \{p_0\} \cup \{A_{0,\#}, A_{l+1,\$}, A_{i,j} \mid i \in \mathbb{N}_l, j \in \mathbb{N}_n\} \cup \{B_{i,j} \mid i \in \{0, 1, \cdots, l+1\}, j \in \{0, 1, \cdots, m\}\}$. Later, we will explain the reason of having t'_s.
– t_e is a transition such that ${}^\bullet t_e = \{p_0\}$ and $t_e^\bullet = \{p'_0\}$. Only t_e connects with reset arcs such that ${}^\circ t_e = \{p_0\} \cup \{A_{0,\#}, A_{l+1,\$}, A_{i,j} \mid i \in \mathbb{N}_l, j \in \mathbb{N}_n\} \cup \{B_{i,j} \mid i \in \{0, 1, \cdots, l+1\}, j \in \{0, 1, \cdots, m\}\}$. The token in p_0 means that the computation ends, and then firing t_e will empty all places in $\{p_0\} \cup \{A_{0,\#}, A_{l+1,\$}, A_{i,j} \mid i \in \mathbb{N}_l, j \in \mathbb{N}_n\} \cup \{B_{i,j} \mid i \in \{0, 1, \cdots, l+1\}, j \in \{0, 1, \cdots, m\}\}$, i.e., the tape is emptied and the machine is not at any state.
– t'_e is a transition such that ${}^\bullet t'_e = \{p'_0\}$ and $t'^\bullet_e = \{o\}$

For each transition $\delta \in \varDelta$, we construct transitions of the reWF-net. We first consider transitions in $\varDelta$ that make the LBA to enter the accept state q_f.

- If δ is the form of $(q_h,\ \#,\ R,\ q_f,\ \#)$, $h \in \{0,\ 1, \cdots,\ m\}$, i.e., the LBA halts correctly and the read/write head is on the leftmost cell, we construct a transition t of the reWF-net such that ${}^\bullet t = \{A_{0,\#},\ B_{0,h}\}$ and $t^\bullet = \{A_{0,\#},\ p_0\}$.
- If δ is the form of $(q_h,\ \$,\ L,\ q_f,\ \$)$, $h \in \{0,\ 1, \cdots,\ m\}$, i.e., the LBA halts correctly and the read/write head is on the rightmost cell, we construct a transition t such that ${}^\bullet t = \{A_{l+1,\$},\ B_{l+1,h}\}$ and $t^\bullet = \{A_{l+1,\$},\ p_0\}$.
- If δ is the form of $(q_h,\ a_j,\ L/R,\ q_f,\ a_k)$, $h \in \{0,\ 1, \cdots,\ m\}, j,\ k \in \mathbb{N}_n$, i.e., the LBA halts correctly but the read/write head is possibly on any cell, we construct for each cell r ($r \in \mathbb{N}_l$) a transition t_r. Formally, $\forall\, r \in \mathbb{N}_l$, a transition t_r is constructed such that ${}^\bullet t_r = \{A_{r,j},\ B_{r,h}\}$ and $t_r^\bullet = \{A_{r,k},\ p_0\}$.

Next, we consider other transitions in $\varDelta$ that have no p_f.

- If δ is the form of $(q_h,\ \#,\ R,\ q_i,\ \#)$, $h,\ i \in \{0,\ 1, \cdots,\ m\}$, i.e., the read/write head scans $\#$, $\#$ is rewritten, the read/write head moves right, and the state is changed into q_i from q_h. For this δ, we construct a transition t of the reWF-net such that ${}^\bullet t = \{A_{0,\#},\ B_{0,h}\}$ and $t^\bullet = \{A_{0,\#},\ B_{1,i}\}$.
- If δ is the form of $(q_h,\ \$,\ L,\ q_i,\ \$)$, $h,\ i \in \{0,\ 1, \cdots,\ m\}$, i.e., the read/write head scans $\$$, $\$$ is rewritten, the read/write head moves left, and the state is changed into q_i from q_h. For this δ, we construct a transition t such that ${}^\bullet t = \{A_{l+1,\$},\ B_{l+1,h}\}$ and $t^\bullet = \{A_{l+1,\$},\ B_{l,i}\}$.
- If δ is the form of $(q_h,\ a_j,\ R,\ q_i,\ a_k)$, $h, i \in \{0,\ 1, \cdots,\ m\}, j,\ k \in \mathbb{N}_n$, then we construct l transitions, i.e., we should consider each cell. Formally, $\forall\, r \in \mathbb{N}_l$, a transition t_r is constructed such that ${}^\bullet t_r = \{A_{r,j},\ B_{r,h}\}$ and $t_r^\bullet = \{A_{r,k},\ B_{r+1,i}\}$.
- If δ is the form of $(q_h,\ a_j,\ L,\ q_i,\ a_k)$, $h,\ i \in \{0,\ 1, \cdots,\ m\}, j,\ k \in \mathbb{N}_n$, then we also construct l transitions, i.e., $\forall\, r \in \mathbb{N}_l$, a transition t_r is constructed such that ${}^\bullet t_r = \{A_{r,j},\ B_{r,h}\}$ and $t_r^\bullet = \{A_{r,k},\ B_{r-1,i}\}$.

In the running process of the Petri net, a marking over $\{p_0\} \cup \{A_{0,\#},\ A_{l+1,\$},\ A_{i,j} \mid i \in \mathbb{N}_l,\ j \in \mathbb{N}_n\} \cup \{B_{i,j} \mid i \in \{0,\ 1, \cdots,\ l+1\},\ j \in \{0,\ 1, \cdots,\ m\}\}$ corresponds to a configuration of the LBA, i.e., tokens in $\{A_{0,\#},\ A_{l+1,\$},\ A_{i,j} \mid i \in \mathbb{N}_l,\ j \in \mathbb{N}_n\}$ correspond to the string on the tape, and the token in $\{p_0\} \cup \{B_{i,j} \mid i \in \{0,\ 1, \cdots,\ l+1\},\ j \in \{0,\ 1, \cdots,\ m\}\}$ represents the current state of the LBA.

We know that the LBA halts correctly and accepts the input string when a token enters p_0, and at this marking, only t_e is enabled. Firing t_e makes all places in $\{p_0\} \cup \{A_{0,\#},\ A_{l+1,\$},\ A_{i,j} \mid i \in \mathbb{N}_l,\ j \in \mathbb{N}_n\} \cup \{B_{i,j} \mid i \in \{0,\ 1, \cdots,\ l+1\},\ j \in \{0,\ 1, \cdots,\ m\}\}$ emptied because t_e have reset arcs with these places. However, some transitions, which correspond to $\varDelta$, are not necessarily enabled in the computing process. It is because for an LBA with an acceptable input string, not all transitions of the LBA are used in the deciding process. Therefore, we use t_s' to produce a token for each place in $\{p_0\} \cup \{A_{0,\#},\ A_{i,j},\ A_{l+1,\$} \mid i \in \mathbb{N}_l,\ j \in \mathbb{N}_n\} \cup \{B_{i,j} \mid i \in \{0,\ 1, \cdots,\ l+1\},\ j \in \{0,\ 1, \cdots,\ m\}\}$, which makes all transitions corresponding to $\varDelta$ have an enabling right again. Clearly, once t_e is fired, all places in $\{p_0\} \cup \{A_{0,\#},\ A_{i,j},\ A_{l+1,\$} \mid i \in \mathbb{N}_l,\ j \in$

$\mathbb{N}_n\} \cup \{B_{i,j} \mid i \in \{0,\ 1, \cdots,\ l+1\},\ j \in \{0,\ 1, \cdots,\ m\}\}$ are emptied again. This guarantees that all transitions corresponding to Δ are live when the LBA accepts the input string. Obviously, if t'_e and then b are fired, the initial marking, $M_0 = i$, is returned.

Notice that, places in $\{A_{0,\#},\ A_{l+1,\$},\ A_{i,j} \mid i \in \mathbb{N}_l,\ j \in \mathbb{N}_n\} \cup \{B_{i,j} \mid i \in \{0,\ 1, \cdots,\ l+1\},\ j \in \{0,\ 1, \cdots,\ m\}\}$ are set in order to consider all possible cases for each cell, but because Δ is finite, some of these places are not used or only have input or output transitions. This makes the constructed net not strongly connected. Therefore, we add a transition d such that

– for each place p in $\{A_{0,\#},\ A_{l+1,\$},\ A_{i,j} \mid i \in \mathbb{N}_l,\ j \in \mathbb{N}_n\} \cup \{B_{i,j} \mid i \in \{0,\ 1, \cdots,\ l+1\},\ j \in \{0,\ 1, \cdots,\ m\}\}$, $p \in {}^{\bullet}d$ and $p \in d^{\bullet}$.

This ensures that the constructed net is strongly connected. Obviously, d does not influence the behavior of the constructed Petri net, and only after firing t'_s, d is enabled. Note that, even though there is no transition d, there also exists a direct path from place i to place o, which is guaranteed by the previous assumption (i.e., for an LBA, there always exists a transition sequence $(q_0, \_, \_, q', \_)$, $(q', \_, \_, q'', \_)$, $\cdots$, $(q^{(k)}, \_, \_, q_f, \_)$.).

For example, the LBA $\Omega_0 = (Q,\ \Gamma,\ \Sigma,\ \Delta,\ q_0,\ q_f,\ \#,\ \$)$ can produce the language $\{a^{i_1}b^{i_1}a^{i_2}b^{i_2} \cdots a^{i_m}b^{i_m} \mid i_1, i_2, \cdots, i_m, m \in \mathbb{N}\}$, where

– $Q = \{q_0,\ q_1,\ q_2,\ q_3,\ q_f\}$
– $\Gamma = \{a,\ b,\ X\}$
– $\Sigma = \{a,\ b\}$
– $\Delta = \{(q_0,\ \#,\ R,\ q_1,\ \#),\ (q_1,\ \$,\ L,\ q_f,\ \$),\ (q_1,\ X,\ R,\ q_1,\ X),\ (q_1,\ a,\ R,\ q_2,\ X),$
$(q_2,\ a,\ R,\ q_2,\ a),\ (q_2,\ X,\ R,\ q_2,\ X),\ (q_2,\ b,\ L,\ q_3,\ X),\ (q_3,\ a,\ L,\ q_3,\ a),$
$(q_3,\ X,\ L,\ q_3,\ X),\ (q_3,\ \#,\ R,\ q_1,\ \#)\}$

Notice that, this LBA accepts the empty string. For the LBA with the input string ab, the Petri net constructed by Construction 2 is $(N^E,\ M_0) = (P,\ T,\ F,\ R,\ i)$ where

– $P = \{i,\ o,\ p_0,\ p'_0\}$
$\cup\{A_{0,\#},\ A_{3,\$},\ A_{i,j} \mid i \in \mathbb{N}_2,\ j \in \mathbb{N}_3\}$
$\cup\{B_{i,j} \mid i,\ j \in \{0,\ 1,\ 2,\ 3\}\}$
– $T = \{b,\ t_s,\ t'_s,\ t_e,\ t'_e,\ d\}$
$\cup\{t_0,\ t_1,\ t_2,\ t_{i,j} \mid i \in \mathbb{N}_7,\ j \in \mathbb{N}_2\}$
– $F = \{(p_0,\ t_e),\ (t_e,\ p'_0),\ (p'_0,\ t'_e),\ (t'_e,\ o),\ (o,\ b),\ (b,\ i)\}$
$\cup\{(i,\ t_s),\ (t_s,\ A_{0,\#}),\ (t_s,\ A_{3,\$}),\ (t_s,\ A_{1,1}),\ (t_s,\ A_{2,2}),\ (t_s,\ B_{0,0})\}$
$\cup\{(t_0,\ p_0),\ (t_0,\ A_{3,\$}),\ (A_{3,\$},\ t_0),\ (B_{3,1},\ t_0)\}$
$\cup\{(t_1,\ B_{1,1}),\ (t_1,\ A_{0,\#}),\ (A_{0,\#},\ t_1),\ (B_{0,0},\ t_1)\}$
$\cup\{(t_2,\ B_{1,1}),\ (t_2,\ A_{0,\#}),\ (A_{0,\#},\ t_2),\ (B_{0,3},\ t_2)\}$
$\cup\{(t_{1,1},\ B_{2,1}),\ (t_{1,1},\ A_{1,3}),\ (A_{1,3},\ t_{1,1}),\ (B_{1,1},\ t_{1,1})\}$
$\cup\{(t_{1,2},\ B_{3,1}),\ (t_{1,2},\ A_{2,3}),\ (A_{2,3},\ t_{1,2}),\ (B_{2,1},\ t_{1,2})\}$
$\cup\{(t_{2,1},\ B_{2,1}),\ (t_{2,1},\ A_{1,3}),\ (A_{1,1},\ t_{2,1}),\ (B_{1,1},\ t_{2,1})\}$
$\cup\{(t_{2,2},\ B_{3,1}),\ (t_{2,2},\ A_{2,3}),\ (A_{2,1},\ t_{2,2}),\ (B_{2,1},\ t_{2,2})\}$
$\cup\{(t_{3,1},\ B_{2,2}),\ (t_{3,1},\ A_{1,1}),\ (A_{1,1},\ t_{3,1}),\ (B_{1,2},\ t_{3,1})\}$
$\cup\{(t_{3,2},\ B_{3,2}),\ (t_{3,2},\ A_{2,1}),\ (A_{2,1},\ t_{3,2}),\ (B_{2,2},\ t_{3,2})\}$
$\cup\{(t_{4,1},\ B_{2,2}),\ (t_{4,1},\ A_{1,3}),\ (A_{1,3},\ t_{4,1}),\ (B_{1,2},\ t_{4,1})\}$

$\cup\{(t_{4,2}, B_{3,2}), (t_{4,2}, A_{2,3}), (A_{2,3}, t_{4,2}), (B_{2,2}, t_{4,2})\}$
$\cup\{(t_{5,1}, B_{0,3}), (t_{5,1}, A_{1,3}), (A_{1,2}, t_{5,1}), (B_{1,2}, t_{5,1})\}$
$\cup\{(t_{5,2}, B_{1,3}), (t_{5,2}, A_{2,3}), (A_{2,2}, t_{5,2}), (B_{2,2}, t_{5,2})\}$
$\cup\{(t_{6,1}, B_{0,3}), (t_{6,1}, A_{1,1}), (A_{1,1}, t_{6,1}), (B_{1,3}, t_{6,1})\}$
$\cup\{(t_{6,2}, B_{1,3}), (t_{6,2}, A_{2,1}), (A_{2,1}, t_{6,2}), (B_{2,3}, t_{6,2})\}$
$\cup\{(t_{7,1}, B_{0,3}), (t_{7,1}, A_{1,3}), (A_{1,3}, t_{7,1}), (B_{1,3}, t_{7,1})\}$
$\cup\{(t_{7,2}, B_{1,3}), (t_{7,2}, A_{2,3}), (A_{2,3}, t_{7,2}), (B_{2,3}, t_{7,2})\}$
$\cup\{(p'_0, t'_s), (t'_s, A_{0,\#}), (t'_s, A_{3,\$})\}$
$\cup\{(t'_s, A_{i,j}) \mid i \in \mathbb{N}_2, j \in \mathbb{N}_3\}$
$\cup\{(t'_s, B_{i,j}) \mid i, j \in \{0, 1, 2, 3\}\}$
$\cup\{(d, A_{0,\#}), (d, A_{3,\$}), (A_{0,\#}, d), (A_{3,\$}, d)\}$
$\cup\{(d, A_{i,j}), (A_{i,j}, d) \mid i \in \mathbb{N}_2, j \in \mathbb{N}_3\}$
$\cup\{(d, B_{i,j}), (B_{i,j}, d) \mid i, j \in \{0, 1, 2, 3\}\}$

– $R = \{[A_{0,\#}, t_e], [A_{3,\$}, t_e], [p_0, t_e]\}$
$\cup\{[A_{i,j}, t_e] \mid i \in \mathbb{N}_2, j \in \mathbb{N}_3\}$
$\cup\{[B_{i,j}, t_e] \mid i, j \in \{0, 1, 2, 3\}\}$

Fig. 3 shows another constructed Petri net that corresponds to the above LBA Ω_0 with an empty string. Notice, the following (reset) arcs are not drawn in Fig. 3:

$t_s'^{\bullet} = \{p_0, A_{0,\#}, A_{1,\$}, B_{0,0}, B_{0,1}, B_{0,2}, B_{0,3}, B_{1,0}, B_{1,1}, B_{1,2}, B_{1,3}\}$
${}^{\circ}t_e = \{p_0, A_{0,\#}, A_{1,\$}, B_{0,0}, B_{0,1}, B_{0,2}, B_{0,3}, B_{1,0}, B_{1,1}, B_{1,2}, B_{1,3}\}$
${}^{\bullet}d = d^{\bullet} = \{A_{0,\#}, A_{1,\$}, B_{0,0}, B_{0,1}, B_{0,2}, B_{0,3}, B_{1,0}, B_{1,1}, B_{1,2}, B_{1,3}\}$

Clearly, if there are no transition d as well as the related arcs, the constructed net is not strongly connected. Notice that, because the input string is an empty one, there are no transitions for (q_1, X, R, q_1, X), (q_1, a, R, q_2, X), (q_2, a, R, q_2, a), (q_2, X, R, q_2, X), (q_2, b, L, q_3, X), (q_3, a, L, q_3, a), and (q_3, X, L, q_3, X). Transitions t_0, t_1, and t_2 correspond to $(q_0, \#, R, q_1, \#)$, $(q_3, \#, R, q_1, \#)\}$, and $(q_1, \$, L, q_f, \$)$, respectively. Firing t_s produces respectively one token for $A_{0,\#}$, $A_{1,\$}$, and $B_{0,0}$, which represents the initial configuration of the LBA, i.e., the tape stores an empty string (i.e., only two bound symbols are on the tape), the machine is at state q_0, and the read/write head is on the leftmost cell. At this marking, only transition t_0 is enabled. t_0 corresponds to $(q_0, \#, R, q_1, \#)$. At the initial configuration, only $(q_0, \#, R, q_1, \#)$ is enabled. After firing $(q_0, \#, R, q_1, \#)$, the configuration of the LBA is that the machine is at state q_1 and the read/write head moves right (i.e., it is moved on the cell storing \$). This is in accordance with t_0 because firing t_0 removes the token from $B_{0,0}$ and put a token into $B_{1,1}$. At this marking, only t_2 is enabled, which is also in accordance with the LBA since at the corresponding configuration only $(q_1, \$, L, q_f, \$)$ is valid. Firing t_2 moves a token into p_0, which means the LBA accepts this input string.

Lemma 4. *An LBA accepts an input string if and only if the Petri net constructed by Construction 2 is live.*

Proof: (***only if***) Because the LBA accepts the input string, we have that for each marking $M \in R(N, M_0)$ such that $M_0[\sigma\rangle M$ but t_e is not in σ, there is a reachable marking

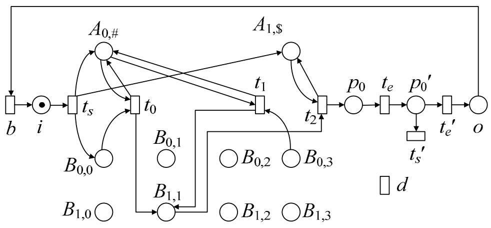


Fig. 3. The reWF-net corresponding to the LBA Ω_0 with the empty string

$M' \in R(N, M)$ such that p_0 is marked at M'. At marking M', only transition t_e is enabled. After firing t_e, marking $M'' = p_0'$ is reached. At marking M'', only t_e' or t_s' is enabled. Firing t_s' produces a token for each place in $\{p_0\} \cup \{A_{0,\#}, A_{i,j}, A_{l+1,\$} \mid i \in \mathbb{N}_l, j \in \mathbb{N}_n\} \cup \{B_{i,j} \mid i \in \{0, 1, \cdots, l+1\}, j \in \{0, 1, \cdots, m\}\}$, which makes transition d and transitions corresponding to Δ have an enabling right. Clearly, once t_e is fired again, all places in $\{p_0\} \cup \{A_{0,\#}, A_{i,j}, A_{l+1,\$} \mid i \in \mathbb{N}_l, j \in \mathbb{N}_n\} \cup \{B_{i,j} \mid i \in \{0, 1, \cdots, l+1\}, j \in \{0, 1, \cdots, m\}\}$ are emptied again. If firing t_e' at marking M'' and then firing b, the initial marking, $M_0 = i$, is returned. Therefore, the constructed Petri net is live.

(***if***) By Construction 2 we know that when the constructed Petri net is live, there is a reachable marking such that transition t_e is enabled at this marking, i.e., this marking marks place p_0. By Construction 2 we know that if place p_0 can be marked by some reachable marking, then the LBA accepts the input string. □

Lemma 5. *Let (P, T, F, R, M_0) be the Petri net constructed for an LBA with an input string by Construction 2. Then, $(P, T \setminus \{b\}, F \setminus \{(o, b), (b, i)\}, R, M_0)$ is a bounded reWF-net.*

Proof: Clearly, (P, T, F, R, M_0) is strongly connected after deleting all reset arcs. Therefore, $(P, T \setminus \{b\}, F \setminus \{(o, b), (b, i)\}, R, M_0)$ is an reWF-net. For boundedness, we only need to observe the constructed transitions corresponding to Δ. Since each of them has two input places and two output places, they neither increase nor decrease the number of tokens at any time. □

Theorem 3. *The soundness problem for bounded reWF-nets is PSPACE-hard.*

Proof: It is derived by Lemmas 3, 4, and 5. Notice that, the reWF-net can be constructed in $O(l \cdot (m + n + k))$ time where $l = |S|$, $m = |Q|$, $n = |\Gamma|$, and $k = |\Delta|$. □

The soundness of bounded reWF-nets is decidable but PSPACE-hard. However, it is still an open problem whether the boundedness problem of reWF-nets is decidable. What we know is that the boundedness problem is undecidable for general Petri nets with reset arcs [7,8].

Similarly, based on the LBA Acceptance problem, we can prove that the soundness problem of bounded WF-nets with inhibitor arcs is PSPACE-hard. The boundedness problem is undecidable for general Petri nets with inhibitor arcs [10,19].

5 Conclusion

The SAT problem is shown to be polynomial time reducible to the soundness problem for bounded WF-nets. This implies the latter is co-NP-hard. The soundness problem of bounded reWF-nets is proven to be PSPACE-hard by reducing the LBA Acceptance problem to it in polynomial time. Co-NP-hardness (resp. PSPACE-hardness) of the soundness problem of bounded WF-nets (resp. reWF-nets) shows a lower limit of the complexity. Therefore, future work focuses on finding if the soundness problem is co-NP-complete for bounded WF-nets and PSPACE-complete for bounded reWF- and inWF-nets.

The results in this paper is meaningful in theory. However, WF- and reWF-nets are of the strong application background in industry and often have special structures such as AND-join and OR-split [3]. Therefore, it is our future work to explore efficient analysis methods for specially structural WF- and reWF-nets.

Acknowledgement. We would like to thank Professor W.M.P. van der Aalst who encourages us to explore the complexity of the soundness problem for general WF-nets, and also thank four anonymous reviewers whose constructive comments improve the qualities of this paper. This paper was partially supported by research grant ”SUTD-ZJU/RES/05/201”.

References

1. Van der Aalst, W.M.P.: Interorganizational Workflows: An Approach Based on Message Sequence Charts and Petri Nets. System Analysis and Modeling 34, 335–367 (1999)
2. Van der Aalst, W.M.P.: Loosely Coupled Interorganizational Wokflows: Modeling and Analyzing Workflows Crossing Organizational Boundaries. Inf. Manage. 37, 67–75 (2000)
3. Van der Aalst, W.M.P.: Structural Characterizations of Sound Workflow Nets. Computing Science Report 96/23, Eindhoven University of Technology (1996)
4. Van der Aalst, W.M.P., Van Hee, K.M., Ter Hofstede, A.H.M., Sidorova, N., Verbeek, H.M.W., Voorhoeve, M., Wynn, M.T.: Soundness of Workflow Nets: Classification, Decidability, and Analysis. Formal Aspects of Computing 23, 333–363 (2011)
5. Cheng, A., Esparza, J., Palsberg, J.: Complexity Results for 1-safe Nets. Theoretical Computer Science 147, 117–136 (1995)
6. Desel, J., Esparza, J.: Free Choice Petri Nets. Cambridge Tracts in Theoretical Computer Science, vol. 40. Cambridge University Press, Cambridge (1995)
7. Dufourd, C., Finkel, A., Schnoebelen, P.: Reset Nets Between Decidability and Undecidability. In: Larsen, K.G., Skyum, S., Winskel, G. (eds.) ICALP 1998. LNCS, vol. 1443, pp. 103–115. Springer, Heidelberg (1998)
8. Dufourd, C., Jančar, P., Schnoebelen, P.: Boundedness of Reset P/T Nets. In: Wiedermann, J., Van Emde Boas, P., Nielsen, M. (eds.) ICALP 1999. LNCS, vol. 1644, pp. 301–310. Springer, Heidelberg (1999)
9. Garey, M.R., Johnson, D.S.: Computer and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman and Company (1976)
10. Hack, M.: Petri Net Languages. Technical Report 159. MIT (1976)
11. 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)

12. Kang, M.H., Park, J.S., Froscher, J.N.: Access Control Mechanisms for Inter-organizational Workflow. In: Proc. of the Sixth ACM Symposium on Access Control Models and Technologies, pp. 66–74. ACM Press, New York (2001)
13. Kindler, E.: The ePNK: An Extensible Petri Net Tool for PNML. In: Kristensen, L.M., Petrucci, L. (eds.) PETRI NETS 2011. LNCS, vol. 6709, pp. 318–327. Springer, Heidelberg (2011)
14. Kindler, E., Martens, A., Reisig, W.: Inter-operability of Workflow Applications: Local Criteria for Global Soundness. In: van der Aalst, W.M.P., Desel, J., Oberweis, A. (eds.) BPM 2000. LNCS, vol. 1806, pp. 235–253. Springer, Heidelberg (2000)
15. Ohta, A., Tsuji, K.: NP-hardness of Liveness Problem of Bounded Asymmetric Choice Net. IEICE Trans. Fundamentals E85-A, 1071–1074 (2002)
16. Tiplea, F.L., Bocaneala, C.: Decidability Results for Soundness Criteria of Resource-Constrained Workflow Nets. IEEE Trans. on Systems, man, and Cybernetics, Part A: Systems and Humans 42, 238–249 (2011)
17. Verbeek, H.M.W., Van der Aalst, W.M.P., Ter Hofstede, A.H.M.: Verifying Worklows with Cancellation Regions and OR-joins: An Approach Based on Relaxed Soundness and Invariants. Computer Journal 50, 294–314 (2007)
18. Verbeek, H.M.W., Wynn, M.T., Van der Aalst, W.M.P., Ter Hofstede, A.H.M.: Reduction Rules for Reset/Inhibitor Nets. BMP Center Report BMP-07-13, BMP-center.org (2007)
19. Van der Vlugt, S., Kleijn, J., Koutny, M.: Coverability and Inhibitor Arcs: An Example. Technical Report 1293, University of Newcastle Upon Tyne (2011)

Cost Soundness for Priced Resource-Constrained Workflow Nets

María Martos-Salgado and Fernando Rosa-Velardo*

Sistemas Informáticos y Computación, Universidad Complutense de Madrid
mrmartos@estumail.ucm.es, fernandorosa@sip.ucm.es

Abstract. We extend workflow Petri nets (wf-nets) with discrete prices, by associating a price to the execution of a transition and, more importantly, to the storage of tokens. We first define the soundness problem for priced wf-nets, that of deciding whether the workflow can always terminate properly, where in the priced setting "properly" also means that the execution does not cost more than a given threshold. Then, we study soundness of resource-constrained workflow nets (rcwf-nets), an extension of wf-nets for the modeling of concurrent executions of a workflow, sharing some global resources. We develop a framework in which to study soundness for priced rcwf-nets, that is parametric on the cost model. Then, that framework is instantiated, obtaining the cases in which the sum, the maximum, the average and the discounted sum of the prices of each all instances are considered. We study the relations between these properties, together with their decidability.

1 Introduction

Workflow nets (wf-nets) are an important formalism for the modeling of business processes, or workflow management systems [1, 2]. Roughly, a wf-net is a Petri net with two special places, *in* and *out*. Its initial marking is that with a token in the place *in* and empty everywhere else, which models the situation in which a task has been scheduled. The basic correctness notion for a workflow is that of soundness. Intuitively, a workflow is sound if it cannot go wrong, so that no supervisor is needed in order to ensure the completion of the task under a fairness assumption. More precisely, and in terms of wf-nets, soundness implies that at any reachable state, it is possible to reach the final state, that with a token in the place *out*, and empty elsewhere. Soundness is decidable for wf-nets, and even polynomial for free-choice wf-nets [2].

Recent works [3–5] study an extension of wf-nets, called resource-constrained wf-nets (rcwf-nets) in which several instances of a workflow execute concurrently, assuming that those instances share some global resources. Even if a singe instance of an rcwf-net is sound, several instances could deadlock because of these shared resources. In [3] the authors define dynamic soundness, the condition

* Authors supported by the MEC Spanish project DESAFIOS10 TIN2009-14599-C03-01, and Comunidad de Madrid program PROMETIDOS S2009/TIC-1465.

stating the existence of a minimum amount of resources for which any number of instances running simultaneously can always reach the final state, that in which all the tasks have been completed and the number of resources is as in the initial state. The paper [4] defines another notion of dynamic soundness, in terms of the absence of instance deadlocks in rcwf-nets, fixing the initial amount of resources though keeping the condition that instances must not change the number of resources. In [5] we continued the work in [4], but we considered that instances may create or consume resources. We proved this notion of dynamic soundness to be undecidable, and we identified a subclass of rcwf-nets, called proper, for which dynamic soundness is decidable.

In the fields of business process management or web services, the importance of QoS properties in general and cost estimation in particular has been identified as central in numerous works [6–10]. As an example, in the previously mentioned models, it may be possible to reach the final marking in different ways, due to the different interleavings of the execution, or to the inherent non-determinism in wf-nets. Moreover, in the case of rcwf-nets, an instance locking some resource may force another instance to take a "less convenient" path (in terms of money, energy or gas emissions, for example), that does not use the locked resource. However, the reason why a workflow should prefer one path over another is something that lies outside the model.

In order to study these problems, in this paper we add prices to our nets, similarly as done for the (untimed) priced Petri nets in [11]. We consider different types of prices, modeled as tuples of integers or naturals. More precisely, we add firing costs to transitions, and more importantly, storage costs to places. Then, the price of firing a transition is computed as the cost of its firing plus the cost of storing all the tokens in the net while the transition is fired. The price of a run is defined as the sum of the prices of the firings of its transitions. Then, we say that a workflow net is price-safe if the price of each of its runs stays under a given threshold. When costs are integers, we prove that price-safety is undecidable, so that in the rest of the paper we restrict ourselves to non-negative costs.

In this priced setting, we restate the soundness problems. For ordinary wf-nets, this is straightforward: a priced wf-net is sound if essentially we can always reach the final marking, without spending more than a given budget. For priced rcwf-nets, the definition of soundness is not so straightforward, since it must consider the behavior of an arbitrary number of instances. We consider a parametric definition of soundness for priced rcwf-nets. For any run, we collect the prices of every instance in the run, so that soundness is parametric in the way in which local prices are aggregated to obtain a global price. The definition is open to many different variants, we study several such variants in this paper: the maximum, the sum, the average and the discounted sum.

We prove decidability of price-safety for the sum, the maximum, and a finite version of discounted sum, relying on the decidability of coverability for a class of Petri nets with names, broadcasts and whole place operations, that can be seen as an unordered version of Data Nets with name creation [12, 13]. In these cases we have decidability of priced soundness within the proper subclass. As a corollary, we obtain the corresponding results for ordinary priced wf-nets (with non-negative costs). For the average, we reduce soundness to the unpriced case, so that it is decidable for proper rcwf-nets. However, price-safety remains open.

Related Work. In parallel to the works on wf-nets and rcwf-nets, there has recently been an increasing interest in the study of quantitative aspects of both finite and infinite state systems. In [14] the authors consider quantitative generalizations of classical languages, using weighted finite automata, that assign real numbers to words, instead of boolean values. They study different problems, which are defined in terms of how they assign a value to each run. In particular, they assign the maximum, limsup, liminf, average and discounted sum of the transition weights of the run.

Numerous works extend timed automata with prices [11, 15, 16]. E.g., the paper [11] defines a model of Timed Petri Nets with discrete prices. In such model, a price is associated to each run of the net. Then, the reachability (coverability) threshold-problem, that of being able to reach (cover) a given final marking with at most a given price, is studied. This study is extended to the continuous case in [17]. In our setting, we require the workflow to behave correctly in any case, without the need of a supervisor, which in the priced setting means that no run reaching the final marking costs too much, as opposed to the threshold problems, in which the existence of one good run is considered.

Quantitative aspects of reactive systems are studied as energy games e.g. in [18–20]. For example, in [20] games are played on finite weighted automata, studying the existence of infinite runs satisfying several properties over the accumulated weights, as ensuring that a resource is always available or does not exceed some bound.

Outline. Sect. 2 gives some notations we will use throughout the paper. Sec. 3 extends wf-nets with prices and proves undecidability of price-safety with negative costs. In Sect. 4 we extend rcwf-nets and give some basic results. In Sect. 5 we study some specific cases of price predicates. Finally, in Sect. 6 we present our conclusions. Missing proofs can be found in [21].

2 Preliminaries

A quasi-order $\leq$ over a set A is a reflexive and transitive binary relation over A. Given a quasi-order $\leq$, we say that $a < b$ if $a \leq b$ and $b \not\leq a$. Given $B \subseteq A$, we denote $B\downarrow = \{a \in A \mid \exists b \in B, a \leq b\}$ the downward closure of B and we say that B is downward-closed if $B\downarrow = B$. Analogously, we define $B\uparrow$, the upward closure of B and say B is upward closed if $B\uparrow = B$. We denote by $\mathbb{N}_\omega = \mathbb{N} \cup \{\omega\}$, the naturals completed with their limit ω and $\mathbf{0} = (0, ..., 0)$. We write $v[i]$ to denote the i^{th} component of $v \in \mathbb{N}_\omega^k$. We denote by $\leq$ the component-wise order in any $\mathbb{N}_\omega^k$ or $\mathbb{Z}^k$, and by $<$ its strict version. A (finite) multiset m over a set A is a mapping $m : A \to \mathbb{N}$ with finite support, that is, such that $supp(m) = \{a \in A \mid m(a) > 0\}$ is finite. We denote by $A^\oplus$ the set of finite multisets over A. For two multisets

m_1 and m_2 over A we define $m_1 + m_2 \in A^{\oplus}$ by $(m_1 + m_2)(a) = m_1(a) + m_2(a)$ and $m_1 \subseteq m_2$ if $m_1(a) \leq m_2(a)$ for every $a \in A$. For a multiset m and $\lambda \in \mathbb{N}$, we take $(\lambda * m)(a) = \lambda * m(a)$. When $m_1 \subseteq m_2$ we can define $m_2 - m_1 \in A^{\oplus}$ by $(m_2 - m_1)(a) = m_2(a) - m_1(a)$. We denote by $\emptyset$ the empty multiset, that is, $\emptyset(a) = 0$ for every $a \in A$, and $|m| = \sum_{a \in supp(m)} m(a)$. We use set notation for multisets when convenient, with repetitions to account for multiplicities greater than one. We write $\{a_1, ..., a_n\} \leq^{\oplus} \{b_1, ..., b_m\}$ if there is an injection $h : \{1, ..., n\} \to \{1, ..., m\}$ such that $a_i \leq b_{h(i)}$ for each $i \in \{1, ..., n\}$.

Petri Nets. A Place/Transition (P/T) net is a tuple $N = (P, T, F)$, where P is a finite set of places, T is a finite set of transitions (disjoint with P) and $F : (P \times T) \cup (T \times P) \to \mathbb{N}$ is the flow function.

A marking of N is an element of $P^{\oplus}$. For a transition t we define ${}^{\bullet}t \in P^{\oplus}$ as ${}^{\bullet}t(p) = F(p, t)$. Analogously, we take $t^{\bullet}(p) = F(t, p)$, ${}^{\bullet}p(t) = F(t, p)$ and $p^{\bullet}(t) = F(p, t)$. A marking m enables a transition $t \in T$ if ${}^{\bullet}t \subseteq m$. In that case t can be fired, reaching the marking $m' = (m - {}^{\bullet}t) + t^{\bullet}$, and we write $m \xrightarrow{t} m'$. A run r is a sequence $m_0 \xrightarrow{t_1} m_1 \xrightarrow{t_2} ... \xrightarrow{t_n} m_n$. If r_1 and r_2 are two runs so that r_1 finishes at the marking in which r_2 starts, we denote by $r_1 \cdot r_2$ the run starting with the transitions in r_1, followed by those in r_2, as expected.

Workflow Petri Nets. We will use the definition in [4]. A workflow Petri net (shortly a wf-net) is a P/T net $N = (P, T, F)$ such that:

- there are $in, out \in P$ with ${}^{\bullet}in = \emptyset$, $in^{\bullet} \neq \emptyset$, ${}^{\bullet}out \neq \emptyset$ and $out^{\bullet} = \emptyset$,
- for each $p \in P \setminus \{in, out\}$, ${}^{\bullet}p \neq \emptyset$ and $p^{\bullet} \neq \emptyset$.

When there is no confusion we will simply refer to the special places given by the previous definition as *in* and *out*, respectively. We denote by m_{in} the marking of N with a single token in *in*, and empty elsewhere. Analogously, m_{out} is the marking of N with a single token in *out* and empty elsewhere. There are several definitions of soundness of wf-nets in the literature. We will use one called weak soundness in [2]. A wf-net is weakly sound if for any marking reachable from m_{in} it is possible to reach m_{out}.

Petri Nets with Dynamic Name Creation and Whole Place Operations. In order to model different instances running in the same net, we will use names, each name representing a different instance. A ν-PN is an extension of Petri nets in which tokens are names and fresh name creation can be performed. We define them here as a subclass of $w\nu$-PNs, a class of nets which we will need in Sect. 5.1, that also allow whole-place operations and broadcasts, similar to Data Nets [13]. Data Nets extend P/T nets by considering a linearly ordered and dense domain of tokens, and in which whole place operations can be performed. Therefore, $w\nu$-PNs can be seen as an unordered version of Data nets [13] in which names can be created fresh. When a transition t of a $w\nu$-PN is fired, four operations are performed: the subtraction of several tokens of different colors,

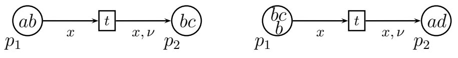


Fig. 1. The firing of a $w\nu$-PN

whole-place operations (affecting every color in the same way), the creation of new names and the addition of tokens.

Let us consider a set Var of variables and $\Upsilon \subset Var$ a set of name creation variables. A $w\nu$-PN is a tuple $N = (P, T, F, G, H)$ where P and T are finite disjoint sets of places and transitions, respectively; for each $t \in T$, $F_t : P \to (Var \backslash \Upsilon)^{\oplus}$ is its subtraction function, $G_t : P \times P \to \mathbb{N}$ is its whole-place operations matrix, and $H_t : P \to Var^{\oplus}$ is its addition function. Moreover, if $x \in H_t(p) \setminus \Upsilon$ then $x \in F_t(p')$ for some $p' \in P$.

Let Id be an infinite set of names. A marking is any $m : P \to Id^{\oplus}$. An a-token in p is an occurrence of $a \in m(p)$. $Id(m)$ is the set of names appearing in m, that is, $Id(m) = \bigcup_{p \in P} supp(m(p))$. We denote by $Var(t) = \{x \in Var \mid \exists p \in P,\ x \in F_t(p) \cup H_t(p)\}$ and $Var(p) = \{x \in Var \mid \exists t \in T,\ x \in F_t(p) \cup H_t(p)\}$. A mode is a mapping $\sigma : Var(t) \to Id$ extended pointwise to $\sigma : Var(t)^{\oplus} \to Id^{\oplus}$. A transition t is enabled at a marking m with mode σ if for all $p \in P$, $\sigma(F_t(p)) \subseteq m(p)$ and for all $\nu \in \Upsilon$, $\sigma(\nu) \notin Id(m)$. Then, we say that t can be fired, reaching a new marking m', where for all $p \in P$, $m'(p) = \sum_{p' \in P}((m(p') - \sigma(F_t(p'))) * G_t(p', p)) + \sigma(H_t(p))$, and we denote this by $m \overset{t(\sigma)}{\to} m'$.

Example 1. Let $N = (\{p_1, p_2\}, \{t\}, F, G, H)$ be a $w\nu$-PN, where:

- $F_t(p_1) = \{x\}$, $F_t(p_2) = \emptyset$.
- $H_t(p_1) = \emptyset$, $H_t(p_2) = \{x, \nu\}$.
- $G_t(p_1, p_1) = 1$, $G_t(p_1, p_2) = 0$, $G_t(p_2, p_1) = 1$, $G_t(p_2, p_2) = 0$.

This net is depicted in Fig 1. Note that although F_t and H_t are represented by arrows labelled by the corresponding variables, the effects of G_t are not depicted.

Let m be the marking of N such that $m(p_1) = \{a, b\}$ and $m(p_2) = \{b, c\}$. Then, t can be fired at m with mode σ, where $\sigma(x) = a$ and $\sigma(\nu) = d$, reaching a new marking m', such that $m'(p_1) = \{b, b, c\}$ and $m'(p_2) = \{a, d\}$. Note that m' is obtained from m by the following steps:

- Removing an a-token from the place p_1, due to the "effect" of F.
- Removing all tokens from p_2 and copying them to p_1, because of G.
- Adding an a-token and a d-token to p_2, because of H.

We write $m_1 \sqsubseteq m_2$ if there is a renaming m'_1 of m_1 such that $m'_1(p) \subseteq m_2(p)$ for every $p \in P$. A marking m is coverable from an initial marking m_0 if we can reach m' from m_0 such that $m \sqsubseteq m'$.

A $w\nu$-PN could be considered as an unordered Data Net, except for the fact that $w\nu$-PNs can create fresh names. In [12] the authors extend Data Nets with fresh name creation and prove that coverability is still decidable by instantiating the framework of Well Structured Transition Systems [22].

Proposition 1. *Coverability is decidable for wν-PN.*

Finally, we define ν-PN [23], which is a fragment of wν-PN without whole-place operations. Formally, a ν-PN is a *wν*-PN in which, for each $t \in T$, G_t is the identity matrix, and we will simply write (P, T, F, H).

In the rest of the paper we will introduce some more models, that will be most of the time priced versions of the models already defined. For the sake of readability, we prefer to present these models in an incremental way, instead of considering a very general model which subsumes all the others.

3 Priced Workflow-Nets

Let us define a priced extension of wf-nets. We follow the cost model in [11]. It essentially amounts to adding to a wf-net two functions, defining the price of the firing of each transition, and the cost of storing tokens during the firing of each transition, respectively.

Definition 1 (Priced workflow net). *A* priced workflow net *(pwf-net) with* price arity $k \geq 0$ *is a tuple $N = (P, T, F, C, S)$ such that:*

- *(P, T, F) is a wf-net, called the underlying wf-net of N,*
- *$C : T \to \mathbb{Z}^k$ is a function assigning firing costs to transitions, and*
- *$S : P \times T \to \mathbb{Z}^k$ is a function assigning storage costs to pairs of places and transitions.*

Notice that costs may be negative. The behavior of a pwf-net is given by its underlying wf-net. In particular, adding prices to a wf-net does not change its behavior, as the costs are not a precondition for any transition. That is the main difference between adding resources and prices. Indeed, firing costs can be seen as resources. However, since storage costs depend not only on the transitions which are fired, but also on the number of tokens in the rest of the places when the transitions are fired, they cannot be seen as resources anymore.

Let us define the price of a transition.

Definition 2 (Price of a run). *Let t be a transition of a pwf-net enabled at a marking m. We define $\mathcal{P}(t, m)$, the* price of the firing of t at m, *as*

$$\mathcal{P}(t, m) = C(t) + \sum_{p \in m - {}^\bullet t} S(p, t)$$

Then, the price of a run $r = m_1 \xrightarrow{t_1} m_2 \xrightarrow{t_2} m_3 \ldots m_n \xrightarrow{t_n} m_{n+1}$ *of a pwf-net is $\mathcal{P}(r) = \sum_{i=1}^{n} \mathcal{P}(t_i, m_i)$.*

Notice that in the definition of $\mathcal{P}(t, m)$ the term $m - {}^\bullet t$ is a multiset, so that if a place p appears twice in it then we are adding $S(p, t)$ twice in turn. It can be seen that firing costs can be simulated by storage costs, though we prefer to keep both to follow the approach in [11]. However, storage costs cannot be simulated by firing costs, since the former are marking dependent, while the latter are not. Next, we define safeness of a pwf-net with respect prices.

Definition 3 (b-p-safeness). *Given $b \in \mathbb{N}_\omega^k$, we say that a pwf-net is* b-p-safe *if for each run r reaching m_{out}, $\mathcal{P}(r) \leq b$.*

Therefore, a pwf-net is b-safe if all the runs that reach the final marking cost less than the given budget. Next, define soundness for pwf-nets.

Definition 4 (b-soundness). *Given $b \in \mathbb{N}_\omega^k$, we say that a pwf-net is* b-sound *if from each marking m, reachable from m_{in} via some run r_1, we can reach m_{out} via some run r_2 such that $\mathcal{P}(r_1 \cdot r_2) \leq b$.*

Intuitively, for a pwf-net to be sound we need to be able to reach the final marking at any point with a price that does not exceed the budget $b \in \mathbb{N}_\omega^k$. It is easy to see that a pwf-net is b-sound iff it is weakly sound and b-p-safe. However, as we prove next, the latter is undecidable.

Proposition 2. *b-p-safeness is undecidable.*

Proof (sketch). We reduce the cost-threshold-reachability problem for PPN with negative costs, which is undecidable [11]. A PPN is a P/T net endowed with storage and firing costs. The cost-threshold-reachability problem consists in, given m_f and $b \in \mathbb{N}_\omega^k$, decide whether there is a run σ with $m_0 \xrightarrow{\sigma} m_f$ such that $C(\sigma) \leq v$. The reduction consists essentially in obtaining the pwf-net as the *inverse* of the PPN, by taking the inverse of each firing and storage cost. Then, if a run σ has price c in the PPN, the simulating run has cost $v - c$, so that it satisfies the cost-threshold-reachability property iff $N \not\models \mathbf{0}$. The details of the proof can be found in [21].

Instead of addressing the problem of b-soundness with non-negative costs directly, we will consider a more general version of the problem, that in which several instances of the workflow execute concurrently, called resource-constrained workflow nets. Then, we will obtain the results regarding pwf-nets as a corollary of the more general problem.

To conclude this section, notice that if a pwf-net is b-p-safe (b-sound) then it is also b'-p-safe (b'-sound) for any $b' > b$, so that the set $\mathcal{B}(N) = \{b \in \mathbb{N}_\omega^k \mid N$ is b-p-safe (b-sound)$\}$ is an upward-closed set. In this situation, we can apply the Valk & Jantzen theorem:

Theorem 1 ([24]). *Let V be an upward-closed set. We can compute a finite basis of V if and only if for each $v \in \mathbb{N}_\omega^k$ we can decide whether $v \downarrow \cap V \neq \emptyset$.*

Therefore, we can compute a finite basis of the set $\mathcal{B}(N)$, i.e., the minimal budgets b for which the pwf-net is b-p-safe (b-sound), provided we can decide b-p-safety (b-soundness) for each $b \in \mathbb{N}_\omega^k$.

4 Priced Resource-Constrained wf-Nets

Let us start by recalling the definition of resource-constrained wf-nets (rcwf-nets). For more details see [5]. The definition we use is equivalent to those in [3, 4], though more convenient for our purposes. We represent each instance in an rcwf-net by means of a different name. Hence, our definition of rcwf-nets is based on ν-PN. A ν-PN can be seen as a collection of P/T nets that can synchronize between them and be created dynamically [25]. We start by defining a subclass of ν-PN, called asynchronous ν-PN, in which each instance can only interact with a special instance (which models resources) that is represented by black tokens. We fix variables $\nu \in \Upsilon$ and $x, \epsilon \in Var \backslash \Upsilon$.

Definition 5 (Asynchronous ν-PN). *An* asynchronous ν-PN *is a ν-PN $N = (P, T, F, H)$ such that:*

- *For each $p \in P$, $Var(p) \subseteq \{x, \nu\}$ or $Var(p) = \{\epsilon\}$.*
- *For each $t \in T$, $Var(t) \subseteq \{\nu, \epsilon\}$ or $Var(t) \subseteq \{x, \epsilon\}$.*

Places $p \in P$ with $Var(p) = \{\epsilon\}$ are called static, and represented in figures by circles in bold. They can only contain (by construction) black tokens, so that they will represent resources, and can only be instantiated by ϵ. Places $p \in P$ with $Var(p) \subseteq \{x, \nu\}$ are called dynamic, and represented by normal circles. They can only contain names (different from the black token), that represent instances, and can only be instantiated by x. We denote P_S and P_D the sets of static and dynamic places respectively, so that $P = P_S \cup P_D$.

Let us introduce some notations we will need in the following definition. Given a ν-PN $N = (P, T, F, H)$ and $x \in Var$ we define the P/T net $N_x = (P, T, F_x)$, where $F_x(p, t) = F_t(p)(x)$ and $F_x(t, p) = H_t(p)(x)$ for each $p \in P$ and $t \in T$. Moreover, for $Q \subseteq P$, by $F|_Q$ we mean each F_t restricted to Q, and analogously for $H|_Q$. Roughly, an rcwf-net is an asynchronous ν-PN that does not create fresh names, and so that its underlying P/T net is a wf-net.

Definition 6 (Resource-constrained workflow nets). *A* resource-constrained workflow net *(rcwf-net) $N = (P, T, F, H)$ is an asynchronous ν-PN such that:*

- *for all $t \in T$, $\nu \notin Var(t)$,*
- *$N_p = (P_D, T, F|_{P_D}, H|_{P_D})_x$ is a wf-net, called the* production net *of N.*

Fig. 2 shows an rcwf-net (for now, disregard the annotations C and S in the figure). In figures we do not label arcs, since they can be inferred (arcs to/from static places are labelled by ϵ, and arcs to/from dynamic places are labelled by x). N_p, the production net of N, is the P/T net obtained by projecting N to its dynamic places.

Intuitively, each instance is given by a name, which is initially in in. Given $m_0 \in P_S^{\oplus}$, for each $j \in \mathbb{N}$ we define the initial marking m_0^j as the marking that contains $m_0(s)$ black tokens in each static place s, j pairwise different names in its place in, and is empty elsewhere. For instance, the marking of the rcwf-net in Fig. 2 is m_0^2, where $m_0 = \{s, s, s\}$. Moreover, for such m_0^j we denote by $\mathcal{M}_{out}^j$ the set of markings in which the same j names are in its place out and every other dynamic place is empty. Now we define the priced version of rcwf-nets, analogously as in Def. 1.

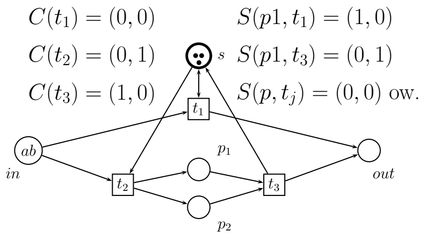


Fig. 2. A priced resource-constrained workflow net

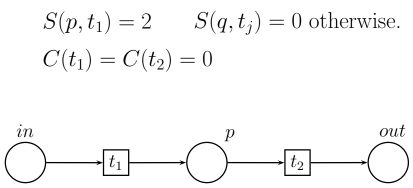


Fig. 3. prcwf-net not *Sum*(b) neither *Max*(b)-dynamically sound for any $b \in \mathbb{N}$

Definition 7 (Priced rcwf-net). *A* priced rcwf-net *(prcwf-net) with* price arity $k \geq 0$ *is a tuple* $N = (P, T, F, H, C, S)$ *such that:*

– (P, T, F, H) *is an rcwf-net, called the underlying rcwf-net of* N*,*
– $C : T \to \mathbb{Z}^k$ *and* $S : P \times T \to \mathbb{Z}^k$ *are functions specifying the firing and storage costs, respectively.*

As for priced wf-nets, the behavior of a priced rcwf-net is given by its underlying rcwf-net. However, its runs have a price associated. We start by defining the price of an instance in a run.

Definition 8 (Price of an instance). *We define the* price of an instance $a \in Id(m_0)$ *in a run* $r = m_0 \xrightarrow{t_1(\sigma_1)} m_1 \xrightarrow{t_2(\sigma_2)} m_2 \ldots m_{n-1} \xrightarrow{t_n(\sigma_n)} m_n$ *of a prcwf-net as*

$$\mathcal{P}(a, r) = \sum_{\substack{i=1 \\ \sigma_i(x)=a}}^{n} (C(t_i) + \sum_{p \in P} |m_{i-1}(p) - \sigma(F_t(p))| * S(p, t_i))$$

Intuitively, we are considering those transitions in r fired by a, and computing its price as we did in Def. 2 for pwf-nets. In particular, we are assuming that when computing the price of the firing of a transition by an instance, the tokens belonging to other instances are accounted for. In other words, a pays a penalization for the storage of all tokens when it fires a transition. We could have also decided that each instance only pays for its own tokens, thus being in a slightly different setting, but the techniques used in our results would also apply.

As we have mentioned before, prices are different from resources in that they do not constraint the behavior of the net. However, once we are interested in checking a priced-soundness problem, it is natural to consider the available "budget" as an extra resource. Indeed, this can be done but only for firing costs, which are local to transitions, but again this is not possible for storage costs.

Since in rcwf-nets we are interested in the behavior of several concurrent instances, we collect their prices in the following definition.

Definition 9 (Price of a run). *Given a run* r *of a prcwf-net starting in* m_0*, we define the* price of r *as the multiset* $\mathcal{P}(r) = \{\mathcal{P}(a, r) \mid a \in Id(m_0)\} \in (\mathbb{Z}^k)^{\oplus}$.

Instead of fixing the condition to be satisfied by all the prices of each instance, we define a parametric version of p-safeness and dynamic soundness. More precisely, those properties for prcwf-nets are parameterized with respect to a price-predicate.

Definition 10 (Price-predicate). *A* price-predicate ϕ *of arity* $k \geq 0$ *is a predicate over* $\mathbb{N}_\omega^k \times (\mathbb{Z}^k)^\oplus$ *such that if* $b \leq b'$ *and* $A' \leq^\oplus A$ *then* $\phi(b, A) \to \phi(b', A')$ *holds.*

Intuitively, b stands for the budget, and A stands for the price of a run. Notice that price-predicates are upward-closed in their first argument, but downward-closed in their second argument. Intuitively, if a price-predicate holds for given budget and costs, then it holds with a greater budget and less costs, as expected. From now on, for a price-predicate ϕ and $b \in \mathbb{N}_\omega^k$, we will denote by $\phi(b)$ the predicate over $(\mathbb{Z}^k)^\oplus$ that results of specializing ϕ with b. Moreover, when there is no confusion we will simply say that a run r satisfies a predicate when $\mathcal{P}(r)$ satisfies it.

We now proceed as in the case of a single instance, defining p-safeness and dynamic soundness, though with respect to a given price-predicate.

Definition 11 (ϕ-p-safeness). *Let* $b \in \mathbb{N}_\omega^k$ *and* ϕ *be a price-predicate. We say that the prcwf-net* N *is* $\phi(b)$-p-safe *for* $m_0 \in P_S^\oplus$ *if for each* $j > 0$*, every run of* N *starting in* m_0^j *satisfies* $\phi(b)$*.*

Definition 12 (ϕ-dynamic soundness). *Let* $b \in \mathbb{N}_\omega^k$ *and* ϕ *be a price-predicate. We say that the prcwf-net* N *is* $\phi(b)$-dynamically sound *for* $m_0 \in P_S^\oplus$ *if for each* $j > 0$ *and for each marking* m *reachable from* m_0^j *by firing some* r_1*, we can reach a marking* $m_f \in \mathcal{M}_{out}^j$ *by firing some* r_2 *such that* $r_1 \cdot r_2$ *satisfies* $\phi(b)$*.*

Ordinary dynamic soundness [5] is obtained by taking ϕ as the constantly true predicate. Let us see some simple facts about ϕ-p-safeness and ϕ-dynamic soundness.

Proposition 3. *The following holds:*

1. *If* $\phi_1 \to \phi_2$ *holds, then* $\phi_1(b)$*-p-safeness implies* $\phi_2(b)$*-p-safeness, and* $\phi_1(b)$*-dynamic soundness implies* $\phi_2(b)$*-dynamic soundness.*
2. *For any* ϕ*,* ϕ*-dynamic soundness implies (unpriced) dynamic soundness.*
3. *In general,* ϕ*-dynamic soundness is undecidable.*

Proof. (1) is straightforward by Def. 11 and Def. 12. (2) follows from (1), considering that any ϕ entails the constantly true predicate. (3) follows from the undecidability of (unpriced) dynamic soundness for general rcwf-nets [5]. □

Therefore, ϕ-dynamic soundness is undecidable for some ϕ, though certainly not for all. As a (not very interesting) example, if ϕ is the constantly false price-predicate, no prcwf-net is ϕ-dynamically sound, so that it is trivially decidable. Now we factorize ϕ-dynamic soundness into unpriced dynamic soundness and p-safety. As we proved in the previous section, if we consider negative costs safeness is undecidable even for priced wf-nets. Therefore, for now on we will focus in rcwf-nets with costs in $\mathbb{N}$.

Proposition 4. *Let ϕ be a price-predicate and N a prcwf-net with non-negative costs. Then N is $\phi(b)$-dynamically sound if and only if it is dynamically sound and $\phi(b)$-p-safe.*

Proof. First notice that for any run r of N and any run r' extending r we have $\phi(b, \mathcal{P}(r \cdot r')) \rightarrow \phi(b, \mathcal{P}(r))$. Indeed, it is enough to consider that, because we are considering that costs are non-negative, $\mathcal{P}(r) \leq^{\oplus} \mathcal{P}(r \cdot r')$ holds and, by Def. 10, ϕ is downward closed in its second parameter. For the if-part, if N is dynamically sound and all its runs satisfy $\phi(b)$ then it is clearly $\phi(b)$-dynamically sound. Conversely, if it is $\phi(b)$-dynamically sound it is dynamically sound by Prop. 3. Assume by contradiction that there is a run r that does not satisfy $\phi(b)$. By the previous observation, no extension of r can satisfy $\phi(b)$, so that N is not $\phi(b)$-dynamically sound, thus reaching a contradiction. $\square$

Therefore, to decide ϕ-dynamic soundness we can consider those two properties separately. Though (unpriced) dynamic soundness is undecidable for general rcwf-nets, it is decidable for a subclass of rcwf-nets that we call proper rcwf-nets [5]. An rcwf-net is proper if its production net is weakly sound and for each transition t in the production net, $t^{\bullet} \neq \emptyset$. The first condition can be intuitively understood as the rcwf-net behaving properly (being weakly sound) if endowed with infinitely-many resources, which amounts to removing the restriction of its behavior by means of resources. The second condition is a slight relaxation of the standard connection condition considered for wf-nets [2]: for any $n \in P \cup T$ there exists a path from *in* to n and from n to *out*. That is because, given a transition $t \in T$, if there exists a path from t to *out*, then there must be $p \in P$ with $p \in ({}^{\bullet}t)$ and therefore $t^{\bullet} \neq \emptyset$.

In turn, it is decidable to check that an rcwf-net is proper [5]. In the following sections, we will study the decidability of $\phi(b)$-p-safeness for various price-predicates, even if N is not proper.

To conclude this section, and as we did in the previous one, notice that for any price-predicate ϕ, the set $\mathcal{B}_{\phi}(N) = \{b \in \mathbb{N}^k_{\omega} \mid N \text{ is } \phi(b)\text{-dynamically sound } (\phi(b)\text{-p-safe})\}$ is upward-closed because of the upward-closure in the first parameter of price-predicates. Therefore, and as we did for pwf-nets, we can apply the Valk & Jantzen result to compute the minimal budgets b for which N is $\phi(b)$-p-safe ($\phi(b)$-dynamically sound) whenever we can decide $\phi(b)$-p-safeness ($\phi(b)$-dynamic soundness) for each $b \in \mathbb{N}^k_{\omega}$.

5 Selected Price Predicates

Now, we study some specific cases of these price predicates. In particular, we study the maximum, the sum, the average and the discounted sum.

5.1 *Sum* and *Max*-Dynamic Soundness

Let us now study the two first of the concrete priced problems for prcwf-nets. When we consider several instances of a workflow net running concurrently, we may be interested in the overall accumulated price, or in the highest price that the execution of each instance may cost.

Definition 13 (*Sum* and *Max* price-predicates). *We define the price-predicates Sum and Max as:*

$$\begin{array}{lcl} Sum(b, A) & \iff & \sum_{x\in A} x \leq b \\ Max(b, A) & \iff & x \leq b \textit{ for all } x \in A \end{array}$$

Sum and *Max* are indeed price-predicates because they satisfy the conditions in Def. 10. They are both upward closed in the first parameter and downward closed in the second. Let us remark that the cost model given by *Sum*, in which all the prices are accumulated, is the analogous to the cost models in [11, 17]. However, since we are here interested in the behavior of an arbitrary number of instances, a necessary condition for $Sum(b)$-p-safeness is the existence of an instance from which the rest of the instances have a null price (for those components in b that are not ω).

Example 2. Consider the prcwf-net N in Fig. 3, and a run of N with n instances, and in which t_2 is not fired until t_1 has been fired n times. The price of the i-th instance in any such run is $2 \cdot (i-1)$. Indeed, the first firing of t_1 costs nothing, because there are no tokens in p, but in the second one there is already a token in p, so that the second firing costs 2 (because $S(p, t_1) = 2$). In particular, the last instance of the net costs $2 \cdot (n-1)$. Therefore, the net is neither $Max(b)$-p-safe nor $Sum(b)$-p-safe for any $b \in \mathbb{N}$.

Now, suppose that $S(p, t) = 0$ for each place p and transition t, $C(t_1) = 1$ and $C(t_2) = 0$. Each instance costs exactly 1, so that it is $Max(1)$-p-safe. However, if we consider a run in which n instances have reached *out*, then the sum of the prices of all instances is n, and the net is not $Sum(b)$-p-safe for any $b \in \mathbb{N}$.

Now we prove decidability of $Max(b)$ and $Sum(b)$-p-safety by reducing them to non-coverability problems in a $w\nu$-PN. Given a prcwf-net N we build a $w\nu$-PN $\mathcal{C}(N)$, the *cost representation net* of N, by adding to N new places, whose tokens represent the costs of each run. Then, the net will be safe iff no marking with $b_i + 1$ tokens in the place representing the i^{th} component of the prices can be covered. More precisely, we will use a-tokens to compute the cost of the instance represented by a. We simulate firing costs by adding to N "normal arcs", without whole-place operations, but for the simulation of storage costs we need the whole-place capabilities of $w\nu$-PN.

Proposition 5. *Max-p-safety and Sum-p-safety are decidable for prcwf-nets. Max-dynamic soundness and Sum-dynamic soundness are decidable for proper prcwf-nets.*

Proof. We reduce *Sum*-p-safety to coverability for $w\nu$-PN. Then, we sketch how to adapt this reduction to the case of *Max*-p-safety. Let $N = (P, T, F, H, C, S)$ be a prcwf-net with price arity k. Let $b \in \mathbb{N}^k_\omega$. We can assume that b has no ω-components, or we could safely remove the cost information of those components. We build the $w\nu$-PN $\mathcal{C}(N) = (P^c, T^c, F^c, G^c, H^c)$ as follows:

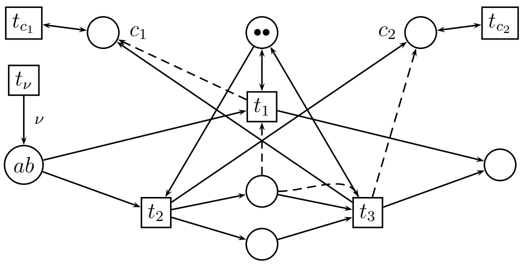


Fig. 4. The costs representation $w\nu$-PN of the prcwf-net in Fig. 2

- $P^c = P \cup \{c_1, ..., c_k\}$,
- $T^c = T \cup \{t_\nu\} \cup \{t_{c_1}, ..., t_{c_k}\}$.
- For each $t \in T$,
  - $F^c_t(p) = F_t(p)$ if $p \in P$, and $F^c_t(p) = \emptyset$, otherwise,
  - $H^c_t(p) = H_t(p)$ if $p \in P$, and $H^c_t(c_i) = C(t)[i] * \{x\}$, otherwise,
  - $G^c_t(p, p') = \begin{cases} S(p,t)[i] & \text{if } p \in P, \text{ and } p' = c_i, \\ 1 & \text{if } p = p', \\ 0 & \text{otherwise.} \end{cases}$
- For each $i \in \{1, ..., k\}$,
  - $F^c_{t_{c_i}}(c_i) = \{x, y\}$, and $F^c_{t_{c_i}}(p) = \emptyset$ otherwise,
  - $H^c_{t_{c_i}}(c_i) = \{x, x\}$, and $H^c_{t_{c_i}}(p) = \emptyset$ otherwise, and
  - $G^c_{t_{c_i}}$ is the identity matrix.
- $F_{t_\nu}(p) = \emptyset$ for any $p \in P^c$, $H_{t_\nu}(in) = \{\nu\}$ and returns the empty multiset elsewhere, and G_{t_ν} is the identity matrix.

Any run r of N can be simulated by a run of $\mathcal{C}(N)$, preceded by several firings of t_ν. Moreover, if r starts in m_0 and finishes in m (seen as a run of $\mathcal{C}(N)$), then by construction of $\mathcal{C}(N)$ it holds that the sum of the prices of the instances in r, is the vector formed by considering the number of tokens (maybe with different colors) in $c_1, ..., c_k$. Finally, as each transition t_{c_i} takes two tokens with different names from c_i, and puts them back, changing the name of one of them by the name of the other token, these transitions allow to reach each markings in which the sum of the prices of all instances of a run is represented by the tokens in the places c_i, all of them with the same name. Then, N is *Sum*(b)-p-unsafe if and only if there is $j \in \{1, ..., k\}$ such that the marking with $b[j] + 1$ tokens of the same color in c_j and empty elsewhere is not coverable, and we are done.

The previous construction with some modifications also yields decidability of *Max*(b)-p-safeness. We add one more place *last* (which will always contain the name of the last instance that has fired a transition) and for each $i \in \{1, ..., k\}$, we add a new place d_i (where we will compute the costs). When a transition $t \in T$ is fired, in $\mathcal{C}(N)$ we replace the name in *last* by the name of the current transition, and reset every place c_i (by setting $G_t(c_i, c_i) = 0$). Moreover, we change the effect of every t_{c_i}: they now take a token from c_i, and put a token of the name in *last* in the place d_i (see Fig. 5).

Therefore, when a transition $t \in T$ is fired, it is possible to reach a marking in which the costs of firing t are added to every d_i (represented by the name of the instance that has fired t) by firing t followed by the firing of every t_{c_i} n_i times, provided t put n_i tokens in c_i. Notice that if another transition fires before, then that run is lossy, in the sense that it is computing an underapproximation of its cost, but it is always possible to compute the exact cost. Therefore, N is $Max(b)$-p-unsafe if and only if there is $j \in \{1, ..., k\}$ such that the marking with $b[j] + 1$ tokens of the same color in d_j and empty elsewhere is not coverable. $\square$

Example 3. Fig. 4 shows the costs representation net of the net N in Fig. 2. For a better readability, we have removed the labels of the arcs. As the prices in N are vectors of $\mathbb{N}^2$, we have added two places, c_1 and c_2, to store the costs; and two transitions t_{c_1} and t_{c_2}, which take two tokens of different colours of the corresponding places and put them back, with the same colour. Moreover, we have added arcs that manage the addition of the cost of transitions. In particular, dashed arcs denote copy arcs, meaning that when the corresponding transition is fired, tokens are copied in the places indicated by the arrows (which is the effect of G in the proof of the previous result). Then, $Sum(b)$-p-safeness is reduced to non-coverability problems: the prcwf-net is $Sum(1, 1)$-p-safe iff neither m_1 (the marking with only two tokens carrying the same name in c_1) neither m_2 (the marking with only two tokens carrying the same name in c_2) are coverable.

We remark that if we consider a cost model in which each instance only pays for its own tokens, as discussed after Def. 8, the previous proof can be adapted by considering a version of $w\nu$-PN with finer whole-place operations, which are still a subclass of the ones considered in [12], so that the result would still apply. To conclude this section, we show that we can reduce the soundness problem for pwf-nets defined in Sect. 3 to *Max*-dynamic soundness for prcwf-nets.

Corollary 1. *b-p-safeness and b-soundness are decidable for pwf-nets with non-negative costs.*

Proof. Let N be a pwf-net. To decide b-safety it is enough to build a prcwf-net N' by adding to N a single static place s, initially containing one token, two new places in' and out' (the new initial and final places), and two new transitions t_{in} and t_{out}. Transition t_{in} can move a name from in' to in whenever there is a token in s, that is, $F_{t_{in}}(in') = \{x\}$, $F_{t_{in}}(s) = \{\epsilon\}$ and $F_{t_{in}}$ is empty elsewhere, and $H_{t_{in}}(in) = \{x\}$ and empty elsewhere. Analogously, t_{out} can move a name from out to out', putting the black token back in s, that is, $F_{t_{out}}(out) = \{x\}$, and empty elsewhere, and $H_{t_{out}}(out') = \{x\}$, $H_{t_{out}}(s) = \{\epsilon\}$, and empty elsewhere. In this way, the concurrent executions of N' are actually sequential. Since there

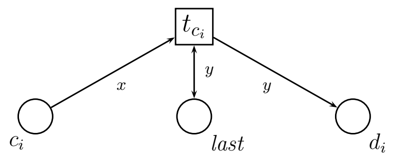


Fig. 5. The mechanism added for *Max*

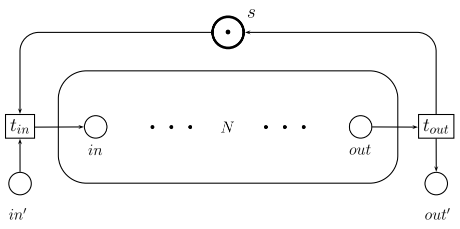


Fig. 6. The construction of Cor. 1

is no other way in which instances can synchronize with each other (because there are no more static places) the potential behavior of all instances coincide, and coincide in turn with the behavior of N. Finally, we take the cost of firing t_{in} and t_{out} as null, as well as the cost of storing tokens in in' and out' for any transition, and the cost of storing tokens in any place for t_{in} and t_{out}. More precisely, $C(t_{in}) = C(t_{out}) = \mathbf{0}$, $S(p, t_{in}) = S(p, t_{out}) = \mathbf{0}$ for any $p \in P$, and $S(in', t) = S(out', t) = \mathbf{0}$ for any $t \in T$. In this way, the cost of each instance is the cost of a run of N. Therefore, N is b-p-safe if and only if N' is $Max(b)$-p-safe. Since weak soundness is decidable for wf-nets [2], we conclude. □

5.2 *Av*-Dynamic Soundness

Now we study the next of the concrete priced-soundness problems. Instead of demanding that the execution of each instance does not exceed a given budget (though the price of one instance depends on the others), we will consider an amortized, or average price.

Definition 14 (*Av* price-predicate). *We define the price-predicate Av as* $Av(b, A) \Leftrightarrow (\sum_{x \in A} x)/|A| \leq b$.

Therefore, N is $Av(b)$-p-safe if in average, the price of each instance does not exceed b, for any number of instances. Alternatively, we could have a slightly more general definition, in which we only considered situations in which the number of instances exceeds a given threshold $l > 0$. More precisely: $Av_l(b, A) \Leftrightarrow |A| \geq l \rightarrow (\sum_{x \in A} x)/|A| \leq b$. We will work with Av, though we claim that with fairly minor changes in our techniques we could also address the slightly more general price-predicate Av_l.

Example 4. Consider the prcwf-net in Fig. 9. The cost of firing t_1 is twice the number of instances in place in when t_1 is fired. Therefore, the net is $Av(2)$-p-safe, though it is not $Max(b)$-p-safe for any $b \in \mathbb{N}$.

Now suppose that we force t_1 to be fired in the first place by adding some static conditions. Then, though the net is not $Av(0)$-p-safe, it is $Av_l(0)$-p-safe if we consider any threshold $l \geq 2$.

We can reduce $Av(b)$-dynamic soundness of a prcwf-net N to (unpriced) dynamic soundness of an rcwf-net N^b. In order to ensure $Av(b)$-p-safeness, the maximum budget we may spend in an execution with n instances is $b * n$. Essentially, the

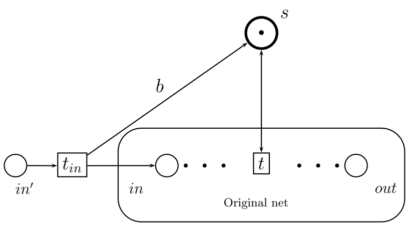


Fig. 7. Construction for $Av(b)$-dynamic soundness

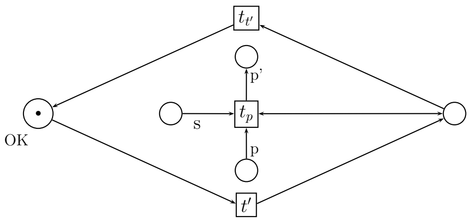


Fig. 8. Schema of the managing of storage costs assuming $S(p, t) = 1$

idea of this construction is to add to N new places s in which tokens represent the remaining budget, and remove tokens from them when transitions are fired. Moreover, each transition will have s as a precondition, so that if the net has consumed all the budget, then it halts before reaching the final marking. Therefore, we add b tokens to s each time an instance starts its execution. The simulation is "lossy" because of how we manage storage costs, but it preserves dynamic soundness. The proof of the next proposition gives a detailed explanation of this construction.

Proposition 6. *Given a prcwf-net N and $b \in \mathbb{N}^k_\omega$, there is an rcwf-net N^b such that N is $Av(b)$-dynamically sound if and only if N^b is dynamically sound.*

Proof. Let k be the price arity of N. We start the construction of N^b by adding to N new static places $s_1, ..., s_k$ that initially contain one token each. These new places store the budget than can be consumed by instances, plus the initial extra token. In order to obtain that, every instance adds $b[i]$ tokens to s_i when it starts. When a transition t is fired, we remove from s_i $C(t)[i]$ tokens to cope with firing costs. We will later explain how to cope with storage costs (notice that N^b is an rcwf-net, and in particular it does not have whole-place operations). Moreover, each transition has $s_1, ..., s_k$ as preconditions and postconditions. Therefore, the net will deadlock when some s_i is empty, meaning that it has used strictly more than the allowed budget. Then, if N is not $Av(b)$-dynamically sound, N^b halts before reaching the final marking for some execution, and therefore, it is not dynamically sound. Moreover, if N is $Av(b)$-dynamically sound, then N^b is dynamically sound, because each place s_i always contains tokens, and therefore the executions of N^b represent executions of N. Fig. 7 shows a schema of the reduction for price arity 1.

Now we address the simulation of storage costs. Fig. 8 depicts the following construction. We simulate them in a "lossy" way, meaning that if the firing of t in N costs v, in the simulation we will remove *at most* $v[i]$ tokens from s_i. To do that, for each place p of N we will add a new place p'.

When a transition t is fired, for each place p we transfer tokens from p to p', one at a time (transition t_p in the figure), removing at each time $S(p, t)[i]$ tokens from s_i. We add the same mechanism for the transfer of tokens from p'

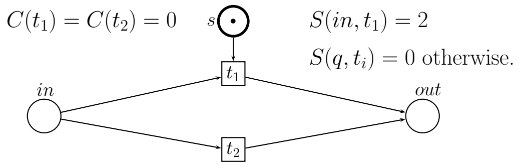


Fig. 9. *Av*-p-safety does not imply *Max*-p-safety

to p. At any point, the transfer can stop (even if some tokens have not been transfered), which finishes the simulation of t. Since we now have two places representing each place p (p and p'), for each transition of N, we need to add several transitions in order to be able to take (or put) tokens from p, p' or both.

Having lossy computations of the cost of a run, if N exceeds the average budget for some execution and some number of instances, then N^b will have a deadlock when this execution is simulated correctly (meaning that all the tokens which have to be transfered are indeed transfered). Then, N^b is not dynamically sound. Conversely, if N is $Av(b)$-dynamically sound (and in particular no run of N exceeds the average budget), then N^b never consumes all the tokens in any s_i, and it behaves as N, so that it is dynamically sound. □

Corollary 2. *Av-dynamic soundness is decidable for proper prcwf-nets.*

5.3 Ordered Prices

So far, we have considered that instances are not ordered in any way, following directly the approaches in [3–5]. Nevertheless, we could consider an order between the instances, and use it to compute the price of a run in such a way that the relative order between instances matter. A sensible way to do that is to assume a linear order between instances within a run given by the order in which they start their execution.

Definition 15 (Order between instances). *Let N be a prcwf-net, and a and b be two instances in a run r of N. We write $a <_r b$ if a is removed from in in r before b, and $a =_r b$ if neither a nor b have been removed from in in r. We write $a \leq_r b$ if $a =_r b$ or $a <_r b$.*

Then, the order $\leq_r$ is a total order over the set of instances in r. In this situation we can write $Id(r) = a_1 \leq_r \cdots \leq_r a_n$ to denote that $a_1, ..., a_n$ are all the instances in r, ordered as indicated. In the following, for a set A we denote by A^* the set of finite words over A.

Definition 16 (Ordered price of a run). *Given a run r of a prcwf-net with $Id(r) = a_1 \leq_r \cdots \leq_r a_n$ we define the* ordered price *of r as the word $\mathcal{P}_o(r) = \mathcal{P}(a_1, r)...\mathcal{P}(a_n, r) \in (\mathbb{N}^k)^*$.*

Notice that the previous definition is correct in the sense that whenever $a =_r b$ then we have $\mathcal{P}(a, r) = \mathcal{P}(b, r) = 0$. Moreover, the instances of a run are always ordered as $a_1 <_r \cdots <_r a_m < a_{m+1} =_r ... =_r a_n$.

Table 1. A ✓ symbol in row ϕ_1 and column ϕ_2 means that $\phi_1(b)$-dynamic soundness implies $\phi_2(b)$-dynamic soundness; a ☑ symbol means that the implication holds possibly for a different b; a $\times$ means that the implication does not hold

	Sum	**Max**	**Ds**	**Av**
Sum	✓	✓	✓	✓
Max	$\times$ (Ex. 2)	✓ [21]	☑ [21]	✓ [21]
Ds	$\times$ (Fig. 3)	$\times$ (Fig. 3)	✓	$\times$ (Fig. 3)
Av	$\times$ (Fig. 9)	$\times$ (Fig. 9)	$\times$ (Fig. 9)	✓

With the notion of ordered price, we can consider price-predicates that depend on the order in which instances are fired. Therefore, ordered price-predicates are predicates over $\mathbb{N}^k_\omega \times (\mathbb{N}^k)^*$. We consider the order $\leq^*$ over $(\mathbb{N}^k)^*$ given by $w_1...w_n \leq^* w_1...w'_m$ iff $n < m$ and for each $0 < i \leq n$, $w_i \leq w'_i$. For instance, following [14], we can model situations in which costs in the future are less important than closer ones.

Definition 17 (*Ds*-price predicate). *Given* $0 < \lambda < 1$, *we define the* discounted-sum *price-predicate* Ds_λ *as* $Ds_\lambda(b, v_1...v_n) \Leftrightarrow \sum_{i=1}^{n} \lambda^i \cdot v_i \leq b$.

Example 5. Let us recall the run of the net N of Fig. 3 described in Ex. 2. We proved that the net is neither $Max(b)$-p-safe nor $Sum(b)$-p-safe for any $b \in \mathbb{N}$. Moreover, the average price of the run is $\sum_{i=1}^{n} 2(i-1)/n$, which equals $n-1$, so that it is not $Av(b)$-p-safe for any $b \in \mathbb{N}$. However, the discounted price of the run is $\sum_{i=1}^{n} 2(i-1)\lambda^i$, with $0 < \lambda < 1$. By using standard techniques, it can be seen that the limit of those sums is $b = 2\lambda^2/(1-\lambda)^2$. Moreover, for $\lambda = 1/c$ with $c > 1$ that formula simplifies to $2/(c-1)^2$. As it is easy to prove that the considered runs are the most expensive ones of N, it follows that it is $Ds_\lambda(b)$-p-safe for that $b \in \mathbb{N}$.

Note that if we consider $\leq^*$, then Ds_λ is downward-closed in its second argument. Decidability of Ds_λ-p-safeness remains open, but a weaker version of this problem, in which we only consider finitely many instances, is decidable.

Definition 18 (*Fds*-price predicate). *Given* $0 < \lambda < 1$ *and* $k \in \mathbb{N}$, *we define the finite-discounted-sum price-predicate* $Fds^k_\lambda(b, v_1...v_n) \Leftrightarrow \sum_{i=1}^{min\{n,k\}} \lambda^i \cdot v_i \leq b$.

For this finite version of discounted-sum, p-safety is decidable.

Proposition 7. *Let* $k \in \mathbb{N}$, $c \in \mathbb{N} \setminus \{0\}$ *and* $\lambda = 1/c$. Fds^k_λ*-p-safety is decidable for prcwf-nets.* Fds^k_λ*-dynamic soundness is decidable for proper prcwf-nets.*

6 Conclusions and Open Problems

We have extended the study of workflow processes, adding prices to them. In particular, we have added firing and storage costs to wf-nets and rcwf-nets, as done for priced Petri nets in [11]. Then, we have defined priced versions of safety and soundness for pwf-nets, and several notions of the same properties for rcwf-nets, depending on how we aggregate local prices to obtain a global price. Table 1 shows the relations between the different predicates. Some of its results are either trivial or proved in [21].

We have proved that b-p-safety is undecidable when negative costs are considered, but decidable for non-negative costs. Moreover, b-soundness is also decidable in this case. Regarding prcwf-nets, we have proved that *Sum*, *Max*, and *Fds*-p-safety are decidable, and *Sum*, *Max*, *Av* and *Fds*-dynamic soundness are decidable for the subclass of proper prcwf-nets.

There are interesting open problems that remain open: decidability of *Av*-p-safety and that of the problems related to the discounted sum. Their study would be a good starting point for the study of more sophisticated aggregation techniques, like the Gini or the Theil index.

There are several ways in which we can extend this work. It would be interesting to consider that storage costs depend on how long tokens stay on places during the transitions. For this purpose, time for rcwf-nets should be considered instead of arbitrary interleavings in the firing of transitions, as done in [17]. The, priced safety and soundness properties could be studied in this timed model.

Moreover, it would be interesting to study the complexity of the problems studied here. It is easy to see that coverability for ν-PN (which has a non-primitive recursive complexity [23]) can be reduced to *Sum* and *Max* safety, so that they are non-primitive recursive. Further research is needed to investigate the complexity for the remaining predicates.

Finally, the study of *Ds*-soundness, leads us to several interesting questions about how the size of the markings and prices of a (sound) rcwf-net may grow. In this sense, we would be interested in studying possible bounds for the number of tokens in places, or for the costs of an instance in terms of the number of instances running in the net.

References

1. van der Aalst, W.M.P., van Hee, K.M.: Workflow Management: Models, Methods, and Systems. MIT Press (2002)
2. van der Aalst, W.M.P., van Hee, K.M., ter Hofstede, A.H.M., Sidorova, N., Verbeek, H.M.W., Voorhoeve, M., Wynn, M.T.: Soundness of workflow nets: classification, decidability, and analysis. Formal Asp. Comput. 23, 333–363 (2011)
3. van Hee, K.M., Serebrenik, A., Sidorova, N., Voorhoeve, M.: Soundness of Resource-Constrained Workflow Nets. In: Ciardo, G., Darondeau, P. (eds.) ICATPN 2005. LNCS, vol. 3536, pp. 250–267. Springer, Heidelberg (2005)
4. Juhás, G., Kazlov, I., Juhásová, A.: Instance Deadlock: A Mystery behind Frozen Programs. In: Lilius, J., Penczek, W. (eds.) PETRI NETS 2010. LNCS, vol. 6128, pp. 1–17. Springer, Heidelberg (2010)
5. Martos-Salgado, M., Rosa-Velardo, F.: Dynamic Soundness in Resource-Constrained Workflow Nets. In: Bruni, R., Dingel, J. (eds.) FORTE/FMOODS 2011. LNCS, vol. 6722, pp. 259–273. Springer, Heidelberg (2011)

6. Sampath, P., Wirsing, M.: Computing the Cost of Business Processes. In: Yang, J., Ginige, A., Mayr, H.C., Kutsche, R.-D. (eds.) UNISCON 2009. LNBIP, vol. 20, pp. 178–183. Springer, Heidelberg (2009)
7. Magnani, M., Montesi, D.: BPMN: How Much Does It Cost? An Incremental Approach. In: Alonso, G., Dadam, P., Rosemann, M. (eds.) BPM 2007. LNCS, vol. 4714, pp. 80–87. Springer, Heidelberg (2007)
8. Tahamtan, A., Oesterle, C., Tjoa, A.M., Hameurlain, A.: Bpel-time - ws-bpel time management extension. In: Zhang, R., Cordeiro, J., Li, X., Zhang, Z., Zhang, J. (eds.) ICEIS (3), pp. 34–45. SciTePress (2011)
9. Liu, K., Jin, H., Chen, J., Liu, X., Yuan, D., Yang, Y.: A compromised-time-cost scheduling algorithm in swindew-c for instance-intensive cost-constrained workflows on a cloud computing platform. IJHPCA 24, 445–456 (2010)
10. Mukherjee, D.: QoS in WS-BPEL Processes. PhD thesis, Indian Institute Of Technology (2008)
11. Abdulla, P.A., Mayr, R.: Minimal Cost Reachability/Coverability in Priced Timed Petri Nets. In: de Alfaro, L. (ed.) FOSSACS 2009. LNCS, vol. 5504, pp. 348–363. Springer, Heidelberg (2009)
12. Abdulla, P.A., Delzanno, G., Begin, L.V.: A classification of the expressive power of well-structured transition systems. Inf. Comput. 209, 248–279 (2011)
13. Lazic, R., Newcomb, T., Ouaknine, J., Roscoe, A.W., Worrell, J.: Nets with tokens which carry data. Fundam. Inform. 88, 251–274 (2008)
14. Chatterjee, K., Doyen, L., Henzinger, T.A.: Quantitative languages. ACM Trans. Comput. Log. 11 (2010)
15. Alur, R., Torre, S.L., Pappas, G.J.: Optimal paths in weighted timed automata. Theor. Comput. Sci. 318, 297–322 (2004)
16. Bouyer, P., Brihaye, T., Bruyère, V., Raskin, J.F.: On the optimal reachability problem of weighted timed automata. Formal Methods in System Design 31, 135–175 (2007)
17. Abdulla, P.A., Mayr, R.: Computing optimal coverability costs in priced timed petri nets. In: LICS, pp. 399–408. IEEE Computer Society (2011)
18. Bouyer, P., Fahrenberg, U., Larsen, K.G., Markey, N.: Timed automata with observers under energy constraints. In: Johansson, K.H., Yi, W. (eds.) HSCC, pp. 61–70. ACM (2010)
19. Chatterjee, K., Doyen, L., Henzinger, T.A., Raskin, J.-F.: Generalized mean-payoff and energy games. In: Lodaya, K., Mahajan, M. (eds.) FSTTCS. LIPIcs, vol. 8, pp. 505–516. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2010)
20. Fahrenberg, U., Juhl, L., Larsen, K.G., Srba, J.: Energy Games in Multiweighted Automata. In: Cerone, A., Pihlajasaari, P. (eds.) ICTAC 2011. LNCS, vol. 6916, pp. 95–115. Springer, Heidelberg (2011)
21. Martos-Salgado, M., Rosa-Velardo, F.: Cost soundness for priced resource-constrained workflow nets (2012), http://antares.sip.ucm.es/frosa/
22. Finkel, A., Schnoebelen, P.: Well-structured transition systems everywhere! Theor. Comput. Sci. 256, 63–92 (2001)
23. Rosa-Velardo, F., de Frutos-Escrig, D.: Decidability and complexity of petri nets with unordered data. Theor. Comput. Sci. 412, 4439–4451 (2011)
24. Valk, R., Jantzen, M.: The residue of vector sets with applications to decidability problems in petri nets. Acta Inf. 21, 643–674 (1985)
25. Rosa-Velardo, F., de Frutos-Escrig, D.: Name creation vs. replication in petri net systems. Fundam. Inform. 88, 329–356 (2008)

On the α-Reconstructibility of Workflow Nets

Eric Badouel

Inria Rennes-Bretagne Atlantique
Campus Universitaire de Beaulieu
F35042 Rennes Cedex, France
eric.badouel@inria.fr

Abstract. The process mining algorithm α was introduced by van der Aalst *et al.* for the discovery of workflow nets from event logs. This algorithm was presented in the context of structured workflow nets even though it was recognized that more wokflow nets should be reconstructible. In this paper we assess α algorithm and provide a more precise description of the class of workflow nets reconstructible by α.

Keywords: Process Mining, Workflows, Net Synthesis.

1 Introduction

One of the purpose of *process mining* [11] is to construct or to reconstruct from an event log a business process model that can generate this event log. The game is to dig out of event logs sufficient informations on the structure of their generating model. As a technique for *model discovery*, process mining has some connections with machine learning.

Process mining may be used for the purpose of modelling. For instance, after collecting over a long period of time information on the health history of many patients, including the diagnosis and treatment steps, one may want to extract from this record an accurate model of the workflow system of an hospital. *Reverse engineering*, which consists of reconstructing from representative use cases an existing but partially unknown system, is another activity of model discovery that can be achieved by process mining. According to [11], process mining can also be used for *conformance* checking or *enhancement* of business process models. For instance, *process-aware systems* record run-time informations used to detect discrepancies between expected and actual behaviours and to refactor these systems.

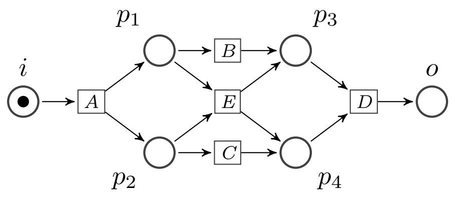
In this paper we focus our attention on model discovery. We fix a subclass of net systems, the so-called *workflow nets*, as the class of target models for process mining. Section 2 introduces the model of workflow nets and we present and illustrate algorithm α [12, 13]. The mining algorithm α tries to reconstruct the places of a workflow net by taking solely into account those pairs of transitions that follow each other in some execution sequence. This assumption has strong impacts, considered in Section 4, on the class of reconstructible workflow nets. In particular, we point out that their inner places are *boundary* places, a condition that we study in more details in Section 5. On that basis we obtain a characterization of the class of workflow nets that are discovered by α. This class contains the structured workflow nets of [12, 13].

2 Discovering Workflow Nets from Event Logs

A log is a finite set of execution sequences of a workflow system. Given a log of the system to be discovered (i.e., constructed or reconstructed), each event reported in this log refers to an *activity*, i.e., a particular step in the workflow system, and to a specific *case*, i.e., a process instance. Additional informations pertaining to events are generally included, for instance a time stamp or the identity of the performer. Usually cases have little or no connection with one another, and time stamps will mainly serve the purpose of indicating the correct ordering of the activities. An event log may then be abstracted to a *set of sequences of activities.* Each sequence represents all activities of a case from the time when it enters the system till the time when it leaves the system.

As the target representation of process mining, we consider a subclass of elementary net systems, called *workflow nets.* The goal of process mining is to synthesize from an event log a workflow net that can reproduce all sequences of activities traced in this log, from the inception of a case to its termination.

Example 2.1. *The workflow net displayed below specifies all possible behaviours of an isolated case in some workflow system. In other words, the firing sequences of the workflow net represent all activities pertaining to a case from the time when the case enters into the system (the input place i is marked) until the case terminates and exits from the system (the output place o is marked). The possible sequences of activities are thus $ABCD, ACBD, AED$.*



A workflow net contains an *input* place i and an *output* place o. The input place i is initially marked, and this initial marking represents the entry of a new case in the system. The output place o gets marked when the case comes to completion, and this final marking represents the exit of the case from the system. The current marking of the workflow net represents the current status of a case. It is assumed that the execution of a case can always reach termination, and that it cannot interfere with the execution of any subsequent case. The latter property, called *soundness* in [13], can be formalized as follows: when the output place is marked, all other places must be empty. The marking in which the output place and no other place is marked is called the *terminal marking.* Therefore, in a (sound) workflow net, the terminal marking must be reachable from any other reachable marking.

Moving the token from the output place back to the initial place is a way to simulate the termination of a case and the inception of a new case. A workflow net with such an implicit feedback may be seen as a cyclic generator, that can iterate in sequence all scenarios of execution of all cases. Instead of adding a feedback transition from the output place to the input place, one might as well coalesce the input place and the output place (confusing thus the initial and terminal markings). With this representation, the two crucial properties of workflow nets N may be reformulated equivalently as follows: *(i)* the net N' obtained by coalescing the input place and the output place of N is *reversible* (the initial marking may be reached from any reachable marking) and *(ii)* the initial (or terminal) marking is the only reachable marking of N' containing the input (or output) place. This is essentially the definition of workflow nets which we adopt below.

Definition 2.2. *A* workflow net *is a contact-free, initially life, place simple and connected elementary net system $N = (P, T, F, M_0)$ where P contains an input place i and an output place o (the remaining places $p \in P \setminus \{i, o\}$ are called* inner places*), such that the following conditions hold:*

1. $^\bullet i = o^\bullet = \emptyset$
2. $(\forall p \in P \setminus \{i, o\}) \quad {}^\bullet p \neq \emptyset \quad \wedge \quad p^\bullet \neq \emptyset$
3. $M_0 = \{i\}$.
4. *The* closed *net system $N' = (P', T, F', \{\iota\})$ obtained from N by replacing places i and o with a unique place ι such that $^\bullet\iota = {}^\bullet o$ and $\iota^\bullet = i^\bullet$ is reversible, and its initial marking $M'_0 = \{\iota\}$ is the unique reachable marking of N' in which the place ι is marked.*

Let us recall that a net system is initially life if any transition is enabled in some reachable marking, and it is life if, for any transition t and for any reachable marking M, the transition t is enabled in some marking reachable from M. Notice that the closed net system N' being both initially life and reversible is life. Let us also recall that a *contact situation* for an elementary net system is given by an accessible marking M and a transition t such that $^\bullet t \subseteq M$ and $M \cap t^\bullet \neq \emptyset$, i.e., transition t is disabled in marking M because of one of its output places. An elementary net is *contact-free* if it has no contact situation. i.e., $M \cap t^\bullet = \emptyset$ for every accessible marking M and transition t such that $^\bullet t \subseteq M$. Any elementary net system is equivalent to a contact-free net obtained by adding complementary places where needed[1]. Contact-free elementary net systems are equivalent to one-safe net systems and, up to this correspondence, Def. 2.2 is an equivalent reformulation of the definition of *sound workflow nets* given in [13].

[1] Places p and $\overline{p}$ are *complementary* places when $^\bullet\overline{p} = p^\bullet$, $\overline{p}^\bullet = {}^\bullet p$, and $p \in M_0 \Leftrightarrow \overline{p} \notin M_0$, then for every marking M accessible (from the initial marking M_0) one has $p \in M \Leftrightarrow \overline{p} \notin M$. If a place has no complementary place one can formally add one such place to the net system without modifying its behaviour (both net systems have isomorphic reachability graph). One can then eliminate the contact situations by adding complementary places to those (output) places involved in some contact situation.

Given a workflow net $N = (P, T, F, M_0)$, the *full log* of N is the language $\mathcal{L}(N) = \{u \in T^* \mid \{i\} [u\rangle \{o\}\}$, i.e., the full log of N is the set of all firing sequences from the initial marking $\{i\}$ to the final marking $\{o\}$. A *log* of N is any subset $W \subseteq \mathcal{L}(N)$ such that every transition $t \in T$ occurs in at least one execution sequence in W. A *workflow log* is any log of a workflow net. Since workflow nets have no dead transitions, the full log of a workflow net is actually a log of this workflow net.

Example 2.3 (Exple. 2.1 continued). *Consider the log $\{ABCD, ACBD, AED\} \subseteq \mathcal{L}(N)$ of the workflow net N shown in Exple. 2.1. Every execution sequence in this log starts with event A and ends with event D. In between, one is left the choice to perform either the event E or the events B and C, which are concurrent since they occur in the log in both orders BC and CB. Using these structural informations extracted from the log, the α algorithm can reconstruct the workflow net N from the considered log. All three execution sequences in this log are actually needed by algorithm α: every activity ought to be reported in at least one execution sequence(thus AED is needed), and for any pair of concurrent events, at least two execution sequences exhibiting the two possible orderings are needed (thus $ABCD$ and $ACBD$ are needed).*

The set of all execution sequences of a workflow net may grow exponentially with the number of events, owing to their possible concurrency. Therefore the execution sequences reported in an event log usually form a small but hopefully representative set of samples of all possible behaviours. In order to discover a workflow net N from some small log $W \subset \mathcal{L}(N)$, a process mining algorithm must carry out some non-trivial generalization over the execution sequences in this log. For that purpose algorithm α is presented as the composition $\alpha = Syn \circ Abs$ of an abstraction function Abs and a synthesis function Syn. The role of the function Abs is to extract from a log W the relevant relations between the events that occur in this log. The role of the function Syn is to reflect, as faithfully as possible, these relations in the structure of a synthesized net system.

Definition 2.4. *The α-abstraction of a workflow log $W \subset T^*$ is the triple $Abs(W) = \langle I_W, C_W, O_W \rangle$ where:*

1. *$I_W = \{t \in T \mid \exists u \in T^* \quad t \cdot u \in W\}$ is the set of transitions starting some execution sequence in the log;*
2. *$O_W = \{t \in T \mid \exists u \in T^* \quad u \cdot t \in W\}$ is the set of transitions ending some execution sequence in the log;*
3. *$C_W = \left\{t \cdot t' \in T^2 \mid \exists u, v \in T^* \quad u \cdot t \cdot t' \cdot v \in W\right\}$ is the set of pairs of transitions appearing consecutively in some execution sequence in the log.*

Definition 2.5. *$W \subseteq \mathcal{L}(N)$ is said to be a* complete *log of workflow net N when $Abs(W) = Abs(\mathcal{L}(N))$*

Hence a complete log W contains already all the information needed by algorithm α: the net systems $\alpha(W)$ and $\alpha(\mathcal{L}(N))$ are identical up to a bijective renaming of their set of places. This relation, called *isomorphism*, will be denoted by $\cong$.

Hence a workflow net is α-reconstructible (i.e., $N \cong \alpha(\mathcal{L}(N))$) if and only if it can be discovered from any of its complete log $W \subseteq \mathcal{L}(N)$ (i.e., $N \cong \alpha(W)$).

From the theory of event structures [14, 15], we know that the behaviour of a net system can be captured, up to net unfolding, by the basic relations of *causality*, *conflict* and *concurrency* between events. When unfolding a net to an event structure, the events are not in bijective correspondence with the transitions of the net, since two occurrences of the same transition may be distinguished by their past history. Nevertheless, given a workflow net N with set of transitions T and a log W of N, one may derive from the α-abstraction of this log three relations $\rightarrow_W$, $\sharp_W$, $\|_W$ between the transitions of N (the activities reported in the log), approximating loosely the relations of causality, conflict and concurrency in the associated event structure. These relations are the following:

$$\begin{array}{lrcl}
\text{causality:} & t \rightarrow_W t' & \Leftrightarrow & t \cdot t' \in C_W \ \wedge \ t' \cdot t \notin C_W \\
\text{conflict:} & t \,\sharp_W\, t' & \Leftrightarrow & t \cdot t' \notin C_W \ \wedge \ t' \cdot t \notin C_W \\
\text{concurrency:} & t \,\|_W\, t' & \Leftrightarrow & t \cdot t' \in C_W \ \wedge t' \cdot \ t \in C_W
\end{array}$$

From these relations between transitions, one may derive again the following relations between sets of transitions.

Definition 2.6. *Let $W \subseteq T^*$ be a workflow log. For any sets of transitions $A, B \subseteq T$, let $A \prec_W B$ when the following three conditions hold:*

1. $(\forall a \in A)(\forall b \in B) \quad a \rightarrow_W b$,
2. $(\forall a_1, a_2 \in A) \quad a_1 \sharp_W a_2$, *and*
3. $(\forall b_1, b_2 \in B) \quad b_1 \sharp_W b_2$

Let $A \prec_W^m B$ when A and B are maximal sets with the property $A \prec_W B$, i.e., $A \prec_W^m B \Leftrightarrow (A \prec_W B) \wedge (A' \prec_W B' \wedge A \subseteq A' \wedge B \subseteq B' \Rightarrow A = A' \wedge B = B')$.

Note that the definition of $\prec_W$ may be applied to any workflow log, and in particular to the full log $\mathcal{L}(N)$ of a workflow net N. Using the above relations, the synthesis function *Syn* used in the α algorithm may be defined as follows.

Definition 2.7. *Let $\langle I_W, C_W, O_W \rangle$ be the α-abstraction of a workflow log $W \subset T^*$. Then $\alpha(W) = Syn(\langle I_W, C_W, O_W \rangle)$ is the elementary net system (P, T, F, M_0) defined as follows:*

1. $P = \{i, o\} \cup \{p_{A,B} \mid A, B \subseteq T \wedge A \prec_W^m B\}$,
2. ${}^\bullet i = \emptyset$, *and* $i^\bullet = I_W$,
3. ${}^\bullet o = O_W$, *and* $o^\bullet = \emptyset$,
4. ${}^\bullet p_{A,B} = A$, *and* ${p_{A,B}}^\bullet = B$,
5. $M_0 = \{i\}$.

Example 2.8. *The algorithm α reconstructs the net system of Exple. 2.1 from its set of firing sequences $W = \{ABCD, ACBD, AED\}$. We have $I_W = \{A\}$, $O_W = \{D\}$, and $C_W = \{AB, BC, CD, AC, CB, BD, AE, ED\}$. For any pair of transitions t and t' one of the following exclusive conditions holds: $t \rightarrow_W t'$, $t' \rightarrow_W t$, $t \sharp_W t'$, or $t \|_W t'$. In the given example we obtain the following classification:*

	B	C	D	E
A	$A \rightarrow_W B$	$A \rightarrow_W C$	$A \sharp_W D$	$A \rightarrow_W E$
B		$B \Vert_W C$	$B \rightarrow_W D$	$B \sharp_W E$
C			$C \rightarrow_W D$	$C \sharp_W E$
D				$E \rightarrow_W D$

The maximal elements of $\prec_W$ are obtained from the smallest ones, namely $\{t\} \prec_W \{t'\}$ for $t \rightarrow_W t'$, by progressively adjoining new elements to each of these two sets while preserving the conditions imposed on $\prec_W$. We obtain:

$$\{A\} \prec_W^m \{B, E\} \qquad \{A\} \prec_W^m \{C, E\} \qquad \{B, E\} \prec_W^m \{D\} \qquad \{C, E\} \prec_W^m \{D\}$$

providing respectively the places p_1, p_2, p_3, and p_4 of the net system displayed in Exple. 2.1.

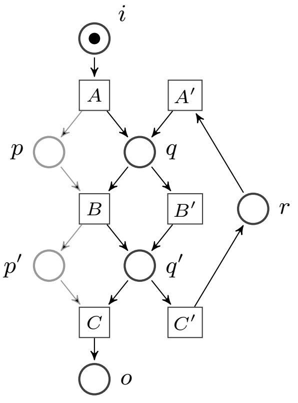
3 Some Observations about the Algorithm α

The reader may feel uncomfortable with the fact that the synthesized places are associated only from the maximal elements of relation $\prec_W$. Of course, associating a place p with every pair such that $A \prec_W B$ would produce a net with a large number of places. We may expect many of these additional places (if not all) to be implicit, so that one can remove them without having an impact on the behaviour of the system.

Definition 3.1. *A place p of a (contact-free) net system $N = (P, T, F, M_0)$ is a* (structurally) implicit place *if for every reachable marking M and transition $t \in p^\bullet$, ${}^\bullet t \setminus \{p\} \subseteq M \Rightarrow p \in M$.*

The next example shows however that some of these places *are not implicit* and therefore the decision to restrict to the maximal elements of the relation $\prec_W$ has an impact of the expressivity of algorithm α.

Example 3.2. *Let us consider the workflow net N depicted next. Its language is $\mathcal{L}(N) = A\,(B'C'A')^*\,B\,(C'A'B')^*\,C$. A complete log is $W = \{ABC, AB'C'A'BC'A'B'C\}$. Places q, q' and r correspond to maximal elements of relation $\prec_W$, namely $\{A, A'\} \prec_W^m \{B, B'\}$, $\{B, B'\} \prec_W^m \{C, C'\}$, and $\{C'\} \prec_W^m \{A'\}$ respectively. Whereas ${}^\bullet p = \{A\} \prec_W^m \{B\} = p^\bullet$ and ${}^\bullet p' = \{B\} \prec_W^m \{C\} = p'^\bullet$ are not. Nevertheless these places are not implicit places since they disable the firing of transition B (respectively C) in the marking reached after firing $ABC'A'$ (resp. AB') where place q (resp. place q') is marked. The net $N' = \alpha(W)$ synthesized by algorithm α is obtained from N by suppressing these two places p and p'. Its language is $\mathcal{L}(N') = A\,(B + B')\,(C'A'\,(B + B'))^*\,C$.*



Algorithm α is *sober* in the sense that for any complete log $W \subseteq \mathcal{L}(N)$ of a workflow net N any larger log $W \subseteq W' \subseteq \mathcal{L}(N)$ is also complete and the minimal size of complete logs of workflow nets is asymptotically negligible w.r.t. the size of their languages. Indeed when N ranges over workflow nets with set of transitions T, the size of $\mathit{Abs}(\mathcal{L}(N))$ is in $O(|T|^2)$. Moreover, a firing sequence of N contained in a log W may contribute several pairs of transitions in C_W. Therefore, one may expect to find complete logs of N with size even smaller than $O(|T|^2)$. Sobriety means that one can assume $W \subseteq \mathcal{L}(N)$ to be a complete log of workflow net N as soon as it contains a reasonable number of its execution sequences. Process reconstruction then amounts to the following:

Workflow net discovery

Problem Reconstruct a workflow net N from one of its complete log $W \subseteq \mathcal{L}(N)$.
Solution N is α-reconstructible if and only if $N \cong \alpha(W)$ if and only if the following two conditions hold: *(i)* $\alpha(W)$ is a workflow net, and *(ii)* $W \subseteq \mathcal{L}(\alpha(W))$, i.e., the synthesized net can reproduce each execution sequence in W.

The reader may have expected any net constructed by algorithm α to be an α reconstructible workflow net. Or more precisely, that α algorithm computes a closure operation providing the best approximation of a given log by a workflow net. This is unfortunately not the case as illustrated by the next example.

Example 3.3. *Algorithm α relies only on the local informations reported in $\mathit{Abs}(\mathcal{L}(N))$. The price to pay for the locality of observations is that one cannot detect a situation where a transition t is an immediate cause of t' (there exists a non implicit place*[2] *in $t^\bullet \cap {}^\bullet t'$) but t' cannot occur immediately after t because it depends on others transitions that can be fired only after t. The following example illustrates this situation. In the language of the workflow net of Fig. 1, namely $W = \{ACD, BCE\}$, event D never follows immediately event A even though A is an immediate cause of D. Note however that p is not an implicit place because it disables D to be fired after the sequence BC: q is marked but not p after firing this sequence. The workflow net derived by algorithm α from W, the language of the workflow net of Fig. 1 is displayed in Fig. 2. This net reproduces the original workflow net of Fig. 1 but for the two places p and p'. Due to the absence of these two places the dependency of D on A (and similarly the dependency of E on B) is not inferred by algorithm α and the resulting net system contains two additional execution sequences ACE and BCD. In this case algorithm α looses precision by generalizing too much (it is underfitted). N_1 is not α-recontructible. Still N_1 is isomorphic to $\alpha(W')$ where $W' = \{ACD, BCE, AD, BE\}$; however $W' \not\subseteq$*

[2] Since implicite place have no impact on the net behaviour an implicit place in $t^\bullet \cup {}^\bullet t'$ can represent a fictitious dependency between t and t'. We say that t is an *immediate cause* of t' when $t^\bullet \cup {}^\bullet t'$ contains a non implicit place.

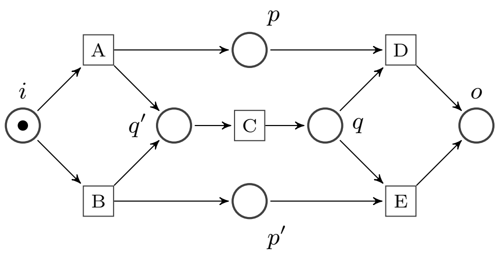


Fig. 1. a workflow net N_1

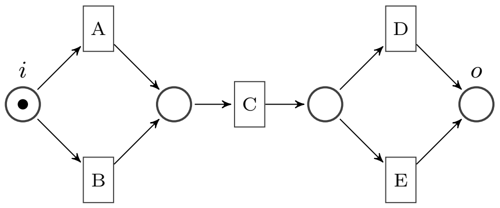


Fig. 2. The workflow net N_2 constructed by algorithm α from the language of the workflow net N_1 of Fig. 1

$\mathcal{L}(N_1) = \{ACD, BCE\}$. *Thus a net produced from a log does not necessarily reproduce every execution sequence contained in the log.*

The preceding example shows that α cannot be associated with a Galois connection $W \subseteq \mathcal{L}(N) \Leftrightarrow N \leq \alpha(W)$ where $\leq$ is some order (or pre-order) relation on net systems. If it were the case we would have $W \subseteq \mathcal{L}(\alpha(W))$ for every $W \subseteq T^*$, in contradiction with the above example. Thus α cannot be used to provide the best net system N, according to some order, or pre-order relation, whose language contains a given log. For that reason we put stress on *process discovery* and in this context sobriety is a crucial property since we need to be assured that the log given as input is a complete log of the net to be discovered.

An alternative process mining algorithm based on regions in elementary net systems [7, 8, 6, 1] was proposed in [3, 4]. It is not the purpose of this paper to provide a detailed presentation of this algorithm for which we refer the reader to the above mentionned references and [2]. We just want to stress on the differences between the two approaches: *process discovery* (α algorithm) versus *process approximation* (ω algorithm). The latter algorithm produces a workflow net $\omega(W)$ whose language is the least language of a workflow net containing the log $W \subseteq T^*$.

Example 3.4. *Let us consider the elementary net system N_1 depicted on left part of Fig. 3 with language $W = \{ACDE, BDCE\}$. This net is not a workflow net because places p_2 (respectively p_3) gets marked after the firing of sequence ACDE (resp. BDCE) hence these places will not be synthesized by algorithm ω.*

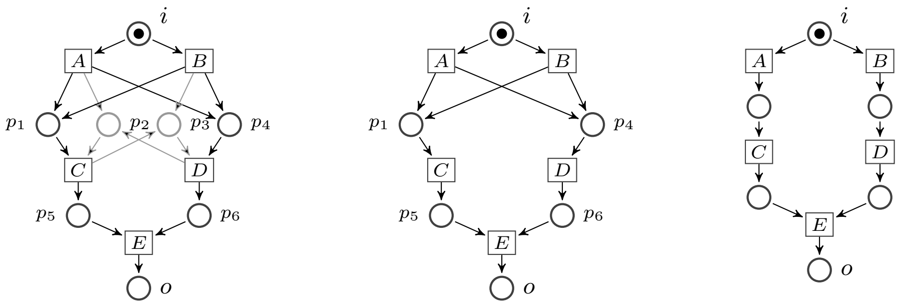


Fig. 3. An elementary net system (left) with language $W = \{ACDE, BDCE\}$, its workflow net approximation given by algorithm ω (center), and the workflow net synthesized by α from W (right)

The net $N_2 = \omega(W)$ [3] shown in the center of Fig. 3 reproduces net N_1 without these two places; its language $\mathcal{L}(N_2) = \{ACDE, ADCE, BCDE, BDCE\}$ is the least language of a workflow net containing W. If we now apply the α-algorithm to the log W, we obtain $A \sharp_W B$, $A \sharp_W D$, $B \sharp_W C$, $C \|_W D$ and the immediate causalities $A \to_W C$, $B \to_W D$, $C \to_W E$, and $D \to_W E$. The resulting net $N_3 = \alpha(W)$, depicted on the right-part of Fig. 3, is a workflow net but $W = \{ACDE, BDCE\} \nsubseteq \mathcal{L}(N_3) = \{ACE, BDE\}$. α may fail to find a workflow net realizing all computation sequences in a given log W, i.e., a workflow net N such that $W \subseteq \mathcal{L}(N)$. In that case, α provides no solution of any kind. In contrast, ω always produces the optimal solution, i.e., a workflow net with the least possible language containing W.

Algorithm ω is also able to discover the workflow net N_1 of Exple. 3.3. This extended expressivity is however at the price of a higher computational complexity due to the fact that regions are global properties of the log whereas α considers only very local informations that are easy to extract: the α-algorithm is much faster and less space consuming that the ω-algorithm.

Since α does not provide useful information when it fails to discover a workflow net it is the more so important to be able to identify the class of α-reconstructible workflow nets. This was the motivation for restricting the α-algorithm to the context of *structured workflow nets*. The main result announced in [13] and proven in the report [12] is that structured workflow nets without short loops (a condition that we introduce in the next section) are α-reconstructible. Let us briefly introduce structured workflow nets. The reader may have observed that the workflow net N_1 of Exple. 3.3 is not a *free-choice* net [5]. The two choices between events A and B and between events D and E are not independent: if one

[3] More precisely, the ω algorithm [2] is presented with two variants. In the first case, places are the so called ω-regions –regions that may appear as extensions of places of workflow nets–. A simplified, and language equivalent, version is obtained by restricting the inner places to be *minimal* regions. It is the simplified version that is depicted in Fig. 3.

chooses A (resp. B), then one must choose D (resp. E). In fact, one cannot choose between events D and E at run time, since both events are never jointly enabled. Structured workflow nets satisfy a property slightly stronger that the free-choice property, that already excludes such interferences between conflict (the sharing of an input place by two transitions) and synchronization (the sharing of two input places by a transition).

Definition 3.5. *A workflow net $N = (P, T, F, M_0)$ is a structured workflow net if it has no structurally implicit places and the following condition holds:*

$$\forall t \in T \quad |{}^\bullet t| > 1 \Rightarrow (\forall p \in {}^\bullet t \quad |{}^\bullet p| = 1 \ \wedge \ |p^\bullet| = 1) \qquad \text{(SWN)}$$

i.e., if a transition t requires the synchronization of several conditions (places), then each of these conditions has a unique cause ($|{}^\bullet p| = 1$) and a unique consequence ($|p^\bullet| = 1$), hence it cannot induce a conflict between t and another transition t'.

The main result established in [12] is the following.

Theorem 3.6. *Structured workflow nets without short loops are α-reconstructible.*

Adding structurally implicit places to a net preserves its language, and removing places from a net satisfying condition (SWN) cannot invalidate this condition. Therefore, one can state the following corollary to Th. 3.6.

Corollary 3.7. *A workflow net N without short loops and satisfying condition (SWN) is always language equivalent to some α-reconstructible workflow net N'.*

Condition (SWN) is a structural condition, hence it can be checked very efficiently. It was argued [13] that the class of nets satisfying this condition supports all basic routing patterns and building blocks used to construct workflow systems in practice. The conditions characterizing structured workflow nets (Def. 3.5) are sufficient, but however not necessary, to ensure α-reconstructibility. Actually, an α-reconstructible net may contain structurally implicit places (Exple. 3.8 below) and it may not satisfy condition (SWN) (Exple. 2.1).

Example 3.8. *Consider transitions C, F, G in the workflow net N depicted on the left of Fig. 4. If we let $W = \mathcal{L}(N)$, then we get $C \sharp_W F$, $C \rightarrow_W G$, and $F \rightarrow_W G$.*

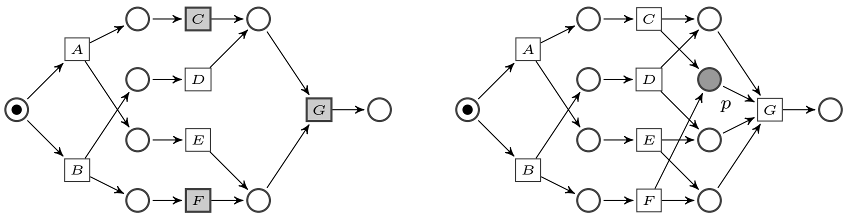


Fig. 4. A place p of an α-reconstructible net which is a structurally implicit place

Therefore, the α-algorithm necessarily produces a place in $C^\bullet \cap F^\bullet \cap {}^\bullet G$. This is indeed the place p that appears in the net $N' = \alpha(\mathcal{L}(N))$ shown on the right of Fig. 4. Another place in $D^\bullet \cap E^\bullet \cap {}^\bullet G$ appears symmetrically in N'. Now N and N' are language equivalent, and therefore, N' is α-reconstructible, and p is clearly a structurally implicit place of N'.

4 α Reconstructible Workflow Nets

In view of the preceding discussion we try in this section to obtain a more precise understanding of α-reconstructibility of workflow nets. The purpose of algorithm α is to deduce places of the net on the basis of the information $Abs(W)$ extracted from the event log. Notice that for every place p of an elementary net one has *(i)* $\forall a, a' \in {}^\bullet p \; a \sharp_N a'$, *(ii)* $\forall b, b' \in p^\bullet \; b \sharp_N b'$, and *(ii)* $\forall a \in {}^\bullet p \; \forall b \in p^\bullet \; a \rightarrow_N b$ where

$$\begin{aligned} t \rightarrow_N t' &\Leftrightarrow t^\bullet \cap {}^\bullet t' \neq \emptyset \\ t \sharp_N t' &\Leftrightarrow ({}^\bullet t \cap {}^\bullet t') \cup (t^\bullet \cap t'^\bullet) \neq \emptyset \\ t \parallel_N t' &\Leftrightarrow ({}^\bullet t \cup t'^\bullet) \cap ({}^\bullet t \cup t'^\bullet) = \emptyset \end{aligned}$$

In order to correctly infer the places of a workflow net N from the abstraction $Abs(\mathcal{L}(N))$ of its language (or of any of its complete log W) the above relations of causality, conflict and concurrency associated respectively with N and W should fit at best.

Proposition 4.1. *Let $W = \mathcal{L}(N)$ be the full log of a workflow net.*

1. $t \rightarrow_W t' \Rightarrow t \rightarrow_N t'$, $t \| _W t' \Rightarrow t \| _N t'$, *and* $t \sharp_N t' \Rightarrow t \sharp_W t'$
2. *if t and t' are co-enabled, i.e. there exists some reachable marking M such that $M[t\rangle$ and $M[t'\rangle$, then $t \| _N t' \Leftrightarrow t \| _W t'$ and $t \sharp_N t' \Leftrightarrow t \sharp_W t'$*

Proof. These properties are easily derived from the firing rule of elementary net systems using the fact that a workflow net is contact-free. More precisely one can successively check that

1. $(M[t \cdot t'\rangle \wedge M[t'\rangle) \Leftrightarrow (t \| _N t' \wedge {}^\bullet t \cup {}^\bullet t' \subseteq M \wedge M \cap (t^\bullet \cup t'^\bullet) = \emptyset)$ and one has $M[t' \cdot t\rangle$ and $M_{t \cdot t'} = M_{t' \cdot t}$ in this case.
2. $M[t \cdot t'\rangle \Leftrightarrow \neg(t \sharp_N t') \wedge {}^\bullet t \cup ({}^\bullet t' \setminus t^\bullet) \subseteq M \wedge M \cap (t^\bullet \cup (t'^\bullet \setminus {}^\bullet t)) = \emptyset$
3. From $M[t \cdot t'\rangle$ and $t^\bullet \cap {}^\bullet t' = {}^\bullet t \cap t'^\bullet = \emptyset$ it follows that $M[t'\rangle$ and hence $t \| _N t'$.
4. If N is contact-free, then $(M[t \cdot t'\rangle \wedge t^\bullet \cap {}^\bullet t' = \emptyset) \Rightarrow M[t'\rangle$
5. Then $t \sharp_N t' \Rightarrow t \sharp_W t'$, and $t \sharp_N t' \Leftrightarrow t \sharp_W t'$ if t and t' are co-enabled, i.e., $M[t\rangle$ and $M[t'\rangle$ for some reachable marking M.
6. The following relations hold if N is contact-free:
   (a) $t \rightarrow_W t' \Rightarrow t \rightarrow_N t'$,
   (b) $t \| _W t' \Rightarrow t \| _N t'$,
   (c) if t and t' are co-enabled then $t \| _N t' \Leftrightarrow t \| _W t'$. □

The following example illustrates the mismatch between the structural ($\rightarrow_N$) and behavioural ($\rightarrow_W$) relations of causality that can arise from contact situations.

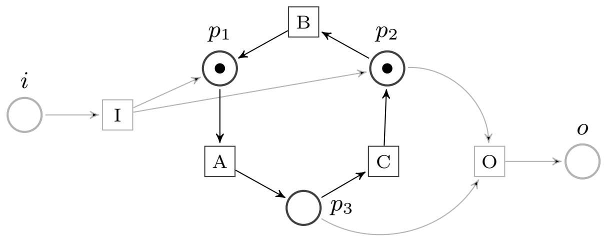


Fig. 5. A workflow with (many) contact situations, the marking indicated in this picture is the marking reached after the firing of the initial transition I

Example 4.2. *The language of the workflow net depicted in Fig. 5 is $I(ABC)^*AO$. A complete log of this language is given by $W = \{IABCAO\}$ from which we deduce the following causal dependencies: $A \rightarrow_W B$, $B \rightarrow_W C$, and $C \rightarrow_W A$. The relations induced by the net system report (fake) immediate causalities in the reverse direction: $B \rightarrow_N A$, $A \rightarrow_N C$, and $C \rightarrow_N B$.*

It can be inferred from the non emptiness of $t^\bullet \cap {}^\bullet t'$ that transition t is an immediate cause of t' only if one can find at least one place $p \in t^\bullet \cap {}^\bullet t'$ that is used as a ressource for t' and not for the purpose of disabling transition t. This is the rationale for requiring workflow nets to be contact-free.

The inclusion $\rightarrow_{\mathcal{L}(N)} \subseteq \rightarrow_N$ shows that the causal relation inferred from the observations $t \cdot t' \in C_{\mathcal{L}(N)}$ and $t' \cdot t \notin C_{\mathcal{L}(N)}$ (according to Def. 2.4) implies the existence of a connecting place $p \in t^\bullet \cap {}^\bullet t'$. As just noticed contact-freeness avoids the production of connecting places representing fake dependencies. But this restriction is not sufficient to guarantee that the converse inclusion $\rightarrow_N \subseteq \rightarrow_{\mathcal{L}(N)}$ holds. In particular, as illustrated by the following example, cyclic dependencies given by short loops is another obstacle to α-reconstructibility.

Definition 4.3. *Two transitions of a (contact-free) elementary net system form a* short loop *if $t^\bullet \cap {}^\bullet t' \neq \emptyset$ and $t'^\bullet \cap {}^\bullet t \neq \emptyset$.*

Example 4.4. *The workflow net N of Fig. 6 has a short loop involving transitions B and C. A complete log of N is $W = \{ABCBD\}$. The abstraction of this log is $Abs(W) = \{\{A\}, \{AB, BC, CB, BD\}, \{D\}\}$. Since both short sequences BC and CB belong to C_W, one gets $B \parallel_W C$. Thus cyclic dependencies within short*

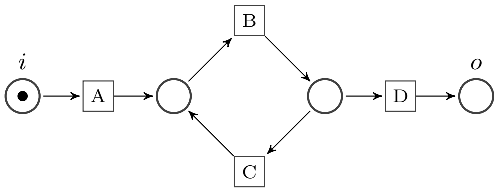


Fig. 6. A workflow net with a short loop

loops are lost when applying the α-algorithm. The net system synthesized by the function Syn from Abs(W) is shown in Fig. 7. This net is not a workflow net since the transition C is isolated, hence α-algorithm fails in that case.

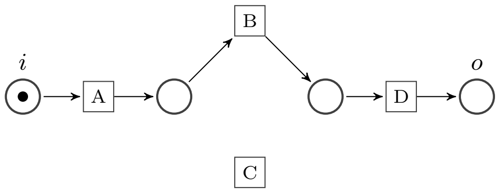


Fig. 7. The net system constructed by the algorithm α from the complete log $W = \{ABCBD\}$ of the workflow net of Fig. 6

Proposition 4.5. *If N is a workflow net without short loops then*

$$t^\bullet \cap {}^\bullet t' \neq \emptyset \quad \Rightarrow \quad t' \cdot t \notin C_{\mathcal{L}(N)}$$

Proof. $t^\bullet \cap {}^\bullet t' \neq \emptyset$ implies $t'^\bullet \cap {}^\bullet t = \emptyset$ (no short loops). Suppose by way of contradiction that $t' \cdot t \in C_{\mathcal{L}(N)}$. Then there exists a reachable marking M such that $M[t' \cdot t\rangle$. Since $t'^\bullet \cap {}^\bullet t = \emptyset$ and the net is contact-free we have $M[t\rangle$ and thus $t \|_N t'$ which is in contradiction with $t^\bullet \cap {}^\bullet t' \neq \emptyset$. $\square$

In view of Proposition 4.5, in order to establish $\rightarrow_N \subseteq \rightarrow_{\mathcal{L}(N)}$ (and thus $\rightarrow_N = \rightarrow_{\mathcal{L}(N)}$ since the converse implication holds by Prop. 4.1) it remains to ensure that $t^\bullet \cap {}^\bullet t' \neq \emptyset$ implies $t \cdot t' \in C_{\mathcal{L}(N)}$. This condidition can be restated as the following requirement on places.

Definition 4.6. *An inner place p of a workflow net is said to be a* boundary *place when:* $\forall t \in {}^\bullet p \quad \forall t' \in p^\bullet \qquad t \cdot t' \in C_{\mathcal{L}(N)}$.

For instance place p (or similarly place p') of workflow net N_1 depicted in Fig. 1 is not a boundary place since ${}^\bullet p = \{A\}$, $p^\bullet = \{D\}$, and there is no firing sequence of this net in which A is immediately followed by D. Indeed, the two non-boundary places p and p' cannot be discovered by algorithm α (Fig. 2).

Corollary 4.7. *Let N be a workflow net without short loops and all of whose inner places are boundary places, then $\rightarrow_N = \rightarrow_{\mathcal{L}(N)}$.*

From the definition of the structural relations of causality $\rightarrow_N$ and conflict $\sharp_N$ associated with a net system N, it follows that for every place p one has

1. $\forall t \in {}^\bullet p\ \forall t' \in p^\bullet \ \ t \rightarrow_N t'$,
2. $\forall t, t' \in {}^\bullet p \ \ t \sharp_N t'$, and
3. $\forall t, t' \in t^\bullet \ \ t \sharp_N t'$.

Using $\sharp_N \subseteq \sharp_{\mathcal{L}(N)}$ (by Prop. 4.1) and $\rightarrow_N \subseteq \rightarrow_{\mathcal{L}(N)}$ (by Cor. 4.7) we deduce the following result.

Corollary 4.8. *${}^\bullet p \prec p^\bullet$ for every place of a workflow net N without short loops and whose inner places are boundary places, where $\prec = \prec_{\mathcal{L}(N)}$ (see Def. 2.6)* [4].

Elements of the (graph of) relation $\prec = \{(A,B) \in \wp(T) \times \wp(T) \mid A \prec B\}$ are generalizations of the (extents of) places of the net to be discovered. In analogy with regions in transition systems, we might called them α-regions. The pair $({}^\bullet p, p^\bullet)$, called the α-*extent* of place $p \in P$, belongs (by above Cor. 4.8) to relation $\prec$ when p is a boundary place. Moreover when equipped with the component-wise order relation $(A,B) \sqsubseteq (A',B') \;\Leftrightarrow\; A \subseteq A' \wedge B \subseteq B'$ the set $\prec$ is downward-closed:

$$(A \prec B \;\wedge\; A' \subseteq A \;\wedge\; B' \subseteq B) \;\Rightarrow\; A' \prec B'$$

The fact that p is a boundary place yields that $A \prec B$ for every $A \subseteq {}^\bullet p$ and $B \subseteq p^\bullet$. We say that place p *justifies* the corresponding observation $A \prec B$. In order to synthesize a net from $Abs(\mathcal{L}(N))$ we have to find enough abstract places (α-regions) for justifying every element $A \prec B$ in that relation. This led us to consider the following class of workflow nets:

Definition 4.9. *A workflow net without short loops and all of whose inner places are boundary places is said to be an α-workflow net when a place p exists such that $A \subseteq {}^\bullet p$ and $B \subseteq p^\bullet$ whenever $A \prec B$.*

The main result of this section is the following theorem, which sums up the various elements presented so far in the course of this section.

Theorem 4.10. *An α-workflow net N whose places have incomparable α-extents (i.e., ${}^\bullet p \subseteq {}^\bullet q \;\wedge\; p^\bullet \subseteq q^\bullet \quad\Rightarrow\quad p = q$) is α-reconstructible; i.e., $N \cong \alpha(W)$ for any complete log $W \subseteq \mathcal{L}(N)$ of N.*

Proof. Let N be an α-workflow net whose places have incomparable extents for the order relation $\sqsubseteq$. We have $i^\bullet = I_{\mathcal{L}(N)}$ and these transitions can occur only as the first elements of execution sequences, symetrically ${}^\bullet o = O_{\mathcal{L}(N)}$ and these transitions can occur only as the last transitions of execution sequences. Thus we also have $i^\bullet = I_W$ and ${}^\bullet o = O_W$ since $W \subseteq \mathcal{L}(N)$ contains at least one occurrence of each transition. Let us now proceed to a comparison of the respective sets of inner places of nets N and $\alpha(W)$. Cor. 4.8 states that the extents of inner places of N are elements of relation $\prec = \prec_W$. The condition stated in Def. 4.9 ensures that all maximal elements of $\prec_W$ are extents of places. Therefore the extents of places are exactly the maximal elements of $\prec_W$, i.e. they are the places of $\alpha(W)$, if and only if they are incomparable for order relation $\sqsubseteq$. $\square$

Notice that, by definition, the places of $\alpha(W)$ have incomparable extents. Thus *an α-workflow net is α-reconstructible if and only if it has incomparable places.* More significantly, we establish in the next section a converse to Theo. 4.10 which allows to *identify α-reconstructible workflow nets with those α-workflow nets whose places have incomparable extents.*

[4] We have also $\prec = \prec_W$ for any complete log $W \subseteq \mathcal{L}(N)$ of N.

Example 4.11 (example 3.2 continued). *The extents of places p and p' of Exple. 3.2 are not maximal elements w.r.t. $\sqsubseteq$ since $({}^\bullet p, p^\bullet) \sqsubseteq ({}^\bullet q, q^\bullet)$ and $({}^\bullet p', p'^\bullet) \sqsubseteq ({}^\bullet q', q'^\bullet)$. The language of the net $N' = \alpha(W)$ synthesized from its complete log W is not the least language of a workflow net containing W (since N, which is also an α-workflow net, gives in that respect a better approximation). But N', unlike N, is α-reconstructible.*

5 Boundary Places of a Workflow Net

In this section we provide a characterization of the boundary places and we describe their relationship with the non structurally implicit places. For some classes of net systems these two notions coincide.

Proposition 5.1. *Let N be a workflow net that satisfies Condition (SWN). An inner place of N is a boundary place if and only if it is a non structurally implicit place.*

Proof. The key argument is the observation that for any pair of transitions t and t' of a workflow net that satisfies Condition (SWN) the set $t^\bullet \cap {}^\bullet t'$ contains at most one place. Let p be a boundary place of N, then there exists accessible markings M and M' such that $M[t\rangle M'[t'\rangle$ and marking M satisfies $p \notin M$ and ${}^\bullet t' \setminus \{p\} \subseteq M$ (since none of these places belong to $t^\bullet$ by the preceding remark). Hence p is a non structurally implicit place. Conversely let p be a non structurally implicit place of N.

1. If $|p^\bullet| > 1$ then for every $t' \in p^\bullet$ one has ${}^\bullet t' = \{p\}$. Since the net is initially life every $t \in {}^\bullet p$ is enabled in some reachable marking M, and since the net is contact-free t' is enabled in the marking M' such that $M[t\rangle M'$. Hence $t \cdot t' \in C_{\mathcal{L}(N)}$ and p is a boundary place.
2. If $p^\bullet = \{t'\}$ and ${}^\bullet t' = \{p\}$ then the same argument applies: $t \in {}^\bullet p$ is enabled in some reachable marking M, and since the net is contact-free t' is enabled in the marking M' such that $M[t\rangle M'$. Hence $t{\cdot}t' \in C_{\mathcal{L}(N)}$ and p is a boundary place.
3. If $p^\bullet = \{t'\}$ and $|{}^\bullet t'| > 1$ then $p'^\bullet = \{t'\}$ for every $p' \in {}^\bullet t'$ and $|{}^\bullet p| = 1$, say ${}^\bullet p = \{t\}$. Let M be a reachable marking such that ${}^\bullet t' \setminus \{p\} \subseteq M$ and $p \notin M$. Since places in ${}^\bullet t'$ are inner places they should be emptied by a transition enabled in a marking M' reachable from M. This transition is necessarily t' and the firing of t' requires that p be filled, hence t be fired beforehand, and t' can then fired immediately after t. Hence p is a boundary place. □

The input and output places are non implicit and we could equivalently have defined structured workflow nets as workflow nets satisfying Condition (SWN) and all whose inner places are boundary places. However in the general case these two notions differ. For instance the place p and p' in the workflow net of Exple. 3.3 are neither boundary places nor structurally implicit places, and the place p in net N' of Exple. 3.8 is both a boundary place and a structurally implicit place.

Let us now proceed to the characterization of boundary places. The language of a workflow net N is closed under the congruence $\sim$ generated by the relations $t \cdot t' \sim t' \cdot t$ pertaining to pairs of concurrent transitions t and t' $(t \|_N t')$. An equivalence class of maximal execution sequences of N is called a *process* of N. Processes of a workflow net N may be represented equivalently as follows.

Definition 5.2. *A* process *of a workflow net $N = (P, T, F, M_0)$ is a pair $\mathcal{R} = (R, \ell)$ consisting of a net $R = (P_R, T_R, F_R)$ and two labelling functions $\ell : T_R \to T$ and $\ell : P_R \to \wp(P)$ satisfying the following conditions:*

1. *There is a place i_R such that ${}^\bullet i_R = \emptyset$, and $\ell(i_R) = \{i\}$ where i is the input place of the workflow net N.*
2. *There is a place o_R such that $o_R{}^\bullet = \emptyset$, and $\ell(o_R) = \{o\}$ where o is the output place of the workflow net N.*
3. *$\forall p_R \in P_R \setminus \{i_R, o_R\} \quad |{}^\bullet p_R| = 1$ and $|p_R{}^\bullet| = 1$.*
4. *$\forall t_R \in T_R \quad {}^\bullet t_R \neq \emptyset$ and $t_R{}^\bullet \neq \emptyset$.*
5. *The underlying graph of R is acyclic.*
6. *$\{\ell(p_R) \mid p_R \in {}^\bullet t_R\}$ is a partition of ${}^\bullet \ell(t_R)$.*
7. *$\{\ell(p_R) \mid p_R \in t_R{}^\bullet\}$ is a partition of $\ell(t_R)^\bullet$.*

The above definition differs slightly from the usual definition of processes as occurrence nets [10]. The sole difference is that here, *each place in a process R is mapped by ℓ to a set of places of N, playing indistinguishable roles in this process.* As a result, processes are free from equivalent places. In the sequel, we let $\ell(M_R) = \bigcup \{\ell(p_R) \mid p_R \in M_R\}$ denote the marking of N associated by ℓ with the marking M_R of R.

Proposition 5.3. *Processes $\mathcal{R} = (R, \ell)$ of a workflow net N are in bijective correspondence with the equivalences classes of complete execution sequences of N modulo permutation of concurrent transitions.*

The proof of this proposition is delayed until some additional results have been proved starting with the following lemma.

Lemma 5.4. *If $\mathcal{R} = (R, \ell)$ is a process of a workflow net N then*

$$\{i_R\}\,[t_1 \cdots t_k\rangle\,\{o_R\} \text{ in } R \quad \Leftrightarrow \quad \{i\}\,[\ell(t_1) \cdots \ell(t_k)\rangle\,\{o\} \text{ in } N$$

Proof. More generally for $X \subseteq T_R$ we let $\ell(X) = \bigcup \{\ell(p_R) \mid p_R \in X\}$, in particular $\ell({}^\bullet t_R) = {}^\bullet \ell(t_R)$ and $\ell({}^\bullet t_R) = {}^\bullet \ell(t_R)$ for every $t_R \in T_R$. $M_R[t_R\rangle M'_R$ in R iff $M_R \setminus M'_R = {}^\bullet t_R$ and $M'_R \setminus M_R = t_R{}^\bullet$ iff $\ell(M_R \setminus M'_R) = \ell(M_R) \setminus \ell(M'_R) = {}^\bullet \ell(t_R)$ and $\ell(M'_R \setminus M_R) = \ell(M'_R) \setminus \ell(M_R) = \ell(t_R)^\bullet$ iff $\ell(M_R)[\ell(t_R)\rangle \ell(M'_R)$ in N. Thus $\{i_R\}\,[t_1 \cdots t_k\rangle\,\{o_R\}$ in R iff $\{i\}\,[\ell(t_1) \cdots \ell(t_k)\rangle\,\{o\}$ in N. $\square$

We recall that if two firing sequences of the form $u \cdot t \cdot t' \cdot v$ and $u \cdot t' \cdot t \cdot v$ are both enabled in the initial marking of an elementary net system N, then one necessarily has $t \|_N t'$ and these sequences are permutation equivalent.

Corollary 5.5. *Let $\mathcal{R} = (R, \ell)$ be a process of a workflow net N. The set $\mathcal{L}_\mathcal{R} = \{\ell(t_1) \cdots \ell(t_n) \mid t_1 \cdots t_n \in \mathcal{L}(R)\}$ where $\mathcal{L}(R)$ is the set of firing sequences of R starting in $\{i_R\}$ and ending in $\{o_R\}$ is closed by permutation of concurrent transitions.*

Lemma 5.6. *Any firing sequence $u = t_1 \cdots t_k \in \mathcal{L}(N)$ of a workflow net N can be associated with a process $\mathcal{R} = (R, \ell)$ of N defined as follows. We let $T_R = \{\tilde{t}_1, \cdots, \tilde{t}_k\}$ with $\ell(\tilde{t}_i) = t_i$. For $1 \le i < j \le k$ we let*

$$P_{i,j} = \{p \in t_i{}^\bullet \cap {}^\bullet t_j \mid \forall i < j' < j \;\; p \notin {}^\bullet t_{j'}\}$$

We further let P_R stand for the set $\{p'_{i,j} \mid P_{i,j} \neq \emptyset\}$ together with two additional places, i_R and o_R, where

- *$\ell(i_R) = i$, and $\ell(o_R) = o$,*
- *$\ell(\tilde{p}_{i,j}) = P_{i,j}$,*
- *${}^\bullet\tilde{t}_1 = i_R$, and ${}^\bullet\tilde{t}_j = \{\tilde{p}_{i,j} \mid 1 \le i < j\}$ for $1 < j \le k$, and*
- *$\tilde{t}_i{}^\bullet = \{\tilde{p}_{i,j} \mid i < j \le k\}$ for $1 \le i < k$, and $\tilde{t}_k{}^\bullet = \{o_R\}$.*

Proof. Indeed the non empty sets $P_{i,j}$ for $j \in \{i+1, \ldots, k\}$ form a partition of $t_i{}^\bullet$ and the non empty sets $P_{i,j}$ for $i \in \{1, \ldots, j-1\}$ form a partition of ${}^\bullet t_j$ and $\mathcal{R} = (R, \ell)$ satisfies the conditions in Def. 5.2. □

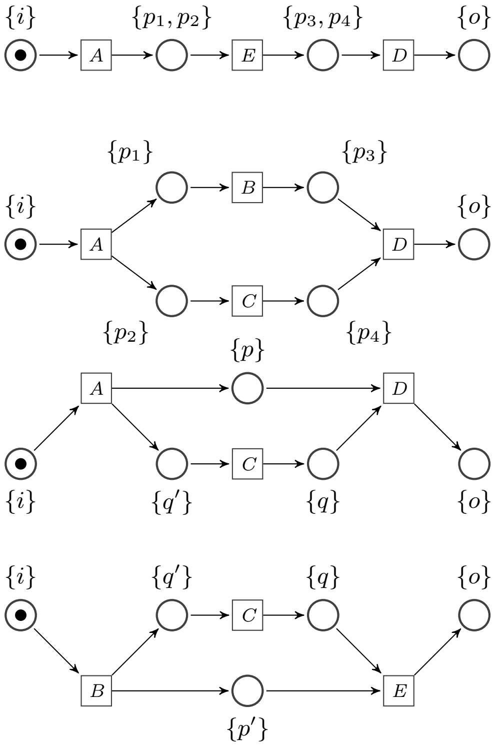
Proof (of Prop. 5.3). In view of the two preceding lemmas and Corollary 5.5 it just remains to prove that any two sequences in $\mathcal{L}(R)$ are permutation equivalent. We note that R is a workflow net, in particular it is contact-free. Moreover, since it contains no conflict it is a persistent net: once a transition is enabled it remains enabled until it is actually fired. Since the net is acyclic a place is never filled twice and a transition is fired exactly once along any maximal firing sequence. It follows by induction that any two sequences in $\mathcal{L}(R)$ are permutation equivalent. □

Proposition 5.7. *An inner place p of a workflow net is a boundary place if and only if for every pair of transitions $t \in {}^\bullet p$ and $t' \in p^\bullet$ there exists a process $\mathcal{R} = (R, \ell)$ of N and a* ***non (structurally) implicit*** *place $p_R \in P_R$ in this process with $p_R \in t_R{}^\bullet \cap {}^\bullet t'_R$ such that $\ell(t_R) = t$, $\ell(t'_R) = t'$ and $p \in \ell(p_R)$.*

Proof. Let p a boundary place of a workflow net N. For every $t \in {}^\bullet p$ and $t' \in p^\bullet$ there exists a firing sequence $u = t_1 \cdots t_k \in \mathcal{L}(N)$ such that $t = t_i$ and $t' = t_{i+1}$ for some index i. Let $\mathcal{R} = (R, \ell)$ be the process of N associated with u as described in Lemma 5.6. Since $p \in P_{i,i+1}$ the place $\tilde{p}_{i,i+1}$ and the transitions $\tilde{t}_i$ and $\tilde{t}_{i+1}$ of the process satisfy $\ell(\tilde{t}_i) = t_i$, $\ell(\tilde{t}_{i+1}) = t_{i+1}$, and $p \in \ell(\tilde{p}_{i,i+1}) = P_{i,i+1}$. Since $\tilde{p}_{i,i+1}$ is the unique element of $\tilde{t}_i{}^\bullet \cap {}^\bullet\tilde{t}_{i+1}$ every input places of $\tilde{t}_{i+1}$ but $\tilde{p}_{i,i+1}$ are marked after firing the sequence $t_1 \ldots t_i$. Therefore place $\tilde{p}_{i,i+1}$ is not an implicit place. Conversely assume that for every pair of transitions $t \in {}^\bullet p$ and $t' \in p^\bullet$ there exists a process $\mathcal{P} = (R, \ell)$ of N and a non implicit place $p_R \in P_R$ in this process with $p_R \in t_R{}^\bullet \cap {}^\bullet t'_R$ such that $\ell(t_R) = t$, $\ell(t'_R) = t'$ and $p \in \ell(p_R)$. There exists a reachable marking where every input place of t'_R but p_R are marked, then t'_R becomes enabled as soon (and only when) transition t_R fires and t'_R can fire immediately then. By Lemma 5.4 we can deduce the existence of a firing sequence of N where t is immediately followed by t' thus showing that p is a boundary place. □

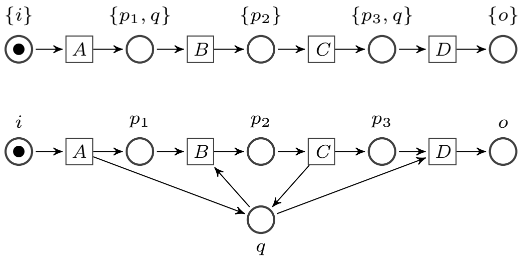
Example 5.8
The workflow net of Exple. 2.1 has two complete processes given next from which we deduce that all of its inner places are boundary places. Actually these two processes contains no implicit places and they allow to cover all possible triples (t, p, t') with $p \in t^\bullet \cap {}^\bullet t'$ from the workflow net.

The workflow net of Exple. 3.3 has two complete processes given next. Place $\{p\}$ is an implicit place of the first process since it is marked in all reachable markings that contain place $\{q\}$. Similarly place $\{p'\}$ is an implicit place of the second process. We deduce that places p and p' in the original workflow net are not boundary places (even though they are not implicit places).



If $p \in t^\bullet \cap {}^\bullet t'$ is a boundary place of a workflow net N and $\tilde{p}$ is a non implicit place and $\tilde{t}$ and $\tilde{t'}$ are transitions of a process $\mathcal{R} = (R, \ell)$ of N such that $\ell(\tilde{t}) = t$, $\ell(t') = t'$, and $p \in \ell(\tilde{p})$, then $\ell(\tilde{p}) \subseteq t^\bullet \cup {}^\bullet t'$. The following example shows that this inclusion may be strict.

Example 5.9
The workflow net shown next (below) has a unique process (the net system shown just above it). Place $q \in A^\bullet \cap {}^\bullet D$ is not a boundary place since there is no place $q_R \in A^\bullet \cap {}^\bullet D$ in the corresponding process.



We then conclude with the characterization of α-reconstructibility.

Theorem 5.10. *A workflow net is α-reconstructible if and only if it is an α-workflow net whose places are incomparable for $\sqsubseteq$.*

Proof. Theo. 4.10 shows that the given conditions are sufficient. It remains to show that the absence of short loops and the fact that inner places are boundary places are necessary conditions for α-reconstructibility. First, by definition of the construction α, an inner place p of the synthesized net is a boundary place. The justification is as follows. For every $t \in {}^\bullet p$ and $t' \in p^\bullet$ we have $t \cdot t' \in C_{\mathcal{L}(N)}$ (and also $t' \cdot t \notin C_{\mathcal{L}(N)}$), i.e. there exists some execution sequence of the form $\sigma = u \cdot t \cdot t' \cdot u' \in \mathcal{L}(N)$. The complete process R associated with this execution sequence has transitions $\tilde{t}$, $\tilde{t'}$ and places $\tilde{p}$ that lift respectively t, t', and p; i.e. such that $\ell(\tilde{t}) = t$, $\ell(\tilde{t'}) = t'$, ${}^\bullet\tilde{p} = \{\tilde{t}\}$, $\tilde{p}^\bullet = \{\tilde{t'}\}$, and $p \in \ell(\tilde{p})$. Since $\tilde{t'}$ can fired immediately after $\tilde{t}$ in this process we deduce that $\tilde{p}$ is a non-implicit place of R, hence place p is a boundary place. Now let us assume that N is a workflow net all of whose inner places are boundary places and containing a short loop given by two transitions t_1 and t_2 such that ${t_1}^\bullet \cap {}^\bullet t_2 \neq \emptyset$ and ${t_2}^\bullet \cap {}^\bullet t_1 \neq \emptyset$. Since the places in ${t_1}^\bullet \cap {}^\bullet t_2$ and ${t_2}^\bullet \cap {}^\bullet t_1$ are boundary places we deduce that $t_1 \cdot t_2$ and $t_2 \cdot t_1$ both belong to $C_{\mathcal{L}(N)}$ and thus $t_1 \|_{\mathcal{L}(N)} t_2$. It follows that the sets ${t_1}^\bullet \cap {}^\bullet t_2$ and ${t_2}^\bullet \cap {}^\bullet t_1$ are both empty when these flow relations are taken in the synthesized net $\alpha(N)$ and therefore N is not isomorphic to $\alpha(\mathcal{L}(N))$ hence it is not α-reconstructible. $\square$

6 Conclusion

The starting point of this work was the desire to achieve a characterization of the class of α-reconstructible workflow nets: even though the original presentation by van der Aalst *et al* was restricted to structured workflow nets it was clear that their method could be applied to a much wider class of nets. In particular none of the two properties shared by structured workflow nets, namely the absence of implicit places and a variant of free-choice property, are necessary to guarantee α-reconstructibility. The main condition for α-reconstructibility states that all (inner) places of the workflow net are boundary places. Boundary places and non implicit places coincide in the context of structured workflow nets but these notions diverge in general. By making this distinction explicit we hope to have gained a better understanding of α-reconstructibility. From a practical point of view however this characterization has probably a limited interest. The fact that the obtained conditions are not structural properties, in contrast with structured workflow nets, makes their verification more involved. More specifically we have to associate a workflow net with a finite set of "patterns" from which all of its processes can be generated.

The main goal of α is the exact reconstruction of processes from sets of execution sequences, and α is particularly good at achieving this goal from complete logs (that need not be full logs) of structured workflow nets. Without these assumptions, α may behave in an unexpected way. For instance, the synthesized net may fail to reproduce some of the execution sequences from the input log. When using region-based net synthesis algorithms, one does not make any assumption on W. One just tries to construct for all W the simplest net model that contains all sequences in this set. Sobriety, which is crucial to process reconstruction algorithms, has minor importance in this different perspective. The α and ω algorithms are the two extreme of a range of process mining algorithms.

By resorting only to local information α is more efficient but less expressive than ω. There is potentially room for the design of intermediate mining algorithms.

References

[1] Badouel, E., Darondeau, P.: Theory of regions. In: Reisig, Rozenberg (eds.) [9], pp. 529–586

[2] Badouel, E., Darondeau, P.: Petri Net Synthesis (2013) (Book in preparation)

[3] Busi, N., Pinna, M.G.: Characterizing workflow nets using regions. In: SYNASC, pp. 399–406. IEEE Computer Society (2006)

[4] Busi, N., Pinna, M.G.: Process discovery and petri nets. Mathematical Structures in Computer Science 19(6), 1091–1124 (2009)

[5] Desel, J., Esparza, J.: Free Choice Petri Nets. Cambridge Tracts in Theoretical Computer Science, vol. 40. Cambridge University Press (1995)

[6] Desel, J., Reisig, W.: The Synthesis Problem of Petri Nets. In: Enjalbert, P., Finkel, A., Wagner, K.W. (eds.) STACS 1993. LNCS, vol. 665, pp. 120–129. Springer, Heidelberg (1993)

[7] Ehrenfeucht, A., Rozenberg, G.: Partial (set) 2-structures. part i: Basic notions and the representation problem. Acta Inf. 27(4), 315–342 (1989)

[8] Ehrenfeucht, A., Rozenberg, G.: Partial (set) 2-structures. part ii: State spaces of concurrent systems. Acta Inf. 27(4), 343–368 (1989)

[9] Reisig, W., Rozenberg, G. (eds.): APN 1998. LNCS, vol. 1491. Springer, Heidelberg (1998)

[10] Rozenberg, G., Engelfriet, J.: Elementary net systems. In: Reisig, Rozenberg (eds.) [9], pp. 12–121

[11] van der Aalst, W.M.P.: Process Mining - Discovery, Conformance and Enhancement of Business Processes. Springer (2011)

[12] van der Aalst, W.M.P., Weijters, T., Maruster, L.: Workflow mining: Which processes can be rediscovered? BETA Working Paper Series, WP 74. Eindhoven University of Technology, Eindhoven (2002)

[13] van der Aalst, W.M.P., Weijters, T., Maruster, L.: Workflow mining: Discovering process models from event logs. IEEE Trans. Knowl. Data Eng. 16(9), 1128–1142 (2004)

[14] Winskel, G.: Event Structures. In: Brauer, W., Reisig, W., Rozenberg, G. (eds.) APN 1986. LNCS, vol. 255, pp. 325–392. Springer, Heidelberg (1987)

[15] Winskel, G.: An Introduction to Event Structures. In: de Bakker, J.W., de Roever, W.-P., Rozenberg, G. (eds.) Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency. LNCS, vol. 354, pp. 364–397. Springer, Heidelberg (1989)

On Profiles and Footprints – Relational Semantics for Petri Nets

Matthias Weidlich[1] and Jan Martijn van der Werf[2,⋆]

[1] Technion – Israel Institute of Technology, Haifa, Israel
weidlich@tx.technion.ac.il

[2] Department of Mathematics and Computer Science
Technische Universiteit Eindhoven
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
j.m.e.m.v.d.werf@tue.nl

Abstract. Petri net systems have been successfully applied for modelling business processes and analysing their behavioural properties. In this domain, analysis techniques that are grounded on behavioural relations defined between pairs of transitions emerged recently. However, different use cases motivated different definitions of behavioural relation sets. This paper focusses on two prominent examples, namely behavioural profiles and behavioural footprints. We show that both represent different ends of a spectrum of relation sets for Petri net systems, each inducing a different equivalence class. As such, we provide a generalisation of the known relation sets. We illustrate that different relation sets complement each other for general systems, but form an abstraction hierarchy for distinguished net classes. For these net classes, namely S-WF-systems and sound free-choice WF-systems, we also prove a close relation between the structure and the relational semantics. Finally, we discuss implications of our results for the field of business process modelling and analysis.

1 Introduction

Business process modelling emerged as a means to capture the operations of an organisation. A process model depicts the major activities conducted to achieve a certain goal along with their temporal dependencies [29]. Drivers for process modelling include, among others, the need to establish a shared understanding of the business processes, certification of operations, or process automation.

In practice, business process modelling is often conducted using domain-specific high-level languages, such as BPMN or EPCs, see [29]. For the analysis of process models, however, the Petri net formalism has been successfully employed for over a decade [1,6]. The simple yet powerful formalism is conceptual close to many of the industrial process languages and, in fact, inspired the definition of execution semantics for many of them. Also, the existing theory on the analysis

⋆ 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).

of Petri nets proved to be valuable for answering many of the analysis questions for business process models, cf., [6].

Behavioural analysis of Petri net systems may be grounded on different types of semantics. For instance, the analysis of deadlock freedom of interacting business processes [30,4] suggests to consider the moment of choice, i.e., the state space of a system. Techniques from the field of process mining, in turn, are typically based on trace semantics [21,2,28]. Recently, sets of behavioural relations, which induce relational semantics for Petri nets, have been utilised for analysis, most prominently the behavioural profile [25] and the behavioural footprint [2]. These relations are defined between pairs of transitions and capture behavioural characteristics, or features in the data mining terminology [12], such as order and exclusiveness. The aforementioned notions of relational semantics are conceptual close. Both define relations that allow to represent the behavioural characteristics of a system as a matrix. Yet, they differ with respect to the captured characteristics since different utility considerations led to their definition. Even though the different relations sets proved to be useful in many use cases, their differences have not been thoroughly investigated so far. Insights into their relation and the induced equivalence classes, however, are needed to select a definition that is appropriate for a specific analysis setting.

In this paper, we address this need and make the following contributions. First, we show that the existing notions of profiles and footprints represent different ends of a spectrum of relation sets for Petri net systems. Based on this observation, we provide a generalization of the notion of a relation set. Second, we investigate the expressiveness of different relation sets in this spectrum. We illustrate that those complement each other for general systems. Third, we prove that relation sets form an abstraction hierarchy for classes of workflow (WF-)systems, i.e., S-WF-systems and sound free-choice WF-systems. For these systems, we also establish a link between the net structure and relational semantics. Finally, we elaborate on the implications of our investigations for the application of relational semantics in the field of business process modelling.

The remainder of this paper is structured as follows. The next section presents formal preliminaries. Sec. 3 generalises existing relation sets to obtain a spectrum of relational semantics. Sec. 4 elaborates on relation sets of distinguished net classes. Sec. 5 outlines implications of our work for the application of relation sets. Finally, we review related work in Sec. 6 and conclude in Sec. 7.

2 Preliminaries

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 S and T are *disjoint* if $S \cap T = \emptyset$. A set of sets $U \subseteq \mathcal{P}(S)$ is a *partitioning* of S iff all sets in U are pairwise disjoint and $\bigcup_{X \in U} X = S$.

We denote the Cartesian product of two sets S and T by $S \times T$. A binary relation R from S to T is defined by $R \subseteq (S \times T)$. For $(x, y) \in R$, we also write $x\,R\,y$. For a relation $R \subseteq (S \times T)$, the inverse relation R^{-1} is defined as

$R^{-1} = \{(y, x) \in (T \times S) \mid x\,R\,y\}$. Let $R \subseteq (S \times S)$ be a binary relation over a set S. Relation R is reflexive if $x\,R\,x$ for all $x \in S$. It is irreflexive if $(x, x) \notin R$ for all $x \in S$. It is symmetric if $x\,R\,y$ implies $y\,R\,x$ for all $x, y \in S$, and asymmetric if relation R is not symmetric. The relation is antisymmetric if $x\,R\,y$ and $y\,R\,x$ imply $x = y$ for all $x, y \in S$.

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.

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^*. We write $a \in \sigma$ if a $1 \leq i \leq |\sigma|$ exists such that $\sigma(i) = a$. *Concatenation* of two sequences $\nu, \gamma \in S^*$, denoted by $\sigma = \nu; \gamma$, is a sequence defined by $\sigma : \{1, \ldots, |\nu| + |\gamma|\} \to S$, such that $\sigma(i) = \nu(i)$ for $1 \leq i \leq |\nu|$, and $\sigma(i) = \gamma(i - |\nu|)$ for $|\nu| + 1 \leq i \leq |\nu| + |\gamma|$.

Definition 1 (Petri Net). A *Petri net* is a tuple $N = (P, T, F)$ where P and T are finite disjoint sets of *places* and *transitions*, respectively, and $F \subseteq (P \times T) \cup (T \times P)$ is the *flow relation*. The set of all nodes $P \cup T$ is denoted by N.

For a node $n \in N$, we define its *preset* by $^\bullet n = \{m \mid (m, n) \in F\}$ and its *postset* by $n^\bullet = \{m \mid (n, m) \in F\}$. We lift the notion of presets (postsets) to sequences by $^\bullet\sigma = \bigcup_{n \in \sigma} {}^\bullet n$ ($\sigma^\bullet = \bigcup_{n \in \sigma} n^\bullet$) for $\sigma \in T^*$. A sequence $\pi \in N^*$ of length n is a *path* of N iff $(\pi(i), \pi(i + 1)) \in F$ for all $1 \leq i < n$. The set of all paths of N from node $x \in N$ to node $y \in N$ is denoted by $\Pi(N)_{(x,y)}$. We assume all nets to be connected, i.e., for all Petri nets $N = (P, T, F)$ we assume $\Pi(N)_{(x,y)} \cup \Pi(N)_{(y,x)} \neq \emptyset$ for all nodes $x, y \in N$.

Definition 2 (System, Enabledness, Firing). Let $N = (P, T, F)$ be a Petri net. A *marking* of N is a bag over P, i.e., $m \in \mathbb{N}^P$. A Petri net $N = (P, T, F)$ with corresponding marking $m \in \mathbb{N}^P$ is called a *system*.

Let (N, m) be a system with $N = (P, T, F)$. A transition $t \in T$ is *enabled* in (N, m), denoted by $(N, m)[t\rangle$, if $^\bullet t \leq m$. An enabled transition can *fire*, resulting in a new marking $m' = m - {}^\bullet t + t^\bullet$, and denoted by $(N, m)[t\rangle(N, m')$.

Given a system (N, m_0) with $N = (P, T, F)$, we extend the firing rule to sequences in a standard way. A sequence $\sigma \in T^*$ is a *firing sequence* of (N, m_0) if markings $m_1, \ldots, m_n \in \mathbb{N}^P$ exist such that $(N, m_{i-1})[\sigma(i)\rangle(N, m_i)$ for all $1 \leq i \leq n$, and denoted by $(N, m_0)[\sigma\rangle(N, m_n)$.

The set of all traces from (N, m_0) is defined by $\mathcal{T}(N, m_0) = \{\sigma \in T^* \mid \exists m \in \mathbb{N}^P : (N, m_0)[\sigma\rangle(N, m)\}$, and the set of all *reachable markings* by $\mathcal{R}(N, m_0) =$

$\{m \mid \exists \sigma \in T^*, m \in \mathbb{N}^P : (N, m_0)[\sigma\rangle(N, m)\}$. Two systems (N, m_0) and (N', m_0') are called *trace-equivalent* iff $\mathcal{T}(N, m_0) = \mathcal{T}(N', m_0')$.

A transition $t \in T$ of a system (N, m_0) is *live*, iff for every marking $m \in \mathcal{R}(N, m_0)$ a reachable marking $m' \in \mathcal{R}(N, m)$ exists, such that $(N, m')[t\rangle$. If all transitions of (N, m_0) are live, the system is called *live*.

A place $p \in P$ of a system (N, m_0) is *k-bounded* for some $k \in \mathbb{N}$ iff $m(p) \leq k$ for every reachable marking $m \in \mathcal{R}(N, m_0)$. If all places of (N, m_0) are k-bounded, the system is called *k-bounded*. A system is called bounded if a $k \in \mathbb{N}$ exists such that N is k-bounded.

Let (N, m_0) be a system with $N = (P, T, F)$. The *concurrency relation* $\|_{co} \subseteq N \times N$ [13] contains all pairs of nodes (x, y) that are marked (in case of a place) or enabled (in case of a transition) concurrently in some reachable marking, i.e., $(x, y) \in \|_{co}$ iff a marking $m \in \mathcal{R}(N, m_0)$ exists such that $m \geq m_x + m_y$ with $m_j = [j]$ if $j \in P$ and $m_j = {}^\bullet j$ if $j \in T$. The concurrency relation is symmetric by definition.

On Petri nets, we define the following subclasses. A Petri net $N = (P, T, F)$ is called an *S-net* iff $|{}^\bullet t| \leq 1$ and $|t^\bullet| \leq 1$ for all transitions $t \in T$. Net N is called a *T-net* iff $|{}^\bullet p| \leq 1$ and $|p^\bullet| \leq 1$ for all places $p \in P$. It is called *free choice* iff ${}^\bullet t_1 \cap {}^\bullet t_2 \neq \emptyset$ implies ${}^\bullet t_1 = {}^\bullet t_2$ for all transitions $t_1, t_2 \in T$. Given a system (N, m), if N is an S-net (free-choice net), we call the system an S-system (FC-system)

A special subclass of Petri nets is the class of workflow nets. A Petri net $N = (P, T, F)$ is a *workflow net* (WF net) if two places $i, f \in P$ exist such that ${}^\bullet i = f^\bullet = \emptyset$ and all nodes are on a path from i to f, i.e., for each node $n \in N$, a path $\pi \in \Pi(N)_{(i,f)}$ exists such that $n \in \pi$. Place i is called the *initial place* of N, place f is called the *final place* of N. We define the short-circuited net of WF net N by $\bar{N} = (P, T \cup \{\bar{t}\}, F \cup \{(f, \bar{t}), (\bar{t}, i)\})$. WF net N is called *sound* if the system $(\bar{N}, [i])$ is live and bounded.

3 A Spectrum of Relational Semantics

This section outlines a spectrum of relational semantics, induced by a spectrum of sets of relations defined over pairs of transitions. We first give an overview of this spectrum in Sec. 3.1, before Sec. 3.2 presents the formal definition of parametrised relation sets. Sec. 3.3 elaborates on equivalences based on relation sets.

3.1 Overview

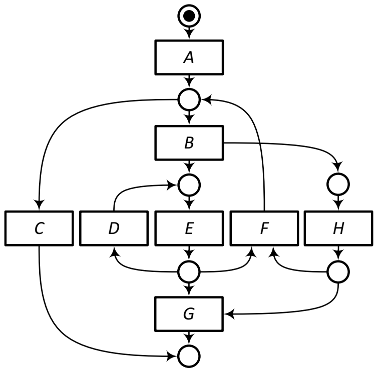
A first set of behavioural relations was presented as part of the α-mining algorithm [7,2]. It aims at the construction of a workflow net system from sequences of observed transition occurrences. To this end, it exploits direct successorship of transition occurrences, i.e., a *directly follows* relation. This relation comprises pairs of transitions that succeed each other. Using this relation, the α-algorithm defines three relations, $\#$ (or $+$ to harmonise notation), $\rightarrow$, and $\|$, over pairs of transitions. Membership of a transition pair for one of these relations is determined using the directly follows relation: two transitions may never follow each other ($+$), follow each other in one direction ($\rightarrow$), or in both directions ($\|$). As such, the relations (with relation $\rightarrow^{-1}$) partition the Cartesian product of the observed transitions. These relations are jointly referred to as a *footprint* [2]. Although proposed for sequences of observed transition occurrences, the relations may be derived for a net system based on all traces.

A similar, yet different set of behavioural relations was presented in [25], dubbed *behavioural profile*. The behavioural profile is grounded on the notion of *weak order*. This relation captures whether a transition is eventually succeeded by another transition. Again, three derived relations ($+$, $\rightarrow$, and $\|$ using the same notation) are constructed by investigating whether two transitions never occur together ($+$), are always ordered if they occur together ($\rightarrow$), or may occur in any order ($\|$). Together with the inverse of the order relation, one obtains a partitioning of the Cartesian product of transitions.

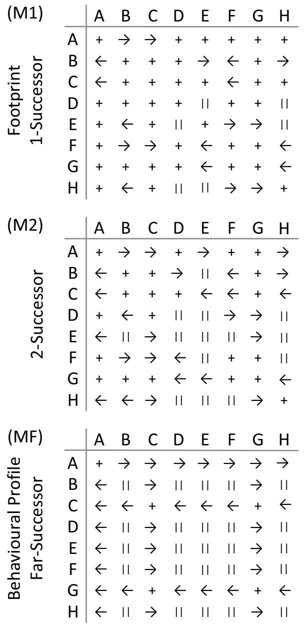
The footprint relations have been defined in the context of process mining. Here, the direct successorship of transitions according to the behavioural relations translates into a structural successorship during the construction of a net system. Behavioural profiles, in turn, have been motivated by the analysis of process model consistency. Models that shall be analysed for consistency typically show only a partial functional overlap, i.e., a certain share of transitions of one net system is without counterpart in the other system. Consistency measurement that is insensitive to such model extensions, therefore, is grounded on indirect behavioural dependencies as captured by the behavioural profile. With the same motivation, behavioural profiles have been applied for change propagation [27], process model abstraction [22], and the derivation of reusable modelling blocks [23].

Besides their differences, the footprint and the profile are conceptually close. Both adopt a binary base relation for transitions that requires the existence of a firing sequence that contains the respective transitions. The transitions are either required to succeed each other with no or an arbitrary number of transitions occurring in between. The difference, therefore, lies in the *look ahead* assumed to build the base relation. For illustration, consider the net system depicted in Fig. 1(a). Matrix $M1$ in Fig. 1(b) depicts the relational semantics induced by the footprint. It holds $D + F$, i.e., both transitions never succeed each other directly. In the footprint, relation $+$ captures transitions that never succeed each other, whereas $\|$ captures concurrent enabling (e.g., $E \| F$) and control flow cycles of length one or two (e.g., $D \| E$). Matrix MF, in turn, shows the profile. Here, it holds $D \| F$ since the transitions may appear in either order. In the profile, relation $+$ captures transitions that never occur together in any trace (e.g., $C + G$). Relation $\|$ captures concurrent enabling and control flow cycles of *any* length.

Both relation sets span a spectrum of relational semantics, which is obtained by step-wise increasing the assumed look ahead when constructing the base relation. We exemplify this spectrum by matrix $M2$. Here, the relations are derived from a *2-successor relation* that holds if two transitions follow each other



(a) An example net system.



(M1)

Footprint 1-Successor

	A	B	C	D	E	F	G	H
A	+	→	→	+	+	+	+	+
B	←	+	+	+	→	←	+	→
C	←	+	+	+	+	←	+	+
D	+	+	+	+	||	+	+	||
E	+	←	+	||	+	→	→	||
F	+	→	→	+	←	+	+	←
G	+	+	+	+	←	+	+	←
H	+	←	+	||	||	→	→	+

(M2)

2-Successor

	A	B	C	D	E	F	G	H
A	+	→	→	+	→	+	+	→
B	←	+	+	→	||	←	+	→
C	←	+	+	+	←	←	+	←
D	+	←	+	||	||	→	→	||
E	←	||	→	||	||	||	→	||
F	+	→	→	←	||	+	+	||
G	+	+	+	←	←	+	+	←
H	←	←	→	||	||	||	→	+

(MF)

Behavioural Profile Far-Successor

	A	B	C	D	E	F	G	H
A	+	→	→	→	→	→	→	→
B	←	||	→	||	||	||	→	||
C	←	←	+	←	←	←	+	←
D	←	||	→	||	||	||	→	||
E	←	||	→	||	||	||	→	||
F	←	||	→	||	||	||	→	||
G	←	←	+	←	←	←	+	←
H	←	||	→	||	||	||	→	||

(b) Three different relational semantics.

Fig. 1. Overview of different relational semantics

directly or with just a single transition occurring in between. In this matrix, we observe $D \rightarrow F$. There exists a trace in which D is succeeded by E and F. However, F is succeeded by D only once at least two transitions, B and E, have occurred.

3.2 Parametrised Relation Sets

In order to obtain a formalisation of the spectrum of relation sets, we first parametrise the base relation. The up-to-k successor relation holds between two transitions, if there exists a trace in which both transitions occur with at most $k-1$ transitions in between.

Definition 3 (k-Successor, up-to-k-Successor, minimal k-Successor). Let (N, m_0) be a system, let $T' \subseteq T$ be a set of transitions, and let $k \in \mathbb{N}$. The *k-successor relation* $\rhd_k \subseteq T' \times T'$ is defined by:

$$x \rhd_k y \Leftrightarrow \exists \sigma \in \mathcal{T}(N, m_0), 1 \leq i \leq |\sigma| : \sigma(i) = x \wedge \sigma(i+k) = y$$

The *up-to-k-successor relation* $>_k \subseteq T' \times T'$ is defined by:

$$x >_k y \Leftrightarrow \exists 1 \leq l \leq k : x \rhd_l y$$

The *minimal k-successor relation* $\trianglerighteq_k \subseteq T' \times T'$ is defined by:

$$x \trianglerighteq_k y \Leftrightarrow x \vartriangleright_k y \wedge (x,y) \notin >_{k-1}$$

Directly from the definition of the concurrency relation, if two transitions occur in the concurrency relation, then they are direct successors.

Proposition 4. Let (N, m_0) be a system, and let $x, y \in T$ be two transitions. If $(x, y) \in \|_{co}$ then $x >_1 y$.

Using the parametrised successor relation, we obtain parametrised relation sets.

Definition 5 (k-Relation set). Let $S = (N, m_0)$ be a system, let $T' \subseteq T$ be a set of transitions, and let $k \in \mathbb{N}$. Given a pair of transitions $(x, y) \in T' \times T'$, we define the k-exclusiveness relation $+_k \subseteq T' \times T'$, the k-order relation $\rightarrow_k \subseteq T' \times T'$ and the k-disorder relation $\|_k \subseteq T' \times T'$ by:
- ◦ $x +_k y$, iff $(x, y) \notin >_k$ and $(y, x) \notin >_k$;
- ◦ $x \rightarrow_k y$, iff $(x, y) \in >_k$ and $(y, x) \notin >_k$;
- ◦ $x \|_k y$, iff $(x, y) \in >_k$ and $(y, x) \in >_k$.

The k-relation set of S over T' is defined as a 3-tuple $\mathcal{S}_k^{T'}(N, m_0) = \{+_k, \rightarrow_k, \|_k\}$. If $T' = T$, we omit the superscript. We overload set comparison operators on relation sets by pairwise comparing the elements of the relation sets.

According to this definition, the footprint of a net system corresponds to its 1-relation set. We observe that the properties proved for footprints, see [2], hold also true for parametrised relation sets.

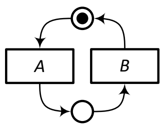
Property 6. Let (N, m_0) be a system, let $k \in \mathbb{N}$ and let $T' \subseteq T$ be a subset of transitions. Then for $\mathcal{S}_k^{T'}(N, m_0) = \{+_k, \rightarrow_k, \|_k\}$, it holds
(1) relation $\rightarrow_k$ is antisymmetric and irreflexive;
(2) relations $+_k$ and $\|_k$ are symmetric;
(3) $+_k$, $\rightarrow_k$ and $\|_k$ are pairwise disjoint; and
(4) $+_k$, $\rightarrow_k$, $\rightarrow_k^{-1}$ and $\|_k$ is a partitioning of $T' \times T'$.

All properties follow directly from the definition of the relations based on the up-to-k-successor relation.

The parametrisation of relation sets induces an unbounded number of relational semantics for net systems. However, we observe that once a certain bound is reached, relation sets for higher parameters are all equal. We characterise this successor bound as follows.

Definition 7 (Successor bound). For a net system (N, m_0), the *successor bound* $b \in \mathbb{N}$ is the smallest number satisfying
- ◦ $\mathcal{S}_b(N, m_0) = \mathcal{S}_k(N, m_0)$ for all $b \leq k$; and
- ◦ $\mathcal{S}_k(N, m_0) \subset \mathcal{S}_b(N, m_0)$ for all $k < b$.

Proposition 8. Given a net system (N, m_0) there exists a unique successor bound with $b \leq |T|^2$.



(a) System S

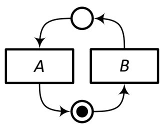
	A	B
A	+	||
B	||	+

1-Relation Set

	A	B
A	||	||
B	||	||

2-Relation Set

(b) Relation sets of S and S'



(c) System S'

Fig. 2. Systems S and S' are k-equivalent for any $k \in \mathbb{N}$, but not trace equivalent

Proof. Follows from the inductive definition of $>_k$ and $\rhd_k$, and Prop. 6(4). $\square$

The successor bound for the relation sets of a net system is related to the notion of a behavioural profile. Apparently, the k-relation set with k being the successor bound coincides with the profile of a net system.

3.3 Equivalence of Relation Sets

Parametrised relation sets induce a number of equivalences. We first consider two types of equivalences. Two system may either show equivalent k-relation sets, or they even agree on all relation sets down to a certain boundary.

Definition 9 (k-Equivalent, down-to-k-equivalent). Let $S = (N, m_0)$ and $S' = (N', m_0')$ be two systems, and let $\mathcal{S}_k(N, m_0) = \{+_k, \rightarrow_k, \|_k\}$ and $\mathcal{S}_k(N', m_0') = \{+_k', \rightarrow_k', \|_k'\}$ be their respective k-relation sets.

- ○ Systems S and S' are *k-equivalent*, denoted by $S \equiv_k S'$, iff $+_k = +_k'$, $\rightarrow_k = \rightarrow_k'$ and $\|_k = \|_k'$.
- ○ Systems S and S' are *down-to-k-equivalent*, denoted by $S \equiv_{\downarrow k} S'$, iff $S \equiv_l S'$ for all $l \in \mathbb{N}$ with $k \leq l$.

Proposition 10. Relations $\equiv_k$ and $\equiv_{\downarrow k}$ are equivalences.

Proof. Relations $\equiv_k$ and $\equiv_{\downarrow k}$ are reflexive. Transitivity and symmetry follow directly from the set equivalences. $\square$

The relation sets are deduced from the up-to-k-successor relation, which formulates statements on the existence of a trace. As a consequence, net systems that show equal sets of traces show equal relation sets for all parameters.

Proposition 11. Let $S = (N, m_0)$ and $S' = (N', m_0')$ be two systems that are trace equivalent. Then $S \equiv_{\downarrow 0} S'$.

Proof. Follows directly from the definition of $>_k$. $\square$

The opposite, however, does not hold. Consider Fig. 2. Systems S and S' are down-to-1-equivalent, i.e., the systems are k-equivalent for any $k \in \mathbb{N}$. However, as $\langle A, B, A \rangle$ is a firing sequence of S, but not of S', they are not trace equivalent.

Turning the focus on the relation between equivalence of relation sets for different parameters, we observe the following: Since relation sets beyond the successor bound do not change, equivalence for one parameter above this bound implies equivalence for all parameters above the bound.

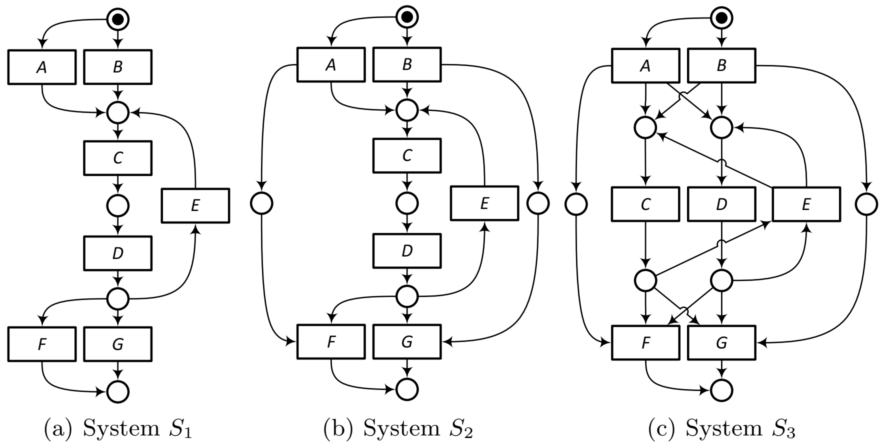


Fig. 3. Equivalences based on different relation sets are incomparable; systems S_1 and S_2 are 1-equivalent, but not 7-equivalent; whereas S_2 and S_3 are 7-equivalent, but not 1-equivalent

Theorem 12. *Let S and S' be two systems. Let b be the successor bound of S, and let b' be the successor bound of S'. Let $\bar{b} = \max\{b, b'\}$. If $S \equiv_{\bar{b}} S'$ then $S \equiv_{\downarrow\bar{b}} S'$.*

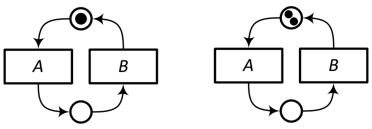
Proof. The proof follows directly from the definition of $\equiv_{\downarrow k}$ and Prop. 8. $\square$

In general, equivalences based on different relation sets are incomparable. Consider for example the systems in Fig. 3. Systems S_1 and S_2 are 1-equivalent but not 7-equivalent. Likewise, systems S_2 and S_3 are 7-equivalent, but not 1-equivalent.

The example systems given in Fig. 2 illustrated already that different initial markings of a system may not be distinguished by relation sets. This is due to the fact that relation sets capture only the dependencies between transitions in terms of their minimal distance in any trace. However, they do not provide any notion of a *start* of a trace. To countervail this effect, we present two more equivalences. Those extend the given equivalences with the requirement of equal sets of initially enabled transitions.

Definition 13 (Start-k-equivalent, start-down-to-k-equivalent). Let $S = (N, m_0)$ and $S' = (N', m_0')$ be two systems. Let $T_0 = \{t \in T \mid (N, m_0)[t\rangle\}$ and $T_0' = \{t \in T' \mid (N', m_0')[t\rangle\}$ be the transitions enabled in the initial markings of both systems.

- Systems S and S' are *start-k-equivalent*, denoted by $S \equiv_k^s S'$ if $T_0 = T_0'$ and $S \equiv_k S'$.
- Systems S and S' are *start-down-to-k-equivalent*, denoted by $S \equiv_{\downarrow k}^s S'$, iff $S \equiv_l^s S'$ for all $l \in \mathbb{N}$ with $k \leq l$.



(a) System S (b) System S'

Fig. 4. Systems S and S' are start-k-equivalent for any $k \in \mathbb{N}$, but not trace equivalent

Proposition 14. Relations $\equiv^s_k$ and $\equiv^s_{\downarrow k}$ are equivalences.

Proof. Relations $\equiv^s_k$ and $\equiv^s_{\downarrow k}$ are reflexive. Transitivity and symmetry follow directly from the set equivalences. □

As for the initially presented equivalences, the successor bound allows to draw conclusions on equivalence of relation sets for different parameters.

Proposition 15. Let $S = (N, m_0)$ and $S' = (N', m'_0)$ be two systems that are start-down-to-k-equivalent. Then, $S \equiv_{\downarrow k} S'$.

Proof. Follows from the definition of $\equiv^s_{\downarrow k}$. □

Also for these equivalences, however, the systems shown in Fig. 3 illustrate that this result does not hold in the general case.

Finally, we observe that, again, trace equivalence implies the equality of relation sets for all parameters and equality of sets of initially enabled transitions.

Proposition 16. Let $S = (N, m_0)$ and $S' = (N', m'_0)$ be two systems that are trace equivalent. Then $S \equiv^s_{\downarrow 0} S'$.

Proof. Follows from Prop. 11 and the definition of $\equiv^s_{\downarrow k}$. □

In contrast to k-equivalence, start-down-to-k-equivalence distinguishes the systems given in Fig. 2. However, even start-down-to-k-equivalence does not imply trace equivalence for general net systems, as illustrated by the systems in Fig. 4. Both systems are start-k-equivalent for any $k \in \mathbb{N}$, but $\langle A, A \rangle$ is a firing sequence of S, but not of S'. Thus, they are not trace equivalent.

4 Relation Sets for Distinguished Net Classes

This section investigates relation sets for distinguished classes of net systems, namely S-WF-systems and free-choice WF-systems. S-WF-systems provide a rather simple class of net systems, since they do not exhibit concurrency. As such, the structure of an S-WF-net is equivalent to its reachability graph. Further, the results for free-choice WF-systems imply those for S-WF-systems. Despite their simplicity and containment in the class of free-choice WF-systems, we first consider S-WF-systems to illustrate the investigated aspects. That is, first, the derivation of relation sets from the structure of the net system is shown. Second, we elaborate on the abstraction of a k-relation set, which yields a $k+1$-relation set. Finally, we investigate the equivalence class induced by relation sets for the respective net systems.

4.1 S-WF-Systems

Derivation. In an S-WF-system, i.e., workflow systems that are also an S-net, we observe a close relation between the length of a directed path between two transitions and the fact whether they are k-successors. The reason for this close relation is the absence of concurrency in S-WF-systems.

Property 17. For a S-WF-system it holds $\|_{co} = \emptyset$.

This property which directly follows from the structure of S-WF-systems, allows for the following structural characterisation of k-successorship.

Lemma 18. *Let $S = (N, m_i)$ be an S-WF-system. Then, $x \rhd_k y$ iff there is a path $\pi \in \Pi(N)_{(x,y)}$ with $k = |\pi| - 1$.*

Proof. Marking m_i marks one place, so do all markings $m \in \mathcal{R}(N, m_i)$.
($\Rightarrow$) Let $x \rhd_k y$. Then, there exists a trace $\sigma \in \mathcal{T}(N, m_i)$ containing x and y, and $k-1$ transitions between them. Since $\|_{co} = \emptyset$, the $k-1$ transitions form a directed path.
($\Leftarrow$) Let there be a directed path comprising $k-1$ transitions between x and y. Since every transition is on a path from the initial to the final place, all transitions have exactly one place in the preset and one place in the postset. Each reachable marking marks one place, hence, the $k-1$ transitions may be fired in the respective order in any marking enabling x, so that $x \rhd_k y$. □

From the above, it follows that the derivation of a k-relation set requires only knowledge on the length of shortest directed paths between all transitions. Those may be determined in low polynomial time to the size of the system.

Theorem 19. *For any S-WF-system holds, any k-relation set is computed from its graph distance matrix.*

Proof. Follows directly from Lm. 18. □

Corollary 20. *Let (N, m_0) be a S-WF-system, and let $k \in \mathbb{N}$. The k-relation set can be calculated in $O(|N|^3)$.*

Proof. Follows from Thm. 19 and the fact that the shortest directed paths between all nodes of a directed graph with N nodes are determined in $O(|N|^3)$ time [24]. □

Abstraction. We introduce a notion of abstraction to describe the interplay between different relation sets of a single net system. Abstraction aims at deriving the $(k+1)$-relation set from a k-relation set using the system structure. To this end, it extends the underlying successor relation. Since S-WF-systems do not show concurrency, abstraction can be done in a sequential way as follows.

Definition 21 (Sequential abstraction). Let (N, m_0) be a system with $N = (P, T, F)$, $>_k \subseteq T \times T$ an up-to-k successor relation. Then, the *sequential abstraction* of $>_k$, denoted by $\alpha^S(>_k) \subseteq T \times T$, is defined by $(x, z) \in \alpha^S(>_k)$, iff $x >_k z$ or a transition $y \in T$ exists with $x >_k y$ and $z \in p^\bullet$ for some $p \in y^\bullet$.

To show that abstraction of a k-relation set indeed yields the $(k+1)$-relation set, we first prove an auxiliary result. It states that structural precedence hints at the enabling of a transition in a certain marking.

Proposition 22. Let (N, m_0) be an S-WF-system with $N = (P, T, F)$. Let $\sigma \in T^*$ and $m \in \mathbb{N}^P$ such that $(N, m_0)[\sigma\rangle(N, m)$. Then $t \in p^\bullet$ for some $p \in \sigma(|\sigma|)^\bullet$ iff $(N, m)[t\rangle$ for any $t \in T$.

Proof. Follows from the structure of S-nets and the boundedness theorem [8]. $\square$

Using this result, we can prove that abstraction indeed allows for generalising relation sets of S-WF-systems.

Proposition 23. Let (N, m_0) be an S-WF-system, and let $k \in \mathbb{N}$ such that $k > 0$. Then $\alpha^S(>_k) = >_{k+1}$.

Proof. Define $N = (P, T, F)$. It suffices to consider the case of $\triangleright_k$, as it implies the result for $>_k$. Let $x, z \in T$ such that $x \triangleright_k z$. Then, a trace $\sigma \in \mathcal{T}(N, m_0)$ and marking $m \in \mathbb{N}^P$ exist with $(N, m_0)[\sigma\rangle(N, m)$ and $\sigma(|\sigma| - k) = x$ and $\sigma(|\sigma|) = z$.
$(\Rightarrow)$ Consider a transition $y \in p^\bullet$ for some $p \in z^\bullet$. By Prop. 22, y may be fired in m, which yields $x \triangleright_{k+1} y$.
$(\Leftarrow)$ Assume σ is extended by firing a transition y in m. Then, $y \in p^\bullet$ for some $p \in z^\bullet$ by Prop. 22. $\square$

Equivalence. Turning the focus on the equivalence classes induced by relation set for S-WF-systems, we observe that those form a hierarchy. A smaller parameter for the relation set yields a stricter equivalence.

Theorem 24. *Let S and S' be two S-WF-systems. If $S \equiv_1^s S'$, then $S \equiv_k^s S'$, for any $k \in \mathbb{N}$ with $k > 0$.*

Proof. We prove the statement by showing that if $S \equiv_l^s S'$ for all $l \leq k$, then $S \equiv_{k+1}^s S'$. Let $(x, y) \in \alpha^S(>_k)$. Then either $(x, y) \in >_k$ or a $z \in T$ and $l < k$ exists with $(x, z) \in >_l$ and $z >_1 y$. Since $S \equiv_l^s S'$ and $S \equiv_1^s S'$, both $(x, y) \in >'_l$ and $z >'_1 y$. Hence, $(x, y) \in \alpha^S(>'_k)$. By Prop. 23, we have $>_{k+1} = >'_{k+1}$. Hence, $S \equiv_{k+1}^s S'$. $\square$

Finally, we show that 1-relation sets provide a complete characterisation of trace semantics for S-WF-systems. Hence, start-down-to-k-equivalence coincides with trace equivalence.

Theorem 25. *Let $S = (N, m_i)$ and $S' = (N', m'_i)$ be S-WF-systems. Then, S and S' are trace equivalent, iff $S \equiv_{\downarrow 1}^s S'$.*

Proof. Follows directly from Thm. 24, Prop. 22 and Prop. 16. $\square$

4.2 Free-Choice WF-Systems

Free-choice Petri nets [8] is a well-studied subclass of Petri nets, for which many nice theoretical results and efficient algorithms exist. Many behavioural properties of free-choice nets are decidable based on the structure of the net, like well-formedness. In addition, free-choice nets are an important class for modelling business processes, since the essentials of common process description languages can be traced back to free-choice nets (exceptions include OR-joins and error handling) [17]. As an example, the BIT process library[1] contains 732 unique process models that all correspond to free-choice nets. In this section, we show that for free-choice nets, the up-to-k-successor can be decided on the structure of the net as well.

Derivation. For free-choice nets, we derive the up-to-k-successor using the minimal k-successor, i.e., the minimal k for which $x \rhd_k y$ holds. Based on the structure of free-choice nets, we introduce the minimal structural successor function (MSS). The MSS is a structural measure to calculate the number of steps needed to enable or mark a node y from a node x.

Definition 26 (Minimal structural successor). Let (N, m_0) be a system with $N = (P, T, F)$. The *minimal structural successor mss* : $N \times N \mapsto \mathcal{P}(N)$ assigns sets of nodes to pairs of nodes $x, y \in N$ as follows:

$$mss(x,y) = \begin{cases} \emptyset & \text{if } (x,y) \notin F^*, \\ \{x\} & \text{if } xF^*y,\ y \in x^\bullet, \\ \{x\} \cup \bigcup_{v \in x^\bullet, (v,y) \notin \|_{co}} mss(v,y) & \text{if } xF^*y,\ y \notin x^\bullet,\ x \in T, \\ \{x\} \cup mss(v,y) & \text{if } xF^*y,\ y \notin x^\bullet,\ x \in P,\ v \in x^\bullet, \\ & \ |mss(v,y)| = \min_{v \in x^\bullet} |mss(v,y)| \end{cases}$$

where F^* denotes the transitive closure of F.

For live and bounded free-choice systems, the minimal structural successor and the minimal k successor coincide, which means that we can compute the up-to-k successor in polynomial time. To prove this, we first show that a marking is reachable in which all necessary places needed to fire the transitions given by the minimal structural successor, i.e., places that are in the preset of these transitions, but not in their postset.

Proposition 27. Let (N, m_0) be a live and bounded system with $N = (P, T, F)$ free-choice. Let $x, y \in T$ such that $(x, y) \notin \|_{co}$ and $m_0(p) = 0$ for all places $p \in P \cap mss(x, y)$. Then a reachable marking $m \in \mathcal{R}(N, m_0)$ exists with $m \geq \sum_{p \in {}^\bullet U \setminus U^\bullet} [p]$ with $U = T \cap mss(x, y)$.

Proof. Since (N, m_0) is live, a marking $m \in \mathcal{R}(N, m_0)$ and firing sequence $\gamma \in (T \setminus U)^*$ exist such that $(N, m_0)[\gamma\rangle(N, m)$ and $(N, m)[x\rangle$, i.e., firing sequence γ enables transition x for the first time.

[1] http://www.zurich.ibm.com/csc/bit/downloads.html (last accessed March 26, 2012).

Define $Q(m) = \{p \in {}^\bullet\nu \setminus \nu^\bullet \mid m(p) > 0\}$, i.e., $Q(m)$ is the set of places marked in m needed to fire a transition of U, but not produced by any transition of U. Let $p \in {}^\bullet U \setminus (U^\bullet \cup {}^\bullet x) \setminus Q(m)$. As p is not in the postset of any transition in $mss(x,y)$, we have $(x,p) \in \|_{co}$. Since (N, m_0) is live, a firing sequence $\tau \in T^*$ and marking $m' \in \mathbb{N}^P$ exist such that $(N,m)[\tau\rangle(N,m')$, $m' \geq {}^\bullet x + [p]$ and $x \notin \tau$, since if x was needed to mark p, then $p \in mss(x,y)$. Suppose $Q(m) \cap {}^\bullet\tau \neq \emptyset$, i.e., some transition $u \in \tau$ consumes from some place $q \in Q(m)$. By the free-choice property, a transition $v \in mss(x,y)$ should then be enabled as well. However, x has not fired in $\gamma;\tau$, thus v cannot be enabled, and hence, u cannot be enabled as well. Thus, $Q(m) \cap {}^\bullet\tau = \emptyset$, and $Q(m') = Q(m) \cup \{q\}$. □

The above proposition shows the existence of a marking m in a free-choice system that generates sufficient tokens in order to fire the transitions of the MSS, as shown in the next proposition.

Proposition 28. Let (N, m_0) be a live and bounded system with $N = (P, T, F)$ is free-choice. Let $x, y \in T$ such that $(x,y) \notin \|_{co}$ and $m_0(p) = 0$ for all places $p \in P \cap mss(x,y)$. Then $x \rhd_k y$ with $k = |T \cap mss(x,y)|$.

Proof. We prove the statement by showing the existence of a firing sequence $\sigma \in \mathcal{T}(N, m_0)$ such that $\sigma(i) = x$ and $\sigma(i+k) = y$ for some $1 \leq i \leq |\sigma|$. We define relation $\sqsubset \subseteq T \times T$ by $a \sqsubset b$ if a $p \in mss(x,y) \cap P$ exists such that $\{(a,p),(p,b)\} \in F$, and $a \sqsubseteq b$ if either $a = b$ or a $c \in T \cap mss(x,y)$ exists such that $a \sqsubseteq c$ and $c \sqsubset b$. By definition, $\sqsubseteq$ is a partial order. Let $\nu \in T^*$ such that $\nu(i) \sqsubseteq \nu(j)$ for all $1 \leq i \leq j \leq |\nu|$ and $t \in \nu$ iff $t \in mss(x,y)$.

By Prop. 27, a firing sequence $\mu \in \mathcal{T}(N, m_0)$ and marking $m \in \mathbb{N}^P$ exist such that $(N, m_0)[\mu\rangle(N,m)$ and $m \geq \sum_{p \in {}^\bullet\nu \setminus \nu^\bullet} [p]$. Next, we show that ν is a firing sequence of (N, m). We prove this by induction on the length of ν.

Since $\nu(1) = x$, we have ${}^\bullet\nu(1) \subseteq \{p \in P \mid p \in {}^\bullet\nu \setminus \nu^\bullet\}$. Hence, ${}^\bullet\nu(1) \leq m$.

Now, suppose $\nu = \nu';\nu''$ with $|\nu''| > 0$, and suppose a marking m' exists such that $(N,m)[\nu'\rangle(N,m')$. By construction of ν, $\nu'^\bullet \cap {}^\bullet u \neq \emptyset$. If ${}^\bullet u \subseteq \nu'^\bullet$, transition u is enabled in m'. Otherwise, a $p \in {}^\bullet u \setminus \nu'^\bullet$ exists. By the construction of ν, we have $m(q) > 0$.

Hence, firing sequence $\sigma = \mu;\nu$ has the desired property. □

Using this result, we establish the relation between the minimal k-successor relation and the minimal structural successor as follows.

Lemma 29. *Let (N, m_0) be a live and bounded system with $N = (P, T, F)$ free-choice. Let $x, y \in T$ such that $(x,y) \notin \|_{co}$ and $m_0(p) = 0$ for all places $p \in P \cap mss(x,y)$. Then $x \unrhd_k y$ iff $|mss(x,y) \cap T| = k$.*

Proof. ($\Rightarrow$) Suppose $x \unrhd_k y$ for some $k \in \mathbb{N}$. Then a firing sequence $\sigma \in \mathcal{T}(N, m_0)$ exists with an $1 \leq i \leq |\sigma|$ such that $\sigma(i) = x$ and $\sigma(i+k) = y$, and for all firing sequences $\tau \in \mathcal{T}(N, m_0)$ and $1 \leq i, j, \leq n$ such that $\tau(i) = x$ and $\tau(j) = y$, then $j - i \geq k$. As a consequence, the sequence $\langle\sigma(i), \ldots, \sigma(i+k)\rangle$ is cycle free, and each transition $\sigma(j)$ for $i \leq j \leq i+k$ is needed in order to enable y, since otherwise $x \leq_{k-1} y$. Hence, $(\sigma(j), y) \notin \|_{co}$ for all $i \leq j \leq i+k$. Thus, $\sigma(i) \in mss(x,y)$ for $i \leq j \leq i+k$, and $|mss(x,y) \cap T| \geq k$.

Suppose a $t \in mss(x,y) \cap T$ exists and no $i \leq j \leq i+k$ exists with $\sigma(j) = t$. Then $(t,y) \notin \|_{co}$ and firing y depends on firing transition t, i.e., a $j < i$ should exist with $\sigma(j) = t$, which is not possible since transition x needs to fire in order to enable t. Hence, such a transition does not exist and $|mss(x,y) \cap T| = k$.

($\Leftarrow$) Suppose $|mss(x,y) \cap T| = k$. To prove $x \trianglerighteq_k y$, we need to show $x \vartriangleright_k y$ and $(x,y) \notin >_{k-1}$. By Prop. 28, we have $x \vartriangleright_k y$.

Since $(x,y) \notin \|_{co}$, no marking exists in which both x and y are enabled. Suppose some place $p \in P \cap mss(x,y)$ is marked by some firing sequence not containing x, i.e., a firing sequence $\gamma \in \mathcal{T}(N, m_0)$ and marking $m \in \mathbb{N}^P$ exist with $(N : m_0 \xrightarrow{\gamma} m)$, $x \notin \gamma$ and $m(p) > 0$. Since (N, m_0) is live, a firing sequence $\sigma \in T^*$ and marking $m' \in \mathbb{N}^P$ exist such that $(N,m)[\sigma\rangle(N,m')$, $x \notin \sigma$ and $(N,m)[x\rangle$. Then $p \in \sigma$, since otherwise place p would be unbounded in (N, m_0). Thus, $m'(p) = 0$. Thus, all places $p \in P \cap mss(x,y)$ are empty when x fires.

Hence $(x,y) \notin >_{k-1}$. □

We conclude that the minimal structural successor suffices to compute the k-relation set of a live and bounded FC-system and, therefore, sound free-choice WF-systems.

Theorem 30. *Let (N, m_0) be a live and bounded FC-system. Then for any $k \in \mathbb{N}$, the k-relation set can be computed from its concurrency relation and its minimal structural successor.*

Proof. Follows from Prop. 4 and Lm. 29. □

Corollary 31. *Let (N, m_0) be a live and bounded FC-system, then for any $k \in \mathbb{N}$ with $k > 0$, the k-relation set can be calculated in $O(|N|^3)$ time.*

Proof. By Thm. 30, calculating the k-relation set for a live and bounded FC-systems can be done from its concurrency relation and the minimal successor length. The concurrency relation is computed in $O(|N|^3)$ time [13,11], the construction of the minimal structural successor also takes $O(|N|^3)$ time. □

Abstraction. As illustrated for S-systems, also for sound free-choice WF -systems there exists an abstraction operation for relation sets. Due to potential concurrency, the sequential abstraction introduced earlier cannot be applied, though. Therefore, we define a more generic abstraction operation based on the notion of the MSS.

Definition 32 (Abstraction). Let (N, m_0) be a system, $\|_{co} \subseteq T \times T$ its concurrency relation, and let $>_k \subseteq T \times T$ be an up-to-k successor relation. Then, the *abstraction* of $>_k$, denoted by $\alpha(>_k) \subseteq T \times T$, is defined by $(x,y) \in \alpha(>_k)$ iff $x >_k y$ or $k = |mss(x,y) \cap T| - 1$.

Abstraction, indeed, allows for deriving the $(k+1)$-relation set from the k-relation set for live and bounded free-choice systems and, therefore, for sound free-choice

WF-systems. Sequential abstraction is a special case of abstraction for free-choice nets, as in S-systems, the MSS equals the shortest path between x and y.

Proposition 33. Let (N, m_0) be a live and bounded system with $N = (P, T, F)$ free-choice. Let $x, y \in T$ be two transitions such that $(x, y) \notin \|_{co}$, and $m_0(p) = 0$ for all places $p \in P \cap mss(x, y)$. Then $(x, y) \in \alpha(>_k)$ iff $(x, y) \in >_{k+1}$.

Proof. Define $l = |mss(x, y) \cap T|$.
($\Rightarrow$) Suppose $(x, y) \in \alpha(>_k)$. Then $(x, y) \in >_k$ or $k = 1 + |mss(x, y) \cap T|$. In the first case, also $(x, y) \in >_{k+1}$. Suppose not $(x, y) \in >_k$. Then $l = k - 1$ By Lm. 29, $x \trianglerighteq_l y$, i.e., $x \notin >_{l-1}$ and $x \triangleright_l y$. Hence, $x \triangleright_{k+1} y$.
($\Leftarrow$) Suppose $(x, y) \in >_{k+1}$. If $l < k + 1$, then $(x, y) \in >_k$. Otherwise, i.e., if $l = k + 1$, then, $k = |mss(x, y) \cap T| - 1$. Hence, $(x, y) \in \alpha(>_k)$. □

Equivalence. The abstraction for free-choice nets equals the $k + 1$ successor only if the places between the two nodes in consideration are initially empty. Therefore, we only consider free-choice WF-systems for equivalences. A sound WF-system is traced back to a live and bounded free-choice system and, initially, all places, except for the initial place, are empty.

Theorem 34. *Let S and S' be sound free-choice WF-systems. If $S \equiv_1^s S'$, then $S \equiv_k^s S'$, for any $k \in \mathbb{N}$ with $k > 0$.*

Proof. Analogously to the proof of Thm. 24 using the abstraction of Def. 32 and Prop. 33. □

Finally, we consider the expressiveness of relation sets. For sound free-choice WF-systems, 1-relation sets provide a complete characterisation of trace semantics. Therefore, down-to-1-equivalence coincides with trace equivalence.

Theorem 35. *Let $S = (N, m_0)$ and $S' = (N', m_0')$ be sound free-choice WF-systems. Then, S and S' are trace equivalent, iff $S \equiv_{\downarrow 1}^s S'$.*

Proof. Follows directly from Thm. 34, Prop. 28, and Prop. 16. □

5 Applications of Relational Semantics

Relation sets as proposed in this paper are a generalisation of existing notions of relational semantics. Earlier, we mentioned that those existing notions have been introduced for diverse applications within the field of business process modelling and analysis. We take up two applications and outline the benefits of using parametrised relation sets. Note that a formalisation of notions and measures for these applications is beyond the scope of this paper.

Process Model Similarity. Management of process model collections requires notions of process model similarity that are exploited for search and retrieval. Recently, different relation sets have been used as a means for similarity measures, see [10,31,14]. The overlap of relations, e.g., determined by the Jaccard coefficient, is used to quantify behavioural similarity. However, those works rely on a single instantiation of relation sets, i.e., choose a certain parameter. On the one hand, this raises the question of how to choose the parameter since different utility considerations may be followed. On the other hand, more fine-granular measures can be obtained once more than one parameter, i.e., more than one relation set, is taken into account. Assume that two transitions are related by 1-order in one system $S1$, but not in another system $S2$. This negatively impacts the similarity score, independent of the fact whether the two transitions are related by 2-order or only by 20-order in $S2$. Taking the whole spectrum of parametrised relation sets (or a reasonable subset thereof) as the basis, therefore, enables to distinguish those cases. The difference between 1-order and 2-order is arguably less severe than the one between 1-order and 20-order, which can be reflected in the similarity score.

Conformance Analysis. Conformance analysis is an integral part of process mining. Given process logs that capture the observed execution sequences of a business process, conformance checking answers the question to which extent the observed behaviour is in line with the behaviour defined by an according business process model, i.e., a net system. Conformance analysis may be based on relation sets. Then, a relation set is constructed not only for a net system, but also for a single execution sequence or a complete process log, cf., [26,2]. Existing work leverages only a single instantiation of relation sets, which, again, raises the question of appropriateness of different relation sets. This holds in particular, since conformance analysis has to cope with alien events (events in a log that are not represented by any transition in a net system) and silent transitions (transitions of a net system that are not represented by any event). Parametrised relation sets provide a means to address these challenges: the parameter of a relation set may be stepwise increased until the relation for a pair of net transitions coincides with the relation of the corresponding log events. A low parameter hints at a less severe deviation compared to a high parameter.

We conclude that techniques defined for a single instantiation of relation sets may benefit from taking the whole spectrum of parametrised relation sets into account. Then, instead of basing the analysis on a fixed distance between transition occurrences, parametrised relation sets allow for additional insights into the differences between two behavioural models. It is important to see that this holds even if only sound free-choice WF-systems are considered. Although Sec. 4 showed that 1-relation sets provide a complete characterisation of trace semantics for this class, parametrised relation sets provide a means to quantify the severity of any deviation. If models that are no sound free-choice WF-systems are considered, the usage of parametrised relation sets is advantageous even in terms of expressiveness. As illustrated with the examples in Fig. 3, different relation sets are incomparable for general net systems. A detailed analysis on the expressiveness of parametrised relation sets for more general classes of net systems is left for future work, though.

6 Related Work

Our work relates to other types of relational semantics and their applications.

Work on process mining does not only rely on the footprint, i.e., the 1-relation set, but also considers other relations. A causal matrix has been used in genetic process mining [3]. It formulates dependencies for transitions using input and output functions that associate subsets of preceding or succeeding transitions to a single transition. Hence, they capture a share of the 1-relation set. To assess the quality of mined models, follows and precedes relations that associate the value 'never', 'sometimes', or 'always' to pairs of transitions have been proposed [21]. However, those relations neglect the distance between the occurrences of transitions. Closely related are also approaches to the declarative modelling of behaviour. Those allow for the restriction of possible behaviour by constraints, e.g., expressed in Linear Temporal Logic, see [5,18].

Relation sets capture complete trace semantics only for restricted system classes, but are a behavioural abstraction for general systems. Other approximations of trace semantics are causal footprints [9]. A causal footprint defines two relations, look-back links and look-ahead links. For a transition, those define a set of transitions of which one must have occurred before or after the transition. In contrast to relation sets, however, there is no unique causal footprint for a single system. Communication fingerprints [19] are another behavioural abstraction for net systems that focusses on boundaries and dependencies for the cardinalities of tokens consumed or produced at dedicated places. As such, this work can be seen to be orthogonal to relation sets.

Besides those mentioned in Sec. 5, applications of relation sets include the verification of hardware specifications, modelled as a labelled net system [20]. Here, relations that classify transitions of a net system as being sequential or parallel are used. Also, the management of business process variants has been addressed using an order matrix [15,16]. The latter is a specific instantiation of relation sets introduced in this paper.

7 Conclusions

In this paper, we took up existing definitions of behavioural relations and provided a generalisation that defines a spectrum of relational semantics for Petri net systems. The relational semantics induce equivalence classes for systems that are not comparable for general net systems. However, for sound free-choice WF-systems, we proved that the relation sets form a hierarchy. This allows to derive the $(k+1)$-relation set from a k-relation set with knowledge about the 1-relation set. Also, we showed that for this class of systems, relation sets are derived from the structure of net systems. Finally, the 1-relation set provides a complete characterisation of trace semantics for sound free-choice WF-systems.

These insights are valuable for applying relation sets for several reasons. First, if the goal is checking equivalence of sound free-choice WF-systems, it is sufficient to consider 1-relation sets. Second, once focus is on measuring behavioural differences, we outlined that even for sound free-choice WF-systems it is advantageous to consider different k-relation sets. Third, we illustrated that k-relation sets are incomparable for general net systems. Hence, for general net systems, the joint application of different relation sets allows for a closer approximation of trace semantics.

In future work, we aim at investigating the expressiveness of k-relation sets beyond the class of sound free-choice WF-systems. The example given in Fig. 3 illustrates that k-relation sets offer a means to distinguish behavioural differences that stem from non-free-choiceness. However, even a joint usage of different k-relation sets may not provide a complete characterisation of trace semantics for net systems. Thus, we aim at exploring for which more general classes of net systems the set of k-relation sets provides a complete trace characterisation. In addition, we aim at investigating k-relation sets for labelled net systems. Although we foresee that the relations may be directly lifted to labelled net systems, the influence on the expressiveness of k-relation sets needs to be clarified.

References

1. van der Aalst, W.M.P.: The application of Petri nets to workflow management. Journal of Circuits, Systems, and Computers 8(1), 21–66 (1998)
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 Medeiros, A.K.A., Weijters, A.J.M.M.T.: Genetic Process Mining. In: Ciardo, G., Darondeau, P. (eds.) ICATPN 2005. LNCS, vol. 3536, pp. 48–69. Springer, Heidelberg (2005)
4. van der Aalst, W.M.P., van Hee, K.M., 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)
5. van der Aalst, W.M.P., Pesic, M., Schonenberg, M.H.: Declarative workflows: Balancing between flexibility and support. Computer Science - R&D 23(2), 99–113 (2009)
6. van der Aalst, W.M.P., Stahl, C.: Modeling Business Processes: A Petri Net-Oriented Approach. MIT Press (2011)
7. van der Aalst, W.M.P., Weijters, A.J.M.M., Maruster, L.: Workflow Mining: Discovering Process Models from Event Logs. Knowledge & Data Engineering 16(9), 1128–1142 (2004)
8. Desel, J., Esparza, J.: Free Choice Petri Nets. Cambridge Tracts in Theoretical Computer Science, vol. 40. Cambridge University Press (1995)
9. van Dongen, B.F., Dijkman, R.M., Mendling, J.: Measuring Similarity between Business Process Models. In: Bellahsène, Z., Léonard, M. (eds.) CAiSE 2008. LNCS, vol. 5074, pp. 450–464. Springer, Heidelberg (2008)
10. Eshuis, R., Grefen, P.W.P.J.: Structural Matching of BPEL Processes. In: ECOWS, pp. 171–180. IEEE Computer Society (2007)
11. Esparza, J.: A polynomial-time algorithm for checking consistency of free-choice signal transition graphs. Fundamenta Informatica 62(2), 197–220 (2004)
12. Huan, L., Motoda, H. (eds.): Feature Extraction, Construction and Selection: A Data Mining Perspective. Springer, Berlin (1998)

13. Kovalyov, A., Esparza, J.: A polynomial algorithm to compute the concurrency relation of free-choice signal transition graphs. In: Proc. of the International Workshop on Discrete Event Systems (WODES), pp. 1–6. IEEE CS (1996)
14. Kunze, M., Weidlich, M., Weske, M.: Behavioral Similarity – A Proper Metric. In: Rinderle-Ma, S., Toumani, F., Wolf, K. (eds.) BPM 2011. LNCS, vol. 6896, pp. 166–181. Springer, Heidelberg (2011)
15. Li, C., Reichert, M., Wombacher, A.: Discovering Reference Models by Mining Process Variants Using a Heuristic Approach. In: Dayal, U., Eder, J., Koehler, J., Reijers, H.A. (eds.) BPM 2009. LNCS, vol. 5701, pp. 344–362. Springer, Heidelberg (2009)
16. Li, C., Reichert, M., Wombacher, A.: Representing block-structured process models as order matrices: Basic concepts, formal properties, algorithms. Technical report, University of Twente, Enschede, The Netherlands (2009)
17. Lohmann, N., Verbeek, E., Dijkman, R.M.: Petri net transformations for business processes - a survey. T. Petri Nets and Other Models of Concurrency 2, 46–63 (2009)
18. Montali, M., Pesic, M., van der Aalst, W.M.P., Chesani, F., Mello, P., Storari, F.: Declarative specification and verification of service choreographiess. TWEB 4(1) (2010)
19. Oanea, O., Sürmeli, J., Wolf, K.: Service Discovery Using Communication Fingerprints. In: Maglio, P.P., Weske, M., Yang, J., Fantinato, M. (eds.) ICSOC 2010. LNCS, vol. 6470, pp. 612–618. Springer, Heidelberg (2010)
20. Rosenblum, L.Y., Yakovlev, A.V.: Analysing semantics of concurrent hardware specifications. In: Int. Conf. on Parallel Processing (ICPP 1989), pp. 211–218 (1989)
21. Rozinat, A., van der Aalst, W.M.P.: Conformance checking of processes based on monitoring real behavior. Inf. Syst. 33(1), 64–95 (2008)
22. Smirnov, S., Weidlich, M., Mendling, J.: Business process model abstraction based on synthesis from well-structured behavioral profiles. International Journal of Cooperative Information Systems, IJCIS (to appear, 2012)
23. Smirnov, S., Weidlich, M., Mendling, J., Weske, M.: Action patterns in business process model repositories. Computers in Industry (COMIND) 63(2), 98–111 (2012)
24. Warshall, S.: A theorem on boolean matrices. Journal of the ACM 9(1), 11–12 (1962)
25. Weidlich, M., Mendling, J., Weske, M.: Efficient consistency measurement based on behavioral profiles of process models. IEEE Trans. Software Eng. 37(3), 410–429 (2011)
26. Weidlich, M., Polyvyanyy, A., Desai, N., Mendling, J., Weske, M.: Process compliance analysis based on behavioural profiles. Inf. Syst. 36(7), 1009–1025 (2011)
27. Weidlich, M., Weske, M., Mendling, J.: Change propagation in process models using behavioural profiles. In: IEEE SCC, pp. 33–40. IEEE CS (2009)
28. van der Werf, J.M.E.M., van Dongen, B.F., Hurkens, C.A.J., Serebrenik, A.: Process Discovery Using Integer Linear Programming. Fundamenta Informatica 94(3-4), 387–412 (2009)
29. Weske, M.: Business Process Management: Concepts, Languages, Architectures. Springer, Berlin (2007)
30. Wolf, K.: Does My Service Have Partners? In: Jensen, K., van der Aalst, W.M.P. (eds.) Transactions on Petri Nets and Other Models of Concurrency II. LNCS, vol. 5460, pp. 152–171. Springer, Heidelberg (2009)
31. Zha, H., Wang, J., Wen, L., Wang, C., Sun, J.: A workflow net similarity measure based on transition adjacency relations. Computers in Industry 61(5), 463–471 (2010)

Data and Abstraction for Scenario-Based Modeling with Petri Nets

Dirk Fahland and Robert Prüfer

Eindhoven University of Technology, The Netherlands
Humboldt-Universität zu Berlin, Germany
d.fahland@tue.nl, pruefer@informatik.hu-berlin.de

Abstract. Scenario-based modeling is an approach for describing behaviors of a distributed system in terms of partial runs, called *scenarios*. Deriving an operational system from a set of scenarios is the main challenge that is typically addressed by either *synthesizing* system components or by providing *operational semantics*. Over the last years, several established scenario-based techniques have been adopted to Petri nets. Their adaptation allows for verifying scenario-based models and for synthesizing individual components from scenarios within one formal technique, by building on Petri net theory. However, current adaptations of scenarios face two limitations: a system modeler (1) cannot abstract from concrete behavior, and (2) cannot explicitly describe data in scenarios. This paper lifts these limitations for scenarios in the style of Live Sequence Charts (LSCs). We extend an existing model for scenarios, that features Petri net-based semantics, verification and synthesis techniques, and close the gap between LSCs and Petri nets further.

Keywords: scenario-based modeling, data, abstraction, Petri nets.

1 Introduction

Designing and implementing a distributed system of multiple components is a complex task. Its complexity originates in the component interactions. Established *scenario-based methods* such as *(Hierarchical) Message Sequence Charts* ((H)MSCs) [26] and *Live Sequence Charts* (LSCs) [6] alleviate this complexity: A system designer *specifies* the system's behaviors as a *set of scenarios*. Each scenario is a self-contained, partial execution usually given in a graphical notation. Then system components are *synthesized* (preferably automatically) that together interact as described in the scenarios. Alternatively, a specification *becomes* a system model by equipping scenarios with *operational semantics*.

A significant drawback of established techniques is that components have to be synthesized in a different formal theory than the one in which the scenarios are given, e.g., HMSCs or LSCs are synthesized into Petri nets or statecharts [7,19]. Also operational semantics for MSCs and LSCs require a translation into another formalism like automata [31], process algebras [30], or require involved formal techniques such as graph grammars [24] or model-checking [20]. Many HMSC and LSC specifications cannot be distributed into components but require centralized control [7,4]. This renders turning scenarios into systems surprisingly technical while scenarios appear to be very intuitive.

Approaches which express scenarios, system behaviors and system model in the same formal theory face less problems. In particular, approaches which describe scenarios in terms of Petri nets and their partially ordered runs, e.g., [9], have been successful. The approach in [2] presents a general solution for synthesizing a Petri net from HMSC-style specifications in a Petri net-based model. [8] shows how to compose complex system behaviors from single Petri net events with preconditions. The model of *oclets* [11,12] adapts ideas from LSCs to Petri nets: a scenario is a partial run with a distinguished precondition; system behavior emerges from composing scenarios based on their preconditions. This idea allows oclets to adapt existing Petri net techniques for a general solution to the synthesis problem for LSC-style scenarios [12].

The Petri net-based scenario techniques [9,2,8,11,12] in their current form only describe control-flow and provide no means for abstracting behavior in a complex specification. Any practically applicable specification technique needs some notion of abstraction as well as some explicit notion of data, and means to describe several components of the same kind.

This paper addresses these problems of practical applicability of Petri-net based scenarios. We show for the model of oclets how to extend scenarios by abstract causal dependencies (abstracting from a number of possibly unknown actions between two dependent actions), and how to express data in scenarios by adapting notions of Algebraic Petri nets [32]. Our contribution is two-fold: First, abstraction and data are two key features of LSCs, so our extension of oclets closes the gap between LSCs and Petri nets further. Second, all our extensions are simple generalizations of existing concepts from Petri net theory, giving rise to the hope that existing verification and synthesis results, e.g., [12], can be transferred to the more expressive model proposed in this paper.

We proceed as follows. Section 2 recalls the scenario-based approach in more detail and explains the basic ideas of oclets by an example. In Sections 3 and 4, abstract causal dependencies and data are introduced into the model, respectively. Section 5 discusses the relation of oclets to Petri nets. We conclude and discuss related and future work in Section 6.

2 Specifying with Scenarios

This section recalls the scenario-based approach by the help of an example and discusses features and limitations of scenario-based specification techniques. Two of these limitations will be addressed in the remainder of this paper.

2.1 Running Example and Requirements for Capturing It

Our running example is a gas station (adapted from [23]) that allows customers to refuel their cars using one of the available pumps as follows. When a customer arrives with his car at a pump, he asks the operator to activate that pump for a certain amount of fuel for which he pays in advance or after he finished pumping the gas. The customer can start an activated pump to refuel his car, pumping gas one unit at a time. The pump stops when all requested gas has been pumped, or when stopped by the customer. The pump then signals the operator the pumped amount and the operator returns corresponding change to the customer. Each customer gets a free snack that he may pick up after starting the pump and before leaving the gas station.

A specification technique capable to express this gas station has to describe (R1) *distributed components* (e.g., pump, customer, operator), (R2) *interaction* between components, (R3) *sequential*, *independent*, and *alternative* ordering of actions, (R4) *preconditions* of actions (e.g., "when all requested gas has been pumped"), (R5) actions that depend on *data* (e.g., returned change, amount of pumped gas), (R6) *multiple instances* of the same kind of component (e.g., multiple customers and pumps). Furthermore, a specification technique also should allow a system designer to keep an overview of larger specifications by (R7) *means of abstraction*. Finally, the specification technique should allow to (R8) *derive the specified behavior in an intuitive* way, that is, the derived behavior should be "correct by construction" and not require additional verification.

2.2 Principles of Scenario-Based Specifications

In the scenario-based approach, a system designer obtains a system model of a distributed system (e.g., our gas station example) in two steps. First she describes the system behavior as *component interactions*. One *scenario* describes how several components interact with each other in a particular situation; a *specification* is a set of scenarios. When she completed the specifications, components are *synthesized* (preferably automatically) such that all components together interact as described in the scenarios.

The most valued feature of this approach is that scenarios *tangibly* decompose complex system behavior into smaller, self-contained stories of component interactions (scenarios) which are easy to understand. Fig. 1 shows 3 scenarios (M0, M1, M2) of the running example in the well-established syntax of MSCs. In each MSC, a vertical *lifeline* describes one *component*, arrows between components describe *interactions*, boxes at components describe *local actions*; M0 is a special case as it describes the creation of a new instance of a customer. Established scenario-based techniques adhere to a few simple principles that allow to *derive system behavior from scenarios* in a comprehensible way as follows.

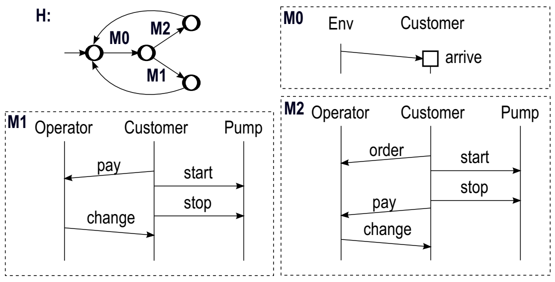


Fig. 1. An HMSC H describing compositions of MSCs M0, M1, M2

S1 A scenario is *partial order* of actions, understood as a *partial run* of the system. A *specification* is a set of scenarios.

S2 System behavior follows from *composing* (appending) scenarios.

S3 Each scenario distinguishes a prefix as a *precondition* describing *when* the scenario can occur; the remainder of the scenario is called *contribution*.

S4 When a system run ends with a scenario's precondition, the run can continue by *appending* the scenario's contribution. Scenarios with the same precondition and different contributions lead to *alternative* runs.

S1 is the most generally agreed upon principle for scenarios and usually expressed in an MSC-like notation as in Fig. 1; other notations are possible [6,9,11,2]. To specify practically relevant systems, more principles are needed. The probably most established scenario-based techniques – (H)MSCs, LSCs and UML Sequence Diagrams – realize these principles differently as we discuss next.

HMSCs proposed principle S2 first, where the order of scenario composition is described by a finite automaton [26]. For instance, HMSC H of Fig. 1 describes that M0 is followed by M1 *or alternatively* by M2, and then M0 can occur again. This way, HMSCs are capable to express the requirements R1-R3 of Sect. 2.1, but not R4. In the HMSC standard, notions of data (R5) are only provided on a syntactical, but not on a semantical level [26]. Also multiple instances of the same component cannot be expressed: the HMSC of Fig. 1 allows only one customer to be served at a time. Means of abstraction (R7) are provided by the possibility of nesting one MSC inside another MSC. UML Sequence Diagrams express scenario composition entirely by nesting scenarios in each other.

It has been repeatedly observed that this approach to scenarios requires a *global* understanding of the entire system as the ordering of scenarios is described in a global automaton [17,12]. Moreover, as MSCs of a HMSC cannot overlap, a specification may have to be refactored when a new scenario shall be included [34] and specifications tend to consist of many small-scale scenarios composed in complex ways, which is counter-intuitive to the idea of one scenario describing a "self-contained story" of the system [38,12].

LSCs extend MSCs in a different way to provide enough expressive power [6]. Altogether, LSCs provide notions for preconditions of scenarios, data, multiple instances and abstraction on component lifelines, satisfying R1-R7 of Sect. 2.1 [21]. LSCs first proposed principle S3 that a scenario is triggered by a precondition; this idea has been adopted in other approach as well [17,35,11]. Fig. 2 shows 3 LSCs corresponding to the MSCs of Fig. 1. However, LSCs specify system behavior not by scenario composition, but each LSC denotes *a linear-time temporal logic formula*: when a system run ends with an LSC's precondition, then the run *must* continue with the LSC's contribution, i.e., the given events occur in the run eventually in the given order.

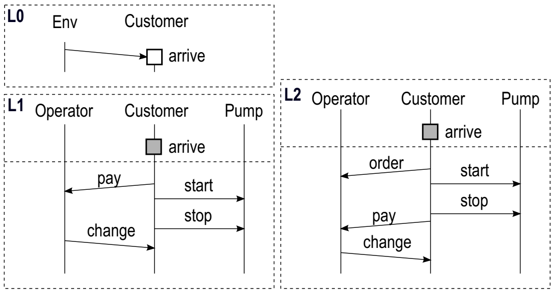


Fig. 2. Three LSCs of the gas station example

For instance, L1 expresses that *after* arrive occurred (precondition), the customer pays, starts and stops the pump, and gets his change (contribution). Additionally, LSCs specify particular events as *implicitly forbidden* at particular stages of an LSC. For instance, in L2, event pay is *forbidden to occur before* event stop. At the same time, L1 requires pay to occur right after arrived (before stop) and *forbids occurrences of* pay *after* stop. In contrast to intuition, L1 and L2 are *contradictory* and no system satisfies both LSCs. This contradiction arises because of the linear-time semantics of LSCs as both L1 and L2 have to occur in the same run. The contradiction vanishes when using a *branching-time semantics* for LSCs as proposed in the model of *epLSCs* [37]: when the pre-condition occurs, *some run* continues with the contribution (principle S4). However, also in this model, deriving system behavior from a specification is cumbersome: whether two epLSCs have to occur in different runs or may occur overlappingly in the same run still requires to *check* their temporal logic formulae, possibly requiring verification on large parts of the specified state-space [20].

Between LSCs and HMSCs. To summarize, HMSCs have a simple semantics that allows to derive specified behaviors by composing scenarios. Though, HMSCs suffer from the global automaton and that scenarios cannot overlap. LSCs allow for local preconditions and overlapping scenarios, but are based on an intricate semantics that makes it hard to understand behavior specified by a set of LSCs. That simplicity of semantics influences the way how components can be synthesized from scenarios can be seen when comparing available techniques. While synthesis is generally infeasible from both HMSCs and LSCs, synthesis from feasible subclasses of scenarios to Petri nets is straight forward for HMSCs [31,2] yielding components that are *correct by construction*, whereas synthesis from LSCs [1,28] requires to *verify the synthesis result* to ensure correctness.

In the following, we derive a scenario-based technique that inherits the advantages of HMSCs and LSCs without their disadvantages: a LSC-style syntax of scenarios with local precondition is given a HMSC-style semantics where system behavior follows from scenario composition. The hope is that a simpler semantic model allows to synthesize from LSC-style scenarios components that are correct by construction. Indeed, this already has been proven to be successful for a simple model of scenarios that we present next.

2.3 A Simple Model for Scenarios Based on Petri Nets

We derive a simple semantic model for LSC-style scenarios by applying principles of Petri net theory. Petri net-based scenarios benefit from expressing scenarios, behavior, and system in the same formal model, which allows to create a Petri-net based operational semantics for scenarios and supports the crucial step from specification to system model. For HMSC-style specifications, corresponding semantics and synthesis techniques are already available [2]

For LSC-style scenarios, a simple model that adopts the semantics of epLSCs [37] to Petri nets has been proposed in the model of *oclets* [11,12] that we recall next. Oclets realize all principles S1-S4 in the following way. Fig. 3 shows four scenarios of the gas station example of Sect. 2.1 in the notation of oclets. The partial order of actions is expressed as a so called *labeled causal net*. A transition (place) of a causal net is called event (condition); the flow relation defines a partial order over events and conditions s.t. the net is conflict-free. The grey-filled (white-filled) nodes indicate the precondition (contribution) of an oclet.

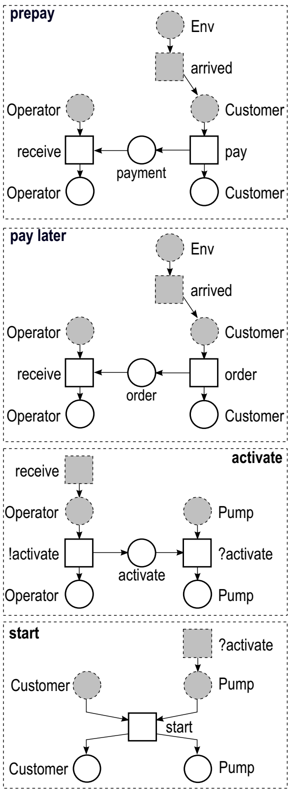


Fig. 3. Scenarios for the gas station example

The four oclets describe some behavior of the gas station example. Oclet prepay: after a customer arrived at the gas station, he asks to activate a pump and pre-pays his gas (event pay). Alternatively (oclet pay later), the customer may just ask the operator to activate the pump (order). Oclet activate: after receiving the order, the operator activates the pump. Oclet start: When a pump is activated, the customer may start it.

Oclets generalize the semantics of Petri net transitions to scenarios. Whenever a run ends with an oclet's precondition, the oclet is *enabled* and the run can continue by appending the oclet's contribution. Two oclets with the same precondition and different contributions are alternatives and hence yield alternative continuations.

For example, consider the run π_0 indicated in Fig. 4. In π_0, oclets pay and use card are enabled. Continuing π_0 with pay yields the run π_1 indicated in Fig. 4(left), that was obtained by appending pay's contribution to π_0. Appending pay later yields the run π'_1 of Fig. 4(right) that is alternative to π_1. In π_1, oclet activate is enabled; appending its contribution yields π_2. This way, oclets *derive* the specified behavior of the gas station by composition.

Properties and limitations. In their current form, oclets allow to express properties (R1-R4) of Sect. 2.1 and allow to analyze scenarios, and synthesize a system by reusing and extending Petri net techniques [12]. Yet, oclets cannot express data (e.g., how much gas to pump) or distinguish instances (e.g., two different pumps). Furthermore, the events in an oclet currently describe a "contiguous piece of behavior." In the worst case, the specification consists of many short scenarios, only. *Abstraction* would allow a system designer to also specify longer scenarios of corresponding, non-contiguous pieces of behavior. In the remainder of this paper, we show how to introduce abstraction and data to oclets, thus providing a scenario-based technique between HMSCs and LSCs. We stay close to the spirit of Petri nets and define an extension in terms of simpler, existing principles.

3 Adding Abstraction: Abstract Dependencies

In this section, we introduce means to abstract from behavior in a scenario. We sketch the idea by our running example before we present formal definitions.

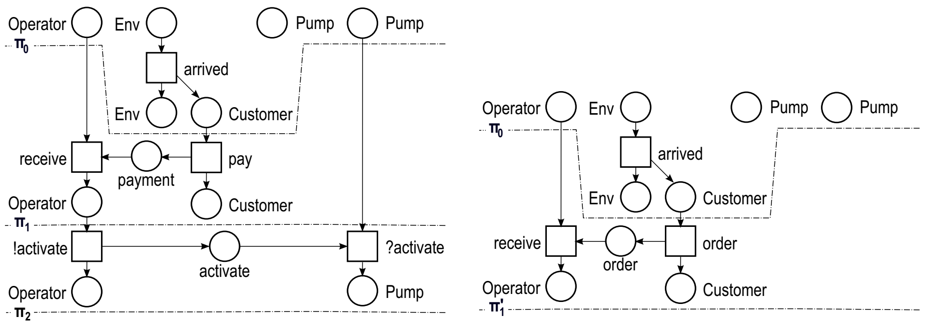


Fig. 4. Two alternative runs of the gas station built by composing scenarios of Fig. 3

3.1 Abstracting Causal Dependencies

As stated in Sect. 2.3, the flow relation of an oclet as described in [11,12] denotes direct causal dependencies which may restrict how a particular system behavior can be specified. *Abstract causal dependencies* allow other events to occur between two events of a scenario.

Fig. 5 shows examples of abstract dependencies; an abstract dependency is drawn as a dashed arrow. The oclet main describes the main interaction between customer and the gas station's operator and the pump (see Sect. 2.1). The oclet abstracts from other behavior taking place at the gas station, some of that behavior is needed to make the customer's interaction happen. For instance, oclet main only abstractly describes the dependency of the two Pump conditions. This allows events which are not depicted to occur between these conditions. In particular event start of oclet start of Fig. 3 can occur here. In other words, oclet start *refines* this abstract dependency of oclet main. There are further abstract dependencies in main that have to be detailed by other oclets. Yet, the main scenario clearly describes that once the pump has been activated, it eventually completes pumping, which will lead to the operator returning change to the customer. We complete the specification in Sect. 4.4.

Abstract dependencies are also useful in a scenario's precondition. Here, they allow to specify that an oclet is enabled if a specific behavior occurred "some time" in the past, instead of immediately. For instance, oclet main[completed] in Fig. 5 expresses that activate must have occurred some time in the past in order to enable event completed, including the possibility that other events occurred in between.

Introducing abstract dependencies in oclets comes at a price: we cannot continue a run with an enabled oclet by appending its contribution. The principle solution is to decompose an oclet into *basic* oclets such as main[completed] in Fig. 5(top right). Each basic oclet contributes exactly one event, its precondition consists of all transitive predecessors in the original oclet. This effectively moves abstract dependencies into preconditions and system behavior emerges from concatenating well-defined single events.

In the remainder of this section, we formalize these ideas by extending syntax and semantics of oclets [11,12] with abstract dependencies. We assume the reader to be familiar with the basic concepts of Petri Nets and their distributed runs; see [33] for an introduction.

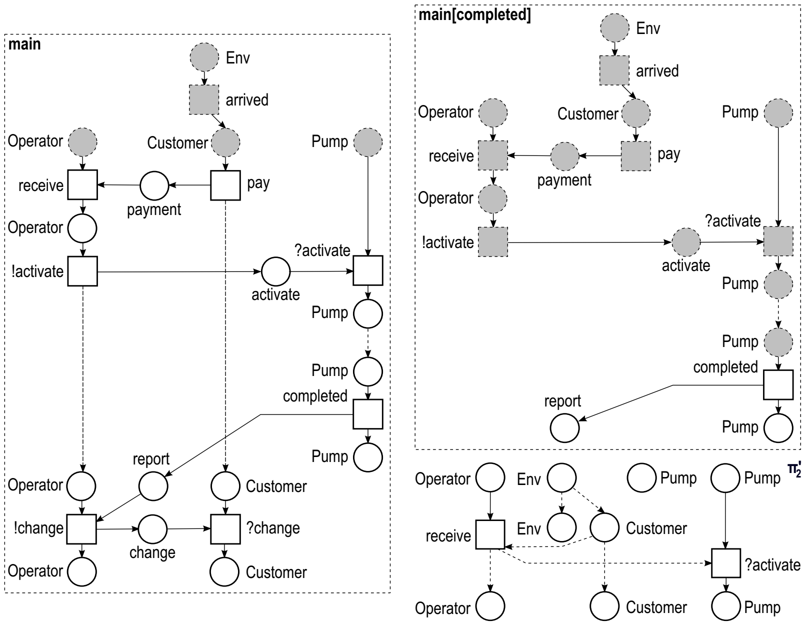


Fig. 5. Main scenario of the gas station example (left), abstract dependencies allow to abstract several details of the system behavior; a basic oclet (top right); an abstract run (bottom right)

3.2 Basic Notions

First, we recall some basic notation. A *partial order* over a set A is a binary relation $\leq \subseteq A \times A$ that is reflexive (i.e. $\forall a \in A : a \leq a$), transitive (i.e. $\forall a, a', a'' \in A : a \leq a' \wedge a' \leq a'' \Rightarrow a \leq a''$), and antisymmetric (i.e. $\forall a, a' \in A : a \leq a' \wedge a' \leq a \Rightarrow a = a'$). Let $a{\downarrow_{\leq}} := \{a' \in A \mid a' \leq a\}$ and $a{\uparrow_{\leq}} := \{a' \in A \mid a \leq a'\}$ denote the *transitive predecessors* and *successors* of $a \in A$, respectively. As usual, for a relation $R \subseteq (A \times A)$, R^+ and R^* denote the transitive, and reflexive-transitive closures of R.

We write a *Petri Net* as $N = (P, T, F)$, with places P, transitions T ($P \cap T = \emptyset$), and arcs $F \subseteq (P \times T) \cup (T \times P)$. Notation-wise, introducing N, N_1, N' implicitly introduces their components P_N, P_1, P' etc. For each *node* $x \in X_N := P \cup T$, ${}^\bullet x = \{y \in X \mid (y, x) \in F\}$ and $x^\bullet = \{y \in X \mid (x, y) \in F\}$ are the *pre-* and *post-set* of x, respectively. A *causal net* $\pi = (B, E, F)$ is a Petri net where (1) $\leq_\pi := F^*$ is a partial order over X_π, (2) for each $x \in X_\pi$, $x{\downarrow_{\leq_\pi}}$ is finite, and (3) for each $b \in B$, $|{}^\bullet b| \leq 1$ and $|b^\bullet| \leq 1$. An element of B (E) is called *condition* (*event*). The arcs of a causal net denote *direct causal dependencies*: x depends on y iff $x \leq y$, and x and y are *concurrent* iff neither $x \leq y$ nor $y \leq x$. We write $\min \pi = \{x \mid {}^\bullet x = \emptyset\}$ and $\max \pi = \{x \mid x^\bullet = \emptyset\}$ for the nodes without predecessor and successor, respectively.

In the following, we consider *labeled* causal nets $\pi = (B, E, F, \ell)$ where each node $x \in X_\pi$ is assigned a label $\ell(x) \in L$ from some given set L. We will interpret a labeled causal net as a *partially ordered run* (of a possibly unknown system); in analogy to Petri nets, an event e describes an occurrence of an action (or transition) $\ell(e)$, and a condition b describes an occurrence of a local state (a token on a place) $\ell(b)$.

3.3 Runs with Abstract Dependencies

We formally introduce *abstract dependencies* by generalizing the notion of a partially ordered run to an *abstract* run. This definition then canonically lifts to oclets with abstract dependencies.

Definition 1 (Partially ordered run). *An* abstract partially ordered run *(*run *for short)* $\pi = (B, E, F, A, \ell)$ *is a labeled causal net* (B, E, F, ℓ) *with* abstract dependencies $A \subseteq X_\pi \times X_\pi$ *s.t.* $\leq_\pi := (F \cup A)^*$ *is a partial order over the nodes* X_π. π *is* concrete *iff* $A = \emptyset$.

We write π^α for any run that is isomorphic to π by the isomorphism $\alpha : \pi \to \pi^\alpha$. Run π *occurs* in run ρ, written $\pi \subseteq \rho$, iff $B_\pi \subseteq B_\rho, E_\pi \subseteq E_\rho, F_\pi \subseteq F_\rho, A_\pi \subseteq A_\rho, \ell_\pi = \ell_\rho|_{X_\pi}$. We will work with two important relations on runs: prefixes and refinement.

Definition 2 (Prefix). *A run* π *is a* prefix *of a run* ρ*, written* $\pi \sqsubseteq \rho$*, iff (1)* $\min \rho \subseteq X_\pi \subseteq X_\rho$ *(2)* $F_\pi = F_\rho|_{X_\rho \times X_\pi}$, $A_\pi = A_\rho|_{X_\rho \times X_\pi}$ *(*π *contains* all *predecessors), (3) for each* $e \in E_\rho$ *and all* $(e, b) \in F_\rho$ *holds* $(e, b) \in F_\pi$ *(events have all post-conditions). The set of all prefixes of* ρ *is* $Pre(\rho) := \{\pi \mid \pi \sqsubseteq \rho\}$.

Fig. 5(bottom right) shows an abstract run; Fig. 4 shows concrete runs; π_1 is a prefix of π_2; π'_1 is not a prefix of π_1. Each abstract run describes a set of concrete runs (without abstract dependencies) that *refine* the abstract run. Intuitively, an abstract dependency in a run π can be refined by a number of nodes that respect the partial order of π. The refinement can *exclude* nodes with a particular label, which we need for oclet semantics.

Definition 3 (Refine an abstract run). *Let* π *and* ρ *be abstract distributed runs. Let* $\kappa : A_\pi \to 2^L$ *assign each abstract dependency in* π *a (possibly empty) set of forbidden labels.* ρ refines π *w.r.t.* κ*, written* $\rho \leq_\kappa \pi$ *iff*

- $B_\pi \subseteq B_\rho, E_\pi \subseteq E_\rho, \forall x \in X_\pi : \ell_\pi(x) = \ell_\rho(x)$,
- $F_\pi \subseteq F_\rho$ *and* $(F_\pi \cup A_\pi)^+ \subseteq (F_\rho \cup A_\rho)^+$, *and*
- $\forall (x, y) \in A_\pi \nexists e \in E_\rho : x \leq_\rho e \leq_\rho y \wedge \ell_\rho(e) \in \kappa(x, y)$.

We write $\pi \leq \rho$ *if* $\kappa(x, y) = \emptyset$ *for all* $(x, y) \in A_\pi$. *Every run* π *describes the set* $[\![\pi]\!] := \{\rho \mid \rho \leq \pi, A_\rho = \emptyset\}$ *of all concrete runs that refine* ρ.

Run π_2 of Fig. 4 refines the abstract run π'_2 of Fig. 5.

3.4 Scenarios with Abstract Dependencies

Syntax. Abstract dependencies canonically lift from abstract runs to oclets. As already sketched in Sect. 3.1, an oclet is an abstract run with a distinguished prefix.

Definition 4 (Oclet). *An oclet* $o = (\pi, pre)$ *consists of an abstract distributed run* π *and a prefix* $pre \sqsubseteq \pi$ *of* π.

Fig. 5 shows oclet main. Def. 4 generalizes "classical" oclets as introduced in [11,12] by abstract dependencies of the underlying run. We call *pre* the *precondition* of *o* and the suffix $con(o) := (B_\pi \setminus B_{pre}, E_\pi \setminus E_{pre}, F_\pi \setminus F_{pre}, A_\pi \setminus A_{pre}, \ell_\pi|_{X_{con(o)}})$ its *contribution*; technically $con(o)$ is not a net as it contains arcs adjacent to nodes of the precondition.

A specification is a set of oclets together with an *initial run* that describes how system behavior starts.

Definition 5 (Specification). *A* specification $\Omega = (O, \pi_0)$ *is a set O of oclets together with an abstract run* π_0 *called* initial *run.*

A specification usually consists of a *finite* set of oclets; we allow the infinite case for technical reasons. This is all syntax that we need.

Semantics. The semantics of oclets is straight forward. An oclet o describes a scenario with a necessary precondition: the contribution of o can occur *whenever* its precondition occurred. Then, we call *o enabled*.

Definition 6 (Enabled Oclet). *Let* $o = (\pi, pre)$ *be an oclet and let* π *be a run. Each* $(x, y) \in A_\pi$ *defines the post-events of y as* forbidden, *i.e.,* $\kappa(x, y) = \{\ell(e) \mid (y, e) \in F_o \cup A_o, e \in E_o\}$*. Oclet o is* enabled in π *iff there exists a refinement* $pre' \preceq_\kappa pre$ *s.t. (1)* $pre' \subseteq \pi$*, (2)* $\max pre' \subseteq \max \pi$*, and (3)* $X_{con(o)} \cap X_\pi = \emptyset$*.*

An oclet is enabled in a run π if the complete precondition occurs at end of π, i.e., “just happened.” The forbidden events κ restrict which events may occur in place of an abstract dependency of *pre* (see Def. 3); this ensures enabling only at a “very recent” occurrence of the precondition. The same model is applied in LSCs [6].

An oclet o can be enabled at several different locations in π (whenever we find *pre* several times at the end of π). We say that o is enabled in π *at location* α iff o^α is enabled in π. For technical reasons, o's contribution is assumed to be disjoint from π so that it can be appended to π.

Definition 7 (Continue a run with an oclet). *Let o be an oclet and let* π *be a distributed run. If o is enabled in* π*, then the* composition *of* π *and o is defined as* $\pi \rhd o := (\pi \cup \pi_o) = (B_\pi \cup B_o, E_\pi \cup E_o, F_\pi \cup F_o, \ell', A_\pi \cup A_o)$ *with* $\ell'(x) = \ell_\pi(x)$*, for all* $x \in X_\pi$*,* $\ell'(x) = \ell_o(x)$*, for all* $x \in X_o$*.*

A specification Ω describes a set $R(\Omega)$ of abstract runs: that is, the *prefixes* of all runs that can be constructed by repeatedly appending enabled oclets of Ω to the initial run. The concrete system behaviors specified by Ω are the concrete runs that refine $R(\Omega)$.

Definition 8 (Semantics of a specification). *Let* $\Omega = (O, \pi_0)$ *be a specification. The abstract runs of* Ω *are the least set* $R(\Omega)$ *of runs s.t.*

1. $Pre(\pi_0) \subseteq R(\Omega)$*, and*
2. *for all* $\pi \in R(\Omega), o \in O$*, if o is enabled in* π *at* α *then* $Pre(\pi \rhd o^\alpha) \subseteq R(\Omega)$*.*

A set R of concrete runs satisfies Ω *iff for each* $\pi \in R(\Omega)$*,* $[\![\pi]\!] \cap R \neq \emptyset$*.*

3.5 Operational Semantics

A system designer can use oclets to specify the behaviors of a distributed system. Harel et al. [20] suggested to turn a specification into an executable system model by providing *operational semantics* for scenarios.

Operational semantics describe system behavior as occurrences of single events. Each event has a *local* precondition, if the precondition holds, the event can occur by being appended to the run. These principles are naturally captured by *basic* oclets that contribute just a single event; we define operational semantics of oclets by *decomposing* complex oclets into basic oclets.

We call an event e of a run π *concrete* iff e has no abstract dependencies, i.e., $\forall (x, y) \in A_\pi : x \neq e \neq y$. Oclet o is *basic* iff its contribution consists of exactly one concrete event e (with post-conditions). If o is not basic, then it can be decomposed into basic oclets. Each concrete event e of o's contribution induces the basic oclet $o[e]$ that contributes e and e's post-set and has as precondition all transitive predecessors $e \downarrow_{\leq_o}$ of e in o.

Definition 9 (Decomposition into basic oclets). *Let $o = (\pi, pre)$ be a basic oclet. A concrete event $e \in E_{con(o)}$ induces the basic oclet $o[e] = (\pi', pre')$ with $X' = e \downarrow_{\leq_o} \cup e^\bullet$, $F' = F|_{X' \times X'}$, $A' = A|_{X' \times X'}$, $\ell' = \ell|_{X'}$, and $pre' \sqsubseteq \pi'$ s.t. $X_{pre'} = e \downarrow_{\leq_o} \setminus \{e\}$. The basic oclets of o are $\hat{o} = \{o[e] \mid e \in E_{con(o)}, e \text{ is concrete}\}$.*

There may be specifications where a particular action a has no corresponding concrete event $e, \ell(e) = a$, i.e., it is always adjacent to some abstract dependency. In this case, the specification provides no information on how to refine these abstract dependencies. We found it useful for concise specifications, that in this case, a non-concrete event e of an oclet $o = (\pi, pre)$ also induces the basic oclet $o[e]$ where the abstract dependencies between e and some condition b are turned into direct dependencies, i.e., replace in o each abstract dependency $(b, e) \in A_o, b \in B_o$ by an arc $(b, e) \in F_o$ and each $(e, b) \in A_o, b \in B_o$ by an arc $(e, b) \in F_o$, and then compute $o[e]$ as in Def. 9.

In both cases of Def. 9 and with additional basic oclets, the operational semantics of an oclet specification follows from its basic oclets.

Definition 10 (Operational semantics). *Let $\Omega = (O, \pi_0)$ be a specification. $\hat{\Omega} = (\bigcup_{o \in O} \hat{o}, \pi_0)$ is the* basic *specification induced by Ω. It defines the* operational semantics *of Ω as $R(\hat{\Omega})$. Ω is* operational *iff $R(\hat{\Omega})$ satisfies Ω.*

We call $R(\hat{\Omega})$ the operational semantics of Ω because $\hat{\Omega}$ describes a set of single events. Each event is enabled when its local precondition holds; an enabled event can occur (by appending it to the run). Not every specification has operational semantics that satisfy the specification. A non-operational specification needs to be refined to become operational. Yet, we can characterize operational specifications.

Theorem 1. *Let $\Omega = (O, \pi_0)$ be a specification s.t. π_0 is concrete and each event in the contribution of each oclet in O is concrete. Then Ω is operational.*

Proof. This theorem has been proven for oclets without abstract dependencies in [11]: by induction on the prefixes of an oclet's contribution, an oclet's contribution can be reconstructed from its basic oclets. In particular, whenever oclet o is enabled in π, also each basic oclet of o is enabled in π or in a continuation $\pi \triangleright o[e_1] \triangleright \ldots \triangleright o[e_k]$. This reasoning applies also when o has abstract dependencies in its precondition (still, each basic oclet $o[e]$ gets enabled whenever o is enabled).

Theorem 1 states a rather strict sufficient condition for operational specifications (abstract dependencies only in preconditions). Next, we present a more general sufficient condition: Ω is operational if all abstract dependencies of Ω can be refined by its basic oclets.

Refining abstract runs lifts to oclets: oclet $o_2 = (\pi_2, pre_2)$ refines oclet $o_1 = (\pi_1, pre_1)$, written $o_2 \preceq o_1$, iff $\pi_2 \preceq \pi_1$ and $pre_2 = pre_1$, i.e., we may only refine contributions. An oclet's refinement can be justified by another oclet.

Definition 11 (Justified by oclet). *Let o_1, o_2 be oclets s.t. o_2 refines o_1. The refinement from o_1 to o_2 is* justified *by an oclet o iff $X_{o_2} \setminus X_{o_1} \subseteq X_{con(o)}$, $F_{o_2} \setminus F_{o_1} \subseteq F_{con(o)}$, $A_{o_2} \setminus A_{o_1} \subseteq A_{con(o)}$.*

Let $\Omega = (O, \pi_0)$ be an oclet specification. The refinement from o_1 to o_2 is justified by *Ω iff there is a sequence of refinements from o_1 to o_2 s.t. each refinement is justified by a basic oclet $o \in \hat{O}$ (or by an isomorphic copy o^α of o).*

Theorem 2. *Let $\Omega = (O, \pi_0)$ be a specification. If each oclet $o_1 \in O$ can be refined to an oclet o_2 justified by Ω s.t. all events of o_2's contribution are concrete, then Ω is operational.*

Lemma 1. *Let $\Omega = (O, \pi_0)$ be a specification. Let o be an oclet that can be refined into an oclet o', justified by Ω, s.t. each $e \in E_{con(o')}$ is concrete. Let π be a run s.t. o is enabled in π. Then there exists a sequence $o_1, \ldots, o_n \in \hat{O} \cup \hat{o}$ of basic oclets s.t. $(\pi \rhd o_1 \rhd \ldots \rhd o_n) = (\pi \rhd o') \preceq (\pi \rhd o)$.*

Proof (Lem. 1). Proof by the number $n = |E_{con(o')}|$. For $n = 0$, the proposition holds trivially. For $n > 0$, let $e \in E_{con(o')}$ be a maximal (no other event of o succeeds e).

Case 1: $e \in E_{con(o')}$ and e is concrete. Obtain o_{-e}, o'_{-e} by removing e and $e^\bullet$ from o, o'. $o'_{-e} \preceq o_{-e}$ justified by Ω and $\rho := \pi \rhd o_1 \rhd \ldots \rhd o_{n-1} = \pi \rhd o'_{-e} \preceq \pi \rhd o_{-e}$ by inductive assumption. From e being complete follows $o[e] \in \hat{o}$ and $o[e]$ enabled in ρ. Thus $(\rho \rhd o[e]) = (\pi \rhd o'_{-e} \rhd o[e]) = (\pi \rhd o') \preceq (\pi \rhd o)$.

Case 2: $e \in E_{con(o')}$ and e is not complete, or for some $b \in e^\bullet$, $b \in \max \pi_o$ (s.t. e is added by refining $(x, b) \in A_o$, see Def. 3). By Def. 11, a basic oclet $\tilde{o}$ of Ω with $\widetilde{pre} \subseteq \pi'$ justifies $e \in E_{o'} \cap E_{\tilde{o}}$ and all $(x, e), (e, y) \in F_{o'} \setminus F_o \subseteq F_{con(\tilde{o})}$. Thus, there ex. oclet o'' s.t. $o' \preceq o'' \preceq o$ where $o' \preceq o''$ is justified by $\tilde{o}$ (and the rest by Ω). All events of o'' (except e) are concrete. Obtain $o_{-e}, o'_{-e}, o''_{-e}$ by removing e and $e^\bullet$ from o, o', o''. By construction holds $o''_{-e} = o'_{-e}$ and $\max \widetilde{pre} \subseteq \max \pi'_{-e}$. Refinement $o''_{-e} \preceq o_{-e}$ is justified by Ω, thus $\rho := (\pi \rhd o_1 \rhd \ldots \rhd o_{n-1}) = (\pi \rhd o'_{-e}) \preceq (\pi \rhd o_{-e})$ holds by inductive assumption. Further, $\widetilde{pre} \subseteq \pi'_{-e} \subseteq \rho$ and $\max \widetilde{pre} \subseteq \max \pi'_{-e} \subseteq \max \rho$ holds. Thus, $\tilde{o}$ is enabled in ρ and $(\rho \rhd \tilde{o}) = (\pi \rhd o'_{-e} \rhd \tilde{o}) = (\pi \rhd o') \preceq (\pi \rhd o)$. □

Proof (Thm. 2). By induction on the semantics of $\Omega = (O, \pi_0)$. Let $\hat{\Omega} = (\hat{O}, \pi_0)$. Base: By Def. 8, $\pi_0 \in R(\Omega)$ and $\pi_0 \in R(\hat{\Omega})$. Step: Show for $\pi \rhd o \in R(\Omega)$ (o enabled in π) that there ex. $\rho \in R(\hat{\Omega})$ s.t. $\rho \preceq \pi \rhd o$. By assumption there ex. a refinement $o' \preceq o$ justified by Ω. As $\hat{o} \subseteq \hat{O}$, Lem. 1 implies that $\pi \rhd o' \in R(\hat{\Omega})$ and $(\pi \rhd o') \preceq (\pi \rhd o)$. □

4 Adding Data: Σ-Oclets

In the current model of oclets, conditions and events can be labeled with specific data values. But the language of oclets does not allow to *concisely* describe manipulation of data values and data-dependent enabling of events. In this section, we extend oclets with notions of data. As we adopt techniques that are well-established in several Petri net formalisms, we just describe our approach in an informal manner; for technical details, see [13]. Similar to Sect. 3, we start with a general description of how to specify data.

4.1 Specifying Data

We propose to incorporate data into oclets in the same way as data has been introduced in Place/Transition nets (*P/T nets*) by several classes of *high level* Petri Nets, e.g., in Coloured Petri nets (CPNs) [27]. Recall that in CPNs, a marking distributes concrete values (from one or several domains) on places; expressions on arcs describe which values are consumed or produced by an occurrence of a transition; a *guard* expression at the transition may restrict consumable and producible values further. The method-wise relevant property of CPNs is that each colored net can be *unfolded* w.r.t. all possible interpretations of its expressions into an equivalent P/T net. Then, a (black) token on a P/T net place (p, v) denotes value v on colored place p; likewise each colored transition unfolds to several P/T net transitions defined by the consumed and produced concrete values. A special class of CPNs are Algebraic Petri nets [32] where expressions and values are defined by a *Σ-algebra* with a *signature* Σ.

We adapt the idea of Algebraic Petri nets to oclets and introduce *Σ-oclets*. In a run, a place p carrying the value v is labeled (p, v). In a Σ-oclet, each condition is labeled with a pair (p, t) where p is a name (e.g., of a component) and t is a term (over function symbols and variables of Σ) describing possible values on p. Events are labeled with names of actions as before; an event may carry an additional guard expression (defined over Σ). A system designer can use different terms in the pre- and post-sets of an event to describe how values change by an occurrence of an event.

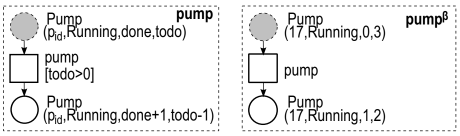


Fig. 6. Σ-oclet pump of the gas station example (left) and an unfolding pump$^\beta$ (right) by assigning each variable a concrete value

Fig. 6 shows Σ-oclet pump of the gas station example; the complete specification is given in Sect. 4.4. Oclet pump describes how the pump at the gas station refuels the customer's car by one unit of fuel and updates its internal records about the provided and the remaining amount of fuel. A Pump's internal record is represented as a 4-tuple. An occurrence of event pump increases done by 1 (3rd entry) and decreases todo by 1 (4th entry). The guard restricts occurrences of pump to those cases where todo $>$ 0. Technically, p_{id}, todo and done are variables and Running is a constant of Σ. As in Petri nets, the semantics of a Σ-oclet o can be understood by unfolding o into a "low-level" oclet that has no terms and variables. The basic idea is to assign a value to each variable in o, and then to replace each term t

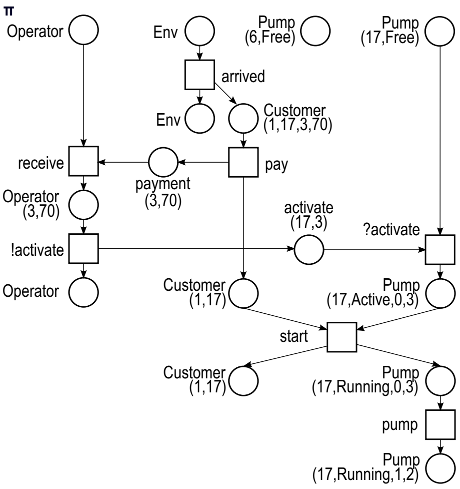


Fig. 7. A distributed run with data

in o by the value obtained by evaluating t. For example, for Σ-oclet pump, the assignment $\beta : \mathsf{p_{id}} \mapsto 17, \mathsf{done} \mapsto 0, \mathsf{todo} \mapsto 3$ yields the classical oclet pump^β shown in Fig. 6. An occurrence of pump^β can be understood as an occurrence of pump in mode β. Other assignments yield other low-level oclets. A Σ-oclet can only be unfolded if all guards evaluate to *true*.

This way an entire set of Σ-oclets O can be unfolded to a (possibly infinite) low-level oclet specification O'. O describes the distributed runs that are described by O', that is, can be constructed from the unfolded oclets. For example, pump^β is enabled in run π of Fig. 7, i.e., oclet pump is enabled for $\mathsf{p_{id}} \mapsto 17$, etc. We can continue π by appending the contribution of pump^β which updates the internal record of the pump. Now pump is enabled for the assignment $\mathsf{todo} \mapsto 2$. Thus, pump can occur two more times, i.e., until the assignment $\mathsf{todo} \mapsto 0$ violates the guard expression.

This notion of data now allows to express *data-dependent behavior*: event pump *repeats* as often as specified by todo. Further, we are now able to distinguish different *instances* of a component. Unlike in run π_2 of Fig. 4, we can now distinguish the two pumps run in π of Fig. 7. This permits to activate exactly the pump that was chosen by the customer.

4.2 Formalization

The formalization of Σ-oclets is straight-forward as it follows exactly the principles of Coloured Petri nets. First, a specification defines an algebraic signature Σ providing sorts (of values), variables, constants and function symbols. We usually assume sorts *Bool* and the usual Boolean operators be given that are interpreted in the usual way. The signature permits to build sorted terms over its symbols and variables. The model of oclets is then extended to Σ-oclets as follows:

1. Label each event and each condition of an abstract distributed run with a pair (a,t) of a name a and a term t over Σ so that the term of an event is of type *Bool* (this terms *guards* the event); such a run is called a *Σ-run*.
2. A *Σ-oclet* is a Σ-run with a distinguished prefix.
3. A *Σ-specification* is a set of Σ-oclets together with an initial run that is assumed to be variable free.

The semantics of Σ-oclets is defined by *unfolding* each Σ-oclet into a set of "low-level" oclets according to Def. 4. Fix a Σ-algebra $\mathcal{A}$ to interpret all sorts, constants, and function symbols. For each oclet o, find an assignment of its variables such that all guards of all events evaluate to *true*, and then replace each term t by its value in $\mathcal{A}$ under this assignment (see Fig. 6). Depending on $\mathcal{A}$, an oclet o can unfold into infinitely many different oclets (each representing a different value). The semantics of the unfolded specification defines the semantics of the Σ-specification. The technical details of this construction are given in [13].

4.3 Operational Semantics

In principle, Σ-oclets gain operational semantics by the operational semantics of their unfolding. However, unfolding a Σ-specification Ω into an (infinite) set of oclets seems impractical to operationalize a Σ-specification. A more practical approach is to directly decompose a Σ-oclet into its basic Σ-oclets by lifting Def. 9 to preserve each node's terms. Then, the operational semantics of Ω are the runs of its basic Σ-specification $\hat{\Omega}$.

While $\hat{\Omega}$ yields semantics based on occurrences of single events, it may introduce spurious behavior (not specified by Ω). This spurious behavior arises if a basic Σ-oclet $o[e]$ of o unfolds to some basic low-level oclet that is not defined by the unfolding of entire o. For instance, consider a Σ-oclet o with two events, where event e_1 carries the guard $(x > 0)$, event e_2 carries the guard $(x < 5)$, and $e_1 \leq e_2$. If x is assigned 17, then $o[e_1]$ may occur while $o[e_2]$ may not – the operational semantics would "get stuck" in the middle of o after e_1. To avoid such behavior, we demand that each Σ-oclet $o \in \Omega$ is *data-consistent*. To this end, o should exhibit two properties: (1) two distant nodes only carry variables that also occur in their joint predecessors, and (2) two guards of events of o do not contradict each other. For Σ-specifications that contain only data-consistent oclets, the operational semantics can be implemented; details are given in [13].

For such specifications, it is sufficient to check for a basic Σ-oclet $o[e]$ first, whether its pre-condition occurs at the end of a run π (ignoring the particular values of π), and then to check whether the variables in $o[e]$ can be bound in a way that the terms in $o[e]$ match the values in π.

4.4 Complete Gas Pump Example

Fig. 8 shows the complete set O of Σ-oclets for the gas pump example including its initial run init. The signature Σ for the Σ-specification $\Omega = (O, \mathsf{init}, \mathcal{A})$ is Boolean and also contains the theory of integers with its usual interpretation, and additionally defines constant symbols $\mathsf{Price}, 6, 17$ of sort integer. Further, it has a sort containing the constant symbols Free, Active and $\mathsf{Running}$. The variables are $V = V_{Nat} = \{\$, \mathsf{change}, \mathsf{c_{id}}, \mathsf{done}, \mathsf{p_{id}}, \mathsf{todo}, \mathsf{left}\}$. The initial run init specifies an environment (Env), the operator, and two inactive pumps with ids 6 and 17.

Oclets prepay (when the customer pays in advance) and pay later (when the customer pays at the end) describe the system behavior at the most abstract level. All other oclets but env justify a refinement of main. The specification is data-consistent, and the run of Fig. 7 is a run of this specification.

Oclet env describes the arrival and leaving of a new customer; technically a new *instance* of Customer is created with id ($\mathsf{c_{id}}$), the amount the customer wants to refuel (todo), the id of the pump he wants to use ($\mathsf{p_{id}}$) and his payment (\$). Only customers who can pay their desired amount of gas may participate. A customer instance is destroyed after the customer received his change.

When the customer paid or ordered his amount, the operator activates the pump (see prepay and pay later). The $\mathsf{p_{id}}$ assures to activate the pump that the customer wants to use. The amount that has to be pumped is passed to the pump.

Afterwards, the pump can be started by the customer (see get gas). Then, the pump starts pumping (see pump). The customer may stop at any time, at the latest when all of the amount requested by him was pumped. When the pump finished, the actual amount that was pumped is passed to the operator. A customer who did not pay in the beginning can pay now (see pay later). Then, the change is calculated (see prepay and pay later). Note that because $\mathsf{c_{id}}, \mathsf{p_{id}}, \mathsf{todo}$ and \$ occur in prepay's precondition, it is not necessary

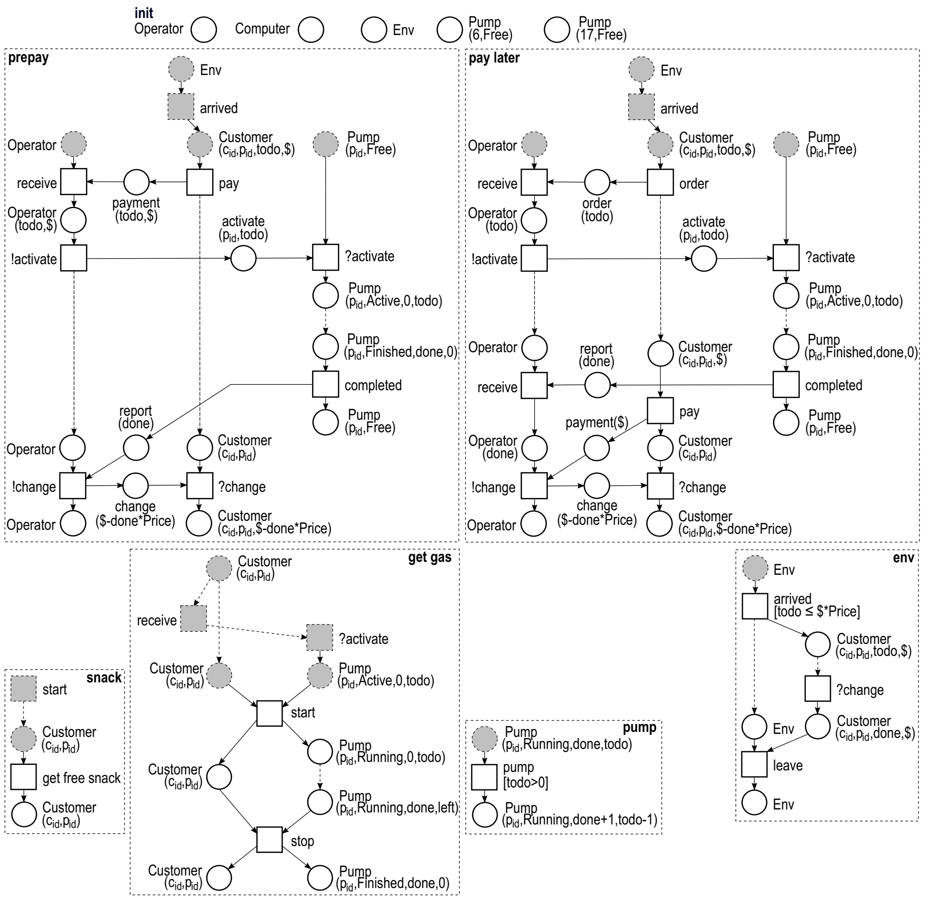

init
Operator
Computer
Env
Pump (6,Free)
Pump (17,Free)
prepay
Env
arrived
Operator
Customer (c_{id},p_{id},todo,$)
Pump (p_{id},Free)
receive
payment (todo,$)
pay
Operator (todo,$)
activate (p_{id},todo)
!activate
?activate
Pump (p_{id},Active,0,todo)
Pump (p_{id},Finished,done,0)
completed
Pump (p_{id},Free)
report (done)
Operator
Customer (c_{id},p_{id})
!change
?change
change ($-done*Price)
Operator
Customer (c_{id},p_{id},$-done*Price)
pay later
Env
arrived
Operator
Customer (c_{id},p_{id},todo,$)
Pump (p_{id},Free)
receive
order
order (todo)
Operator (todo)
activate (p_{id},todo)
!activate
?activate
Pump (p_{id},Active,0,todo)
Operator
report (done)
Customer (c_{id},p_{id},$)
Pump (p_{id},Finished,done,0)
receive
completed
pay
Pump (p_{id},Free)
payment($)
Operator (done)
Customer (c_{id},p_{id})
!change
?change
change ($-done*Price)
Operator
Customer (c_{id},p_{id},$-done*Price)
get gas
Customer (c_{id},p_{id})
receive
?activate
Customer (c_{id},p_{id})
Pump (p_{id},Active,0,todo)
start
Customer (c_{id},p_{id})
Pump (p_{id},Running,0,todo)
Pump (p_{id},Running,done,left)
stop
Customer (c_{id},p_{id})
Pump (p_{id},Finished,done,0)
snack
start
Customer (c_{id},p_{id})
get free snack
Customer (c_{id},p_{id})
pump
Pump (p_{id},Running,done,todo)
pump [todo>0]
Pump (p_{id},Running,done+1,todo-1)
env
Env
arrived [todo ≤ $*Price]
Customer (c_{id},p_{id},todo,$)
?change
Env
Customer (c_{id},p_{id},done,$)
leave
Env

Fig. 8. Complete specification of the gas station example with Σ-oclets

that the operator continuously carries any of these values up to the occurrence of change. Thus, he may start serving a second customer before the first one gets his change back.

Between starting the pump and leaving the gas station, the customer can get one free snack. Oclet snack can occur only once as an occurrence of get free snack after start prevents oclet snack from being enabled.

Tool Support. The approach presented here is implemented in the Eclipse-based tool *Greta* [14]. Greta provides a graphical editor for oclets and animated execution via a simulation engine which implements the operational semantics of oclets including abstract dependencies and Σ-oclets. Greta finds variable assignments and evaluates terms using the simulation engine of CPN Tools via Access/CPN [40]. Additionally, Greta allows to *verify* whether a Petri net implements a given specification, and to *synthesize* a minimal labeled Petri net that implements a specification — both techniques operate on McMillan-prefixes for oclets [12]. Greta is available at `www.service-technology.org/greta`.

5 On the Relation of Oclets to Petri Nets

This section discusses the relation between oclets and Petri nets. As above, oclets without abstraction and data representation are called "classical."

Classical Oclets vs. P/T Nets. Classical oclets contain P/T-nets: for each P/T net N exists a specification Ω s.t. their behaviors are identical: $R(\hat{\Omega}) = R(N)$ [11]. The converse does not hold. Even classical oclets can mimic a Turing machine by their preconditions [12]. However, if Ω is *bounded* (i.e., there exists a k s.t. no run $\pi \in R(\hat{\Omega})$ of Ω has more than k maximal conditions with the same label), then there is a *labeled* Petri net N with the same behavior: $R(N) = R(\hat{\Omega})$. N can be *synthesized* automatically from Ω by first building a McMillan-prefix [10] for Ω that finitely represents $R(\hat{\Omega})$ and then folding that prefix to N [12].

Σ-Oclets vs. P/T Nets. Σ-oclets *extend* classical oclets. Clearly, a Σ-specification Ω with an infinite domain has no equivalent finite P/T net. If Ω has only finite domains, it unfolds to a finite low-level specification $val(\Omega)$. If abstract dependencies in $val(\Omega)$ can be refined s.t. they only occur in pre-conditions, then semantics of classical oclets carries over (Thm. 2). Thus not every Σ-specification has a P/T net with the same behavior. Yet it seems plausible that every *bounded* Σ-specification with finite domains has a net with the same behavior: finitely many oclets will allow to continue markings of finite size only in finitely many ways. This suggests that the synthesis from oclets [12] can also be generalized to Σ-oclets. If abstract dependencies in Ω cannot be refined, the synthesis of a P/T net also has to find a refinement of the abstract dependencies.

Σ-Oclets vs. Algebraic Petri Nets. As synthesizing Algebraic Petri nets from Σ-oclets is out of the scope of this paper, we just sketch some basic observations here. First, every Algebraic net N with term-inscribed arcs and term-inscribed transition guards has an equivalent Σ-specification: translate each transition and its pre- and post-places to a basic oclet by moving arc inscriptions to the respective pre- and post-condition. The reverse direction is more difficult. Term annotations are not an issue as both models are based on the same concepts. But enabledness of an event of a Σ-oclet depends on a "history" in the execution while enabledness of a transition depends on the current marking only. Hee et al. [22] have shown how to express history of tokens in a data-structure of an Algebraic Petri net, and how to use them in guards. Whether all data-dependencies expressible in Σ-oclets can be expressed this way is an open question. Yet, token histories of [22] have drawbacks regarding analysis. Thus, synthesizing a net without an explicit and complete recording of token histories is an interesting, open problem as well. In any case, the signature Σ of a specification has to be extended to allow remembering behavior in the past.

6 Conclusion and Future Work

In this paper, we proposed a formal model for scenarios that combines the advantages of HMSCs (simple semantics by composition of scenarios) with the advantages of LSCs (intuitive notion of scenarios with local preconditions and a flexible style of specifying systems). Our model called *oclets* is based on Petri nets and their distributed runs.

The basic semantic notions of composing LSC-style scenarios to distributed runs have been proposed earlier; in this paper we have shown how the basic model of oclets can be lifted to the expressive means of LSCs by introducing a notion of abstract dependencies and adopted concepts from Algebraic Petri nets to represent data. All of our extensions are constructed such that (1) they generalize and embed the "classical" oclet concept and (2) their semantics can be described in terms of classical oclets. Existing operational semantics for oclets canonically lift to our extended model. Oclets deviate from LSCs where their semantics turns problematic for the aim of deriving specified behavior by composing scenarios (see Sect. 2). Our approach is implemented in the tool Greta and was validated on a number of elaborate examples. Finally, composition, decomposition, abstraction, refinement, and unfolding suggest oclets to be an interesting model for a *calculus* of scenarios. The contribution of this model of scenarios becomes obvious when considering existing works that relate scenario-based techniques to Petri nets.

From Scenarios to Petri Nets. To bridge the gap between scenario-based specifications and Petri nets, several approaches have been proposed. Methods to transform UML sequence diagrams to Coloured Petri nets (CPNs) are described in [5], [16], and [41]; the latter approach is used in [15] to model a variant of the gas station example used in this paper. Further, there exist approaches to provide Petri net semantics for MSCs [25,29] and to synthesize a Petri net out of a MSC specification [36]. While these approaches are straight-forward, the scenario languages lack expressive power (MSCs, UML sequence diagrams) or tend to yield complex specifications in practice (HMSCs, see Sect. 2).

Approaches to transform a LSC specification to a CPN have been described in [1] and [28]. As two LSCs of a specification may be *contradictory* [20], both synthesis approaches need to generate the state space of the LSC and the synthesized CPN to check equivalency of both models. Also *operational semantics* of LSCs [20] requires model checking to find a correct play-out step. We have shown in this paper that Σ-oclets do not require model checking for operational semantics. The gives rise to the hope that components can be synthesized from Σ-oclets as correct by construction (as in the case of HMSCs).

To the best of our knowledge, no other Petri net-based model for scenarios features data and abstract causal dependencies. The notion of history-dependent behavior in Petri nets was introduced in [22]. Oclets particulary relate to *Token History Petri nets* where each token records its "traveling" through the net; LTL-past guards at transitions restrict enabledness to tokens with a particular history. The scenario-based approach of oclets provides a graphical syntax for a subclass of these guards. Particularly, Token History Petri nets might allow to synthesize components from an oclet specification. There are numerous refinement and abstraction techniques for Petri net system models, e.g., by modular refinement [39] or using rules [3]. Refinement of actions of a distributed run has been studied in [18]; we think these results can be lifted to refine abstract dependencies in oclets in a systematic way.

Future Work. A next step for research work is to develop *symbolic* semantics for Σ-oclets allowing to concisely describe infinitely many behaviors. This should support solving the main challenge of scenarios: to synthesize high level Petri net components from a Σ-specification without unfolding into a concrete low-level model.

Acknowledgements. The research leading to these results has received funding from the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement n^o 257593 (ACSI).

References

1. Amorim, L., Maciel, P.R.M., Nogueira Jr., M.N., Barreto, R.S., Tavares, E.: Mapping live sequence chart to coloured petri nets for analysis and verification of embedded systems. ACM SIGSOFT Software Engineering Notes 31(3), 1–25 (2006)
2. 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)
3. Berthelot, G.: Checking Properties of Nets Using Transformation. In: Rozenberg, G. (ed.) APN 1985. LNCS, vol. 222, pp. 19–40. Springer, Heidelberg (1986)
4. Bontemps, Y., Schobbens, P.Y.: The computational complexity of scenario-based agent verification and design. Journal of Applied Logic 5(2), 252–276 (2007)
5. Bowles, J., Meedeniya, D.: Formal transformation from sequence diagrams to coloured petri nets. In: Han, J., Thu, T.D. (eds.) APSEC, pp. 216–225. IEEE Computer Society (2010)
6. Damm, W., Harel, D.: LSCs: Breathing Life into Message Sequence Charts. Form. Methods Syst. Des. 19(1), 45–80 (2001)
7. 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)
8. Desel, J., Erwin, T.: Hybrid specifications: looking at workflows from a run-time perspective. Computer Systems Science and Engineering 15(5), 291–302 (2000)
9. Desel, J.: Validation of process models by construction of process nets. In: Business Process Management, pp. 110–128 (2000)
10. Esparza, J., Heljanko, K.: Unfoldings - A Partial-Order Approach to Model Checking. Springer (2008)
11. Fahland, D.: Oclets – Scenario-Based Modeling with Petri Nets. In: Franceschinis, G., Wolf, K. (eds.) PETRI NETS 2009. LNCS, vol. 5606, pp. 223–242. Springer, Heidelberg (2009)
12. Fahland, D.: From Scenarios To Components. Ph.D. thesis, Humboldt-Universität zu Berlin (2010), http://repository.tue.nl/685341
13. Fahland, D., Prüfer, R.: Data and Abstraction for Scenario-Based Modeling with Petri Nets. Technical Report 12-07, Eindhoven University of Technology (2012)
14. Fahland, D., Weidlich, M.: Scenario-based process modeling with Greta. In: BPM Demos 2010. CEUR-WS.org, vol. 615, pp. 52–57 (2010)
15. Fernandes, J.M., Tjell, S., Jørgensen, J.B.: Requirements engineering for reactive systems with coloured petri nets: the gas pump controller example. In: Jensen, K. (ed.) 8th Workshop on Practical Use of Coloured Petri Nets and the CPN Tools, Aarhus (October 2007)
16. Fernandes, J.M., Tjell, S., Jørgensen, J.B., Ribeiro, O.: Designing tool support for translating use cases and uml 2.0 sequence diagrams into a coloured petri net. In: Proceedings of the Sixth International Workshop on Scenarios and State Machines, SCESM 2007, p. 2. IEEE Computer Society, Washington, DC (2007), http://dx.doi.org/10.1109/SCESM.2007.1
17. Genest, B., Minea, M., Muscholl, A., Peled, D.A.: Specifying and Verifying Partial Order Properties Using Template MSCs. In: Walukiewicz, I. (ed.) FOSSACS 2004. LNCS, vol. 2987, pp. 195–210. Springer, Heidelberg (2004)
18. van Glabbeek, R.J., Goltz, U.: Refinement of actions and equivalence notions for concurrent systems. Acta Inf. 37(4/5), 229–327 (2001)
19. Harel, D., Kugler, H.: Synthesizing State-Based Object Systems from LSC Specifications. In: Yu, S., Păun, A. (eds.) CIAA 2000. LNCS, vol. 2088, pp. 1–33. Springer, Heidelberg (2001)

20. Harel, D., Kugler, H., Marelly, R., Pnueli, A.: Smart Play-Out of Behavioral Requirements. In: Aagaard, M.D., O'Leary, J.W. (eds.) FMCAD 2002. LNCS, vol. 2517, pp. 378–398. Springer, Heidelberg (2002)
21. Harel, D., Marelly, R.: Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine. Springer-Verlag New York, Inc., Secaucus (2003)
22. van Hee, K.M., Serebrenik, A., Sidorova, N., van der Aalst, W.M.P.: Working with the past: Integrating history in petri nets. Fundam. Inform. 88(3), 387–409 (2008)
23. Heimbold, D., Luckham, D.: Debugging Ada Tasking Programs. IEEE Softw. 2, 47–57 (1985)
24. Hélouët, L., Jard, C., Caillaud, B.: An event structure based semantics for high-level message sequence charts. Math. Structures in Comp. Sci. 12(4), 377–402 (2002)
25. Heymer, S.: A semantics for msc based on petri net components. In: Sherratt, E. (ed.) SAM. VERIMAG, IRISA, SDL Forum (2000)
26. ITU-T: Message Sequence Chart (MSC). Recommendation Z.120, International Telecommunication Union, Geneva (2004)
27. Jensen, K., Kristensen, L.M.: Coloured Petri Nets - Modelling and Validation of Concurrent Systems. Springer (2009)
28. Khadka, B., Mikolajczak, B.: Transformation from live sequence charts to colored petri nets. In: Wainer, G.A. (ed.) SCSC, pp. 673–680. Simulation Councils, Inc. (2007)
29. Kluge, O.: Modelling a Railway Crossing with Message Sequence charts and Petri Nets. In: Ehrig, H., Reisig, W., Rozenberg, G., Weber, H. (eds.) Petri Net Technology for Communication-Based Systems. LNCS, vol. 2472, pp. 197–218. Springer, Heidelberg (2003)
30. Mauw, S., Reniers, M.A.: An algebraic semantics of Basic Message Sequence Charts. The Computer Journal 37, 269–277 (1994)
31. Mukund, M., Narayan Kumar, K., Thiagarajan, P.S.: Netcharts: Bridging the Gap between HMSCs and Executable Specifications. In: Amadio, R.M., Lugiez, D. (eds.) CONCUR 2003. LNCS, vol. 2761, pp. 296–310. Springer, Heidelberg (2003)
32. Reisig, W.: Petri Nets and Algebraic Specifications. Theor. Comput. Sci. 80(1), 1–34 (1991)
33. Reisig, W.: Elements of distributed algorithms: modeling and analysis with Petri nets. Springer (1998)
34. Ren, S., Rui, K., Butler, G.: Refactoring the Scenario Specification: A Message Sequence Chart Approach. In: Masood, A., Léonard, M., Pigneur, Y., Patel, S. (eds.) OOIS 2003. LNCS, vol. 2817, pp. 294–298. Springer, Heidelberg (2003)
35. Sengupta, B., Cleaveland, R.: Triggered message sequence charts. IEEE Trans. Software Eng. 32(8), 587–607 (2006)
36. Sgroi, M., Kondratyev, A., Watanabe, Y., Lavagno, L., Sangiovanni-Vincentelli, A.: Synthesis of petri nets from message sequence charts specifications for protocol design. In: DASD 2004 (2004)
37. Sibay, G., Uchitel, S., Braberman, V.A.: Existential live sequence charts revisited. In: ICSE 2008, pp. 41–50. ACM (2008)
38. Uchitel, S., Kramer, J., Magee, J.: Synthesis of Behavioral Models from Scenarios. IEEE Transactions on Software Engineering 29, 99–115 (2003)
39. Vogler, W.: Modular Construction and Partial Order Semantics of Petri Nets. LNCS, vol. 625. Springer, Heidelberg (1992)
40. Westergaard, M.: Access/CPN 2.0: A High-Level Interface to Coloured Petri Net Models. In: Kristensen, L.M., Petrucci, L. (eds.) PETRI NETS 2011. LNCS, vol. 6709, pp. 328–337. Springer, Heidelberg (2011)
41. Yang, N., Yu, H., Sun, H., Qian, Z.: Modeling uml sequence diagrams using extended petri nets. Telecommunication Systems 48, 1–12 (2011), http://dx.doi.org/10.1007/s11235-011-9424-5, doi:10.1007/s11235-011-9424-5

Maximal Confluent Processes

Xu Wang

International Institute of Software Technology, United Nations University
P.O. Box 3058, Macau
wx@iist.unu.edu

Abstract. In process semantics of Petri Net, a non-sequential process is a concurrent run of the system represented in a partial order-like structure. For transition systems it is possible to define a similar notion of concurrent run by utilising the idea of confluence. Basically a confluent process is an acyclic confluent transition system that is a *partial unfolding* of the original system. Given a non-confluent transition system G, how to find maximal confluent processes of G is a theoretical problem having many practical applications.

In this paper we propose an unfolding procedure for extracting maximal confluent processes from transition systems. The key technique we utilise in the procedure is the construction of *granular configuration structures* (i.e. a form of event structures) based on diamond-structure information inside transition systems.

1 Introduction

Confluence is an important notion of transition systems. Previously there has been extensive work devoted to its study, e.g. [10,7,6,9]. In [7] confluence is studied from the perspective of non-interleaving models, where it was concluded that in order to characterise the class of confluent transition systems the underlying event-based models needs to support the notion of *or-causality* [16,19].

In this paper we are going to study the idea of maximal confluent sub-systems of a non-confluent transition system, also from a non-interleaving perspective. It can be regarded as an extension of the notion of non-sequential pocesses in Petri Net [5,2,3] onto transition systems. We call it *maximal confluent process* (MCP). Intuitively a maximal confluent process is a concurrent run of the system that is maximal both in length and in degree of concurrency. A non-confluent system has multiple such runs. Non-maximal concurrent runs can be deduced from maximal ones, e.g. by restricting concurrency (i.e. strengthening causality relation).

Like non-sequential processes, which can be bundled together to form *branching processes* of Petri Net, the set of maximal confluent processes (extracted from a given transition system) can coalesce into a *MCP branching processes* of the original system. Such branching processes record, in addition to causality information, also the 'choice points' of the system at which different runs split from each other. In a non-interleaving setting the 'choice points' are formalised as *(immediate) conflicts* on events. The arity of the conflicts can be *non-binary*, thus giving rise to the so called *finite conflicts*. For instance, in state $s0$ of Figure 1 actions a, b and c form a ternary conflict, which induces the three maximal concurrent runs of the system (i.e. the three subgraphs on the right).

In this paper we propose an unfolding procedure to construct *granular configuration structures* from transition systems. The procedure preserves the maximality of confluence in such a way that each generated configuration corresponds to a prefix of some maximal concurrent run. Configuration structures are an event structure represented in a global-state based fashion [15,14]. They support or-causlity as well as finite conflicts.

2 Motivating Examples

We first look at two examples in order to build up some intuitions for maximal confluent processes.

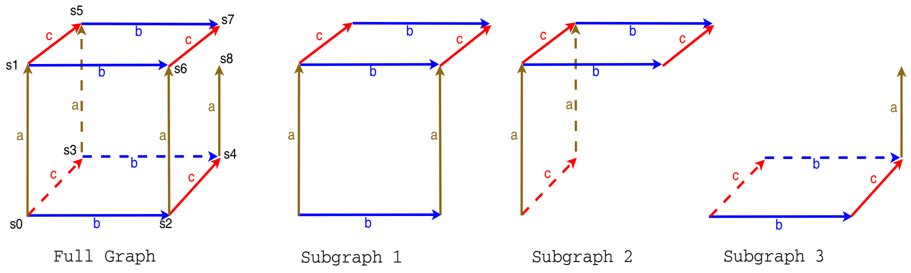


Fig. 1. A running example

The first example is the left-most graph in Figure 1, which is an LTS in the shape of a *broken cube* (i.e. replacing transition $s_4 \xrightarrow{s} s_8$ by $s_4 \xrightarrow{s} s_7$ will give rise to a true cube-shaped LTS). The three subgraphs on its right are confluent subgraphs of the broken cube. Moreover, they are maximal such subgraphs; adding any state or transition to them will invalidate their confluence. They are exactly the maximal confluent processes we are looking for.

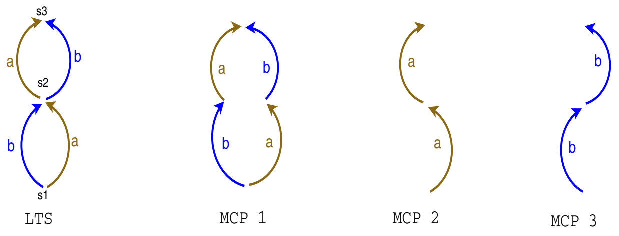


Fig. 2. The second example

For the general cases, however, maximal confluent processes do not coincide with maximal confluent subgraphs. Let us look at the left-most LTS in Figure 2. The maximal confluent subgraphs of such system are the four maximal simple paths in the graph, i.e. $s_1 \xrightarrow{a} s_2 \xrightarrow{a} s_3$, $s_1 \xrightarrow{a} s_2 \xrightarrow{b} s_3$, $s_1 \xrightarrow{b} s_2 \xrightarrow{a} s_3$, and

$s_1 \xrightarrow{b} s_2 \xrightarrow{b} s_3$. But its maximal confluent processes have three members, MCP 1-3 in Figure 2. Two subgraphs $s_1 \xrightarrow{a} s_2 \xrightarrow{b} s_3$ and $s_1 \xrightarrow{b} s_2 \xrightarrow{a} s_3$ are combined into one process, MCP 1. MCP 1 is not a subgraph because state s_2 of the original LTS is split into two states.

The idea of maximal confluent processes has interesting applications. The extraction of maximal confluent processes from a given transition system can be regarded as a deep form of commutativity analysis on the system, which is fully dynamic (i.e. state-dependent) and global (i.e. checking infinite number of steps into future). For instance, they can be used in partial order reduction [11,4,13] to define a canonical notion of optimal reduction, *weak knot* [8]. The challenge, however, lies in how to find a procedure that uses only *local* diamond-structure information inside transition systems to extract maximal confluent processes.

Now let us develop a formal framework to study the problem.

3 Maximal Confluent Processes

Definition 1. *A* transition system *(TS) is a 4-tuple* $(S, \Sigma, \Delta, \hat{s})$ *where*

– S *is a set of* states*,*
– Σ *is a finite set of* actions *(ranged over by a, b, etc.),*
– Δ *is a partial function from* $S \times \Sigma$ *to* S *(i.e. the* transition function*)*[1]*, and*
– $\hat{s} \in S$ *is the* initial state*.*

Fix a TS, $G = (S, \Sigma, \Delta, \hat{s})$, and define:

– a transition $t = s \xrightarrow{a} s'$ means $(s, a, s') \in \Delta$;
– a consecutive sequence of transitions $L = s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} s_2 \ldots s_{n-1} \xrightarrow{a_n} s_n$ means $s_{i-1} \xrightarrow{a_i} s_i$ for all $1 \leq i \leq n$. L is called an *execution* (i.e. sequential run) of G from s_0 to s_n producing *trace* $a_1 \cdots a_n$. When $s_0 = \hat{s}$ we further call it a *system execution* of G; and we use $\mathcal{L}(G)$ to denote the set of system executions of G.
– $s \xrightarrow{a_1 \cdots a_n} s'$ means there exists an execution of G from s_0 to s_n producing trace $a_1 \cdots a_n$;
– $s \xrightarrow{a}$ means $\exists s' : s \xrightarrow{a} s'$, while $s \not\xrightarrow{a}$ means $\neg s \xrightarrow{a}$;
– $eb(s)$ denotes the set of actions enabled at s, i.e. $\{a \mid s \xrightarrow{a}\}$;
– $Reach(s)$ denotes the set of states reachable from s;
– given any $s \in Reach(\hat{s})$, $G/s = (S, \Sigma, \Delta, s)$ denotes the new transition system generated after the evolution to s;
– and if G is acyclic, we further define:
  - $s \sqsubseteq s'$ means $s' \in Reach(s)$, i.e. s' is a *subsequent* state of s (or s is an *earlier* state of s');
  - given any $X \subseteq S$, $min(X)$ denotes the set of $\sqsubseteq$-minimal states inside X, while $X^{\downarrow}$ denotes the $\sqsubseteq$-downward closure of X;

[1] Note that our transition systems are actually deterministic transition systems in the classical sense. It gives us simplicity in theory presentation while at the same time sacrificing few technical insights.

- $s \,||_{\sqsubseteq}\, s'$ means s and s' are incomparable w.r.t. $\sqsubseteq$;
- and given any $s \in \mathit{Reach}(\hat{s})$, s/G denotes the restriction of G to $\{s\}^{\downarrow}$, i.e. $s/G = (\{s\}^{\downarrow}, \Sigma, \{(s_0, a, s_1) \mid (s_0, a, s_1) \in \Delta \wedge s_0, s_1 \in \{s\}^{\downarrow}\}, \hat{s})$.

When there is any danger of confusion, we use $\rightarrow_G$ to say the transitions come from a TS named G. Similarly we use S_G for the set of states and $\hat{s}_G$ for the initial state of G.

TSes can be related to each other by *partial unfolding* relation:

– We say G is a partial unfolding of G' if there exists a function f from S_G to $S_{G'}$ such that $f(\hat{s}_G) = \hat{s}_{G'}$ and $s \xrightarrow{a}_G s' \implies f(s) \xrightarrow{a}_{G'} f(s')$.

As its name suggests partial unfolding unwinds just part of a transition system. When f is injective, partial unfolding is reduced to *subgraph* relation. In the rest of the paper, whenever the homomorphism f of any subgraph relation is left unspecified, we assume f is the *identity function.*

Of cause, we can also fully unwind TSes, giving rise to the *unfolding* relation:

– We say G is an unfolding of G' if G is a partial unfolding of G' and, for all system execution $\hat{s}_{G'} \xrightarrow{a_1}_{G'} s'_1 \xrightarrow{a_2}_{G'} s'_2 \ldots s'_{n-1} \xrightarrow{a_n}_{G'} s'_n$ of G', there is a system execution $\hat{s}_G \xrightarrow{a_1}_G s_1 \xrightarrow{a_2}_G s_2 \ldots s_{n-1} \xrightarrow{a_n}_G s_n$ of G s.t. $s'_i = f(s_i)$ for all $1 \leq i \leq n$.

Of all TSes, a particular interesting subclass of TSes is *confluent TSes.*

– G is confluent if, for all $s \in S_G$ and $a, b \in eb_G(s)$ (with $a \neq b$), a and b form a local diamond at s, i.e. $\exists s_3 \in S_G : s \xrightarrow{ab}_G s_3 \wedge s \xrightarrow{ba}_G s_3$.

In the rest of the paper we use $a \diamond_s b$ to denote a diamond rooted at s and built from a and b actions. The notation can be extended to multi-dimension diamonds. We use $\diamond_s A$ to denote a n-dimension (where $n = |A|$) diamond rooted at s and built from members of A, i.e. given any $B \subseteq A$, there exists a unique $s' \in S$ such that $s \xrightarrow{a_1 \cdots a_m} s'$ for all permutation $a_1 \cdots a_m$ of B.

It is interesting to note that all local diamonds inside a partial unfolding are inherited from those of the original TS. They are the unwinded versions of the original diamonds (c.f. MCP 1 in Figure 2). Furthermore, since a partial unfolding can visit a state of the original TS more than once (esp. when the original TS is cyclic), we can choose to unwind a different diamond on subsequent visits to the state.

Now we are ready to define the notion of concurrent runs of TSes:

– We say an acyclic confluent TS F is a *confluent process* of G if F is a partial unfolding of G.

A confluent process F can be finite or infinite. For a finite confluent process F, it has a unique maximal state, denoted $\check{s}_F$.

– We say a confluent process F of G is a *maximal confluent process* (MCPs) if F is maximal w.r.t. partial unfolding relation, i.e. F is a partial unfolding of another confluent process F' implies F' and F are isomorphic[2].

When restricted to confluent processes, partial unfolding relation is reduced to subgraph relation (c.f. the lemma below). Thus MCPs are 'maximal confluent subgraphs'. In addition, there is a unique minimal confluent subgraph of G, denoted $\hat{G}$. $\hat{G}$ is the trivial TS with a single state and empty transition function, i.e. $\hat{G} = (\{\hat{s}_G\}, \Sigma, \{\}, \hat{s}_G)$.

Lemma 1. *Given two confluent processes F and F' of G, F is a partial unfolding of F' implies the homomorphism f between F and F' is injective.*

In the rest of the paper we will use $\preceq$ to denote subgraph relation on confluent processes. Relation $\preceq$ allows a confluent process to be reduced in two different dimensions: the degree of concurrency and the length of causality chains. Thus MCPs represent the longest possible runs of the system in a maximally concurrent fashion. In a transition system with cycles that implies MCPs are often infinite graphs: finite MCPs are those derived from terminating runs (i.e. ending in a state where there is no outgoing transitions).

More refined relations on confluent processes that reduces only one of the dimensions can also be defined:

– Given two confluent processes F and F', we say F is a (concurrency) *tightening* of F' (or F' is a *relaxation* of F), denoted $F \preceq_r F'$, if F is a subgraph of F' and, for all $s \in S_F$ and $a \in eb_{F'}(s)$, there exists a subsequent state $s' \in S_F$ of s s.t. $a \in eb_F(s')$.
– Given two confluent processes F and F' of G, we say F is a *prefix* of F' (or F' is an *elongation* of F'), denoted $F \preceq_e F'$, if F is a subgraph of F' and there exists a function p from S_F to 2^{Σ} (i.e. *the pending action function*) s.t. $s \in S_F \implies p(s) = eb_{F'}(s) \setminus eb_F(s)$ and $s \xrightarrow{a}_F s' \implies p(s) \subseteq p(s')$.

Subgraph relation is decomposable into the two refined relations.

Lemma 2. *$F \preceq F''$ 1) iff there exists some F' s.t. $F \preceq_r F' \preceq_e F''$ and 2) iff there exists some F' s.t. $F \preceq_e F' \preceq_r F''$.*

The intuition behind the refined relations can better be understood using the notion of 'events'.

– Given a confluent process F, we say a state $s \in S_F$ is the *origin* of an action occurrence, say a, if $a \in eb_F(s)$ and $s_0 \xrightarrow{b}_F s \implies a = b \vee a \notin eb_F(s_0)$.
– An occurrence of a with origin s gives rise to a *granular event*, denoted T, which is the set of a-transition reachable from s by firing only non-a transitions in F. Conversely, given T we use $lb_F(T)$ and $o_F(T)$ to denote its label a and origin s resp.

[2] In the rest of the paper we will freely use F to denote an acyclic confluent graph or to denote its isomorphism class.

- Given two confluent processes $F \preceq F'$ and two granular events T in F and T' in F', we say T and T' are the *same event* if $T \subseteq T'$ and $o_F(T) = o_{F'}(T')$; and we say T is a *postponed occurrence* of T' if $T \subseteq T'$ and $o_F(T) \neq o_{F'}(T')$.
- We say two granular events T and T' of F are *or-causally coupled* if $T \cap T' \neq \{\}$.

The or-causal coupling relation is reflexive and symmetric. Its transitive closure, which is an equivalence relation, can be used to partition the set of granular events in F. That is, each equivalence class $\mathcal{E}$ gives rise an *event* $T = \bigcup_{T_0 \in \mathcal{E}} T_0$. Note that an event does not have a unique origin; thus we replace $o_F(T)$ by $O_F(T)$ to denote its set of origins.

- Given two confluent processes $F \preceq F'$ and two events T in F and T' in F', we say T and T' are the *same event* if $T \subseteq T'$ and $O_F(T) = O_{F'}(T') \cap S_F$, and we say T is a *delayed occurrence* of T' if $T \subseteq T'$ and $O_F(T) \neq O_{F'}(T') \cap S_F$.

Based on the notions of events we can see that tightening on F' *delays* (but not removes) events in F' while prefixing on F' *removes* (but not delays) events inside F'.

One fact noteworthy is that, as we elongate a confluent process, events can become 'enlarged' through the addition of new granular events (even though they remain the same events). However, this addition has an upper-limit as the 'size' of an event will eventually stablise.

Lemma 3. *Given a strictly increasing (w.r.t. $\preceq_E$) infinite sequence of confluent processes $F_0\, F_1 \ldots F_i \ldots$, T is an event in F_i implies there exists some $j \geq i$ and event T' in F_j s.t. T and T' are the same event and T' is* stablised *at j, i.e. for any $n \geq j$, T'' of F_n is the same event as T' implies $O_{F_j}(T') = O_{F_n}(T'')$.*

Some further facts about the refined relations are:

Lemma 4. *Given two finite confluent processes $F \preceq F'$, we have 1) $F \preceq_r F'$ iff $\check{s}_F = \check{s}_{F'}$, and 2) $F \preceq_e F'$ iff $F = \check{s}_F / F'$.*

In another word the set of prefixes of F' corresponds 1-1 to the set of states of F'.

- A confluent process F is said to be a *maximally relaxed process* (i.e. MRP) if F is maximal w.r.t. $\preceq_r$.
- A confluent process F is said to be an *MCP prefix* if there exists an MCP F' s.t. F is a prefix of F'. MCP prefixes are the initial parts of complete maximally-concurrent runs.

Naturally one can imagine that MCPs are generated step by step by unfolding local diamonds in the states it visits; MCPs usually prefers to unfold larger diamonds in each step. However, the maximality of MCPs, unlike diamonds, is a global property. Sometimes choosing a strictly smaller diamond to unfold at an early state might lead to a larger diamond in subsequent states. This phenonmenon is similar to the phenonmenon of *confusion* in Petri Net.

- Given a state $s \in S_G$, we say $A \subseteq \Sigma$ is an *MCP step* (MPS) at s if there exists a MCP F of G s.t. $\exists s_F \in S_F : f(s_F) = s \wedge eb_F(s_F) = A$.

Given a state s, the set of its MPSes are not necessarily downward closed or mutually imcomparable (w.r.t subsethood). As an example, imagine an event structure with four events $e1$, $e2$, $e3$ and $e4$ labelled by action a, b, c and d resp. $e3$ and $e4$ causally depend on $e2$ while $e3$ is in conflict with $e1$. In the transition system generated by the event structure, a and b form a maximal diamond at the initial state. However, taking the $a \diamond b$ diamond will destroy the future $c \diamond d$ diamond which is reachable by taking the b action only. Thus $\{a, b\}$ and $\{b\}$ are both MPSes at the initial state whilst $\{a\}$ is not.

Similarly we can see that not all MCP prefixes are MRPs, even though all MRPs are MCP prefixes:

Lemma 5. *A confluent process F is an MRP implies F is an MCP prefix.*

Maximal confluence is a global property which is generally hard to establish. However, once established, the property is *preserved* by system evolutions:

Lemma 6. *1) F is an MCP of G implies F/s_F is an MCP of $G/f(s_F)$ for all $\hat{s}_F \xrightarrow{a}_F s_F$; 2) $\hat{s}_G \xrightarrow{a}_G s_G$ and F' is an MCP of G/s_G implies there exists an (not necessarily unique) MCP F of G s.t. $\hat{s}_F \xrightarrow{a}_F s_F$, $s_G = f(s_F)$ and $F/s_F = F'$.*

Now we can develop the notion of *maximal back-propagation* that will form the basis of our unfolding procedure in the next section.

- If F is a confluent process of G/s, then we say there is a *concurrent run F from s*, denoted $s \overset{F}{\Rightarrow}_G$. If F is finite and $f(\check{s}_F) = s' \in S_G$, we further say that there is a *concurrent run F from s to s'*, denoted $s \overset{F}{\Rightarrow}_G s'$.
- We say an action $a \in \Sigma$ is *fired* in a concurrent run $s \overset{F}{\Rightarrow}_G$ if an a-labelled transition is reachable in F. We say an action $a \in \Sigma$ is *blocked* in a concurrent run $s \overset{F}{\Rightarrow}_G$ if there exists a path $\hat{s}_F \xrightarrow{a_1 \cdots a_{n-1}}_F s_F \xrightarrow{a_n}_F s'_F$ in F s.t. $a_1, \cdots, a_n \in \Sigma \setminus \{a\}$ and $a \diamond_{f(s_F)} a_n$ does not hold in G; otherwise we say a is *unblocked* in $s \overset{F}{\Rightarrow}_G$.
- Given a confluent process F of G, we say an action $a \in \Sigma \setminus eb_F(s_F)$ is *postponed* at s_F (or, more accurately, the potential granular event with label a and origin s_F has postponed occurrence in F) if a is unblocked in F/s_F and there exists a granular event T' in F s.t. $lb(T') = a$ and $o(T') \sqsupset s_F$.
- Furthermore, we say $a \in \Sigma \setminus eb_F(s_F)$ is *p-pending* (partially pending) at $s_F \in S_F$ if a is unblocked but fired in $F/s_F(a)$ and there is no granular event T' in F s.t. $lb(T') = a$ and $o(T') \sqsupset s_F$, and we say $a \in \Sigma \setminus eb_F(s_F)$ is *pending* at $s_F \in S_F$ if a is neither fired nor blocked in $F/s_F(a)$.

For instance, given G (the leftmost graph) and its three confluent processes in the figure below, we can see c is pending at $s1$, p-pending at $s2$ and postponed at $s1'$ in the three confluent processes resp.

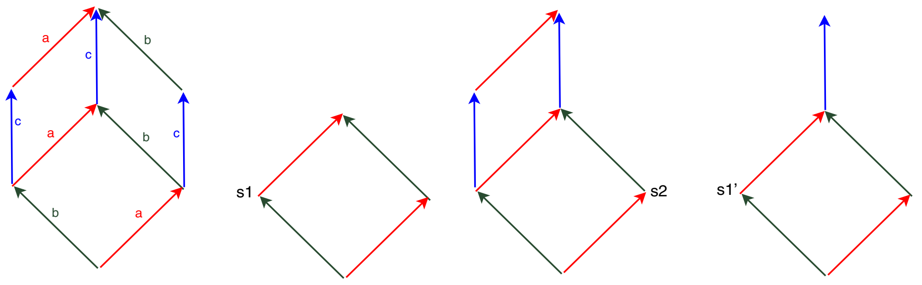


Fig. 3. Pending, p-pending and postponed

- A confluent process F is called *primary confluent process* (PCP) if no action is postponed at any state in F.
- Given a finite confluent process F and an action $a \in eb_G(f(\check{s}_F))$, we define the *pending back-propagation* of a over F to be $bp_{pn}(a, F) = \{s_F \in S_F \mid a$ is pending at $s_F\}$, and the *maximal pending back-propagation* of a over F to be $mbp_{pn}(a, F) = min_F(bp_{pn}(a, F))$.
- Similarly, given an action $a \in \Sigma$ which is p-pending at some state of F, we define the *p-pending back-propagation* of a over F to be $bp_{pp}(a, F) = \{s_F \in S_F \mid a$ is p-pending at $s_F\}$, and the *maximal p-pending back-propagation* of a over F to be $mbp_{pp}(a, F) = min_F(bp_{pp}(a, F))$.

Lemma 7. *A confluent process F is an MRP iff there is no postponed or p-pending action at any $s_F \in S_F$.*

Lemma 8. *Given an action $a \in \Sigma$ postponed or p-pending at a state s_F of F, there exists a unique minimal relaxation F' of F, denoted $F' = F{\uparrow}^a_{s_F}$, s.t. $a \in eb_{F'}(s_F)$.*

Lemma 9. *Given an action $a \in eb_G(f(\check{s}_F))$ and a state $s_F \in bp_{pn}(a, F)$, there exists a unique minimal elongation F' of F, denoted $F \overset{a}{\rightsquigarrow}_{s_F} F'$, s.t. $a \in eb_{F'}(s_F)$.*

Theorem 1. *Given a finite primary confluent process F, if $s_F \in mbp_{pp}(a, F) \wedge F' = F \uparrow^a_{s_F}$ or $s_F \in mbp_{pn}(a, F) \wedge F \overset{a}{\rightsquigarrow}_{s_F} F'$, then F' is a primary confluent process.*

4 Coalescing Confluent Processes

A confluent process records one possible history of system evolution. To see other possible evolutions and pinpoint where different evolutions come to deviate and split from each other, we need to coalesce a set of confluent processes into a branching structure. Coalescing operation merges the shared part of evolution histories, and in so doing, makes the 'branching points' explicit.

- Given a set $\mathcal{F}$ of confluent processes of G, we use $pr(\mathcal{F})$ to denote the set of finite prefix of $\mathcal{F}$. Then we can construct a *general transition system* G'[3], called the *coalescing* of $\mathcal{F}$, s.t. $S_{G'} = pr(\mathcal{F})$, $\hat{s}_{G'} = \hat{G}$, and $F \xrightarrow{a}_{G'} F'$ iff $F \preceq_E F'$ and $\check{s}_F \xrightarrow{a}_{F'} \check{s}_{F'}$. It is crucial to note that, for all $F \in S_{G'}$, F/G' is isomorphic to F and, therefore, a confluent process of G.
- The notions of granular events can be extended onto G': $t_1 = F_1 \xrightarrow{a}_{G'} F_1'$ and $t_2 = F_2 \xrightarrow{a}_{G'} F_2'$ belong to a same granular event in G' iff t_1 belongs to T_1 in F_1'/G', t_2 belongs to T_2 in F_2'/G' and $min(T_1) = min(T_2)$.

Although we can coalesce arbitrary sets of confluent processes, it makes more sense to coalesce a set of confluent processes that are 1) mutually incomparable w.r.t. $\preceq$ and 2) able to *fully cover* the set of system evolutions. The second requirement can be formalised in the same spirit as for the definition of unfolding. A confluent process F covers a set of system executions, i.e. those which are a linearisation of some prefix of F, denoted $lin(pr(\{F\}))$. $\mathcal{F}$ fully covers the set of system evolutions if $\mathcal{L}(G) = lin(pr(\mathcal{F}))$. We call such set of confluent processes an *evolution cover* of G.

- An evolution cover $\mathcal{F}$ of G is an MCP evolution cover if all $F \in \mathcal{F}$ are MCPs.
- A transition system G' is a *CP unfolding* of G if there exists an evolution cover $\mathcal{F}$ of G s.t. G' is the coalescing of $\mathcal{F}$.
- A transition system G' is a *MCP unfolding* of G if there exists an MCP evolution cover $\mathcal{F}$ of G s.t. G' is the coalescing of $\mathcal{F}$.
- If, furthermore, for all $L \in \mathcal{L}(G)$, there exists a unique $F \in pr(\mathcal{F})$ s.t. $L \in lin(F)$, we call $\mathcal{F}$ *determinate.*

For determinate evolution covers, we can give a simplified (alternative) definition to CP unfolding:

- We say an acyclic TS G is a *confluent tree* if, for all $s \in S_G$, s/G is confluent (denoting a concurrent run).
- We say a confluent tree G' is a *confluent tree unfolding* of TS G if G' is an unfolding of G.

Lemma 10. *G' is a confluent tree unfolding of G iff there is a determinate evolution cover $\mathcal{F}$ s.t. G' is the coalescing of $\mathcal{F}$.*

The notion of *events* on top of confluent tree unfoldings is exactly the same as that on top of confluent processes, i.e. granular events quotiented by an equivalence which is the transitive closure of or-causal coupling relation.

However, the above definition is not extendable to general CP unfolding because of indeterminate evolution cover. For instance, when the two confluent processes in the above figure are coalesced into G', transitions $s_0 \xrightarrow{b} s_1$ and $s_0' \xrightarrow{b} s_1'$ will collapse in G' into one transition, and so are states s_2 and s_2' into

[3] The reason general transition systems are needed here is largely due to indeterminate evolution covers (introduced below).

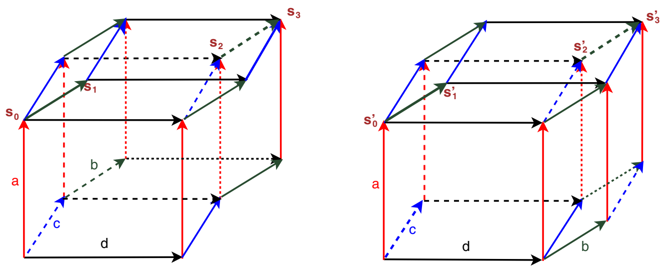


Fig. 4. Label ambiguity v.s. event ambiguity

one state, say $g_2 \in S_{G'}$. (Note that states s_3 and s_3' will not collapse in G'; they are mapped resp. to say $g_3, g_3' \in S_{G'}$.) Since $s_0 \xrightarrow{b} s_1$ and $s_2 \xrightarrow{b} s_3$ belong to the same granular event in the left confluent process and $s_0' \xrightarrow{b} s_1'$ and $s_2' \xrightarrow{b} s_3'$ to the same granular event in the right, using the above definition we can see that $s_2 \xrightarrow{b} s_3$ and $s_2' \xrightarrow{b} s_3'$ belong to the same event in G'. Thus, from state g_2 by firing the same event we can reach two different states, i.e. g_3 and g_3'. This is contradictory with the intuition of events. In summary, confluent tree unfolding supports the notion of events, whereas CP unfolding only supports the notion of granular events.

So far our problem statement and foundation work are developed mostly within the interleaving framework. But we have witnessed the usefulness of 'event intuition' in understanding notions like prefix, postpone and back-propogation for confluent processes. As we start to deal with more sophisticated *CP branching processes*, however, we will see that it is crucial (due to simplicity and intuitiveness) to reason directly in terms of events, concurrency, causality and conflict rather than in terms of transitions, commutativity, enabling and disabling.

Thus, we will move gradually into event-based models, e.g. configuration structures and granular configuration structure. Configuration structures are the non-interleaving incarnation of confluent tree unfolding and is thus built from events; while granular configuration structure is the non-interleaving incarnation of CP unfolding and is built from granular events.

Below we start with a quick introduction to the two structures, focusing on the correspondence with their transition system incarnations. Granular configuration structure will be the basis of our MCP unfolding (that requires indeterminate evolution covers, c.f. the example in Figure 1). We will present a formal and detailed introduction of granular configuration structure in the next section[4].

A configuration structure (E, C) consists of a set E of events and a set C of configurations. Each configuration $c \in C$ is a subset of events, which represents the global state reached after firing exactly the set c of events. The empty configuration represents the initial state.

For example, the left graph in the Figure below is a confluent tree G. We can coarsely partition transitions in G into events as shown in the middle graph, or

[4] A formal and detailed introduction of configuration structures can be found in [8].

we can finely group them into granular events, which do not form a partitioning of transitions (e.g. $e3$ and $e3'$ share a transition), as shown in the right graph. In the middle graph each state in G is mapped to a configuration. Given a state s mapped to c, if s can transit to s' via a transition belonging to e, then the s' is mapped to $c \cup \{e\}$. The soundness of this rule is implied by the fact that, no matter what system execution one uses to reach a given state in F, the set of events fired by the execution is the same.

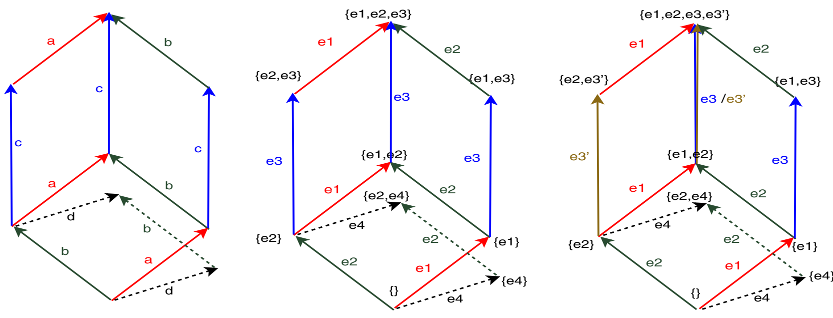


Fig. 5. From transitions to events and granular events

Similarly in the right graph each state s in G is mapped to a granular configuration c. But the property that all system executions to a same state fire the same set of events is no longer true. c here denotes, instead, the set of granular events whose member (i.e. transition) has occured in s/G. Thus it is possible that, by firing one transition in G, we can fire more than one event in the granular configuration structure, e.g. from $\{e1, e2\}$ to $\{e1, e2, e3, e3'\}$ by c and from $\{e1, e3\}$ to $\{e1, e2, e3, e3'\}$ by b in the right graph.

Discussion: The notion of confluent tree unfolding can be of independent interests. Indeed it provides a powerful tool for analysing previous works such as [18,12] and [8].

In [18,12] the class of confluent processes one can produce by unfolding a TS G is highly constrained. It is because an independence relation is imposed on top of G, i.e. the so called *transition systems with independence* (TSI). The independence relation marks (statically) a selected subset of diamonds in G as 'true diamonds', and requires all diamonds in confluent processes originating from true diamonds. Furthermore, Axiom 3 of TSI requires that no true diamond can be unfolded sequential, i.e. if two consecutive edges of the diamond are unfolded in F, then the whole diamond is unfolded in F.

The work in [8] removes the static independence restriction on transitions. Thus a confluent process can utilise any possible diamond in G, and a diamond can be unfolded sequentially in one confluent process while concurrently in another one (c.f. the example in Figure 10 of [8]).

Furthermore, Axiom 4 of TSI imposes a transitivity-like condition on the set of true diamonds so that they form a global network of diamonds and the existence of a true diamond at one location implies the existence of a set of true diamonds at its neighboring locations. Therefore, Axiom 4 combined with Axiom 3 ensures that 1) or-causality does not occur in confluent processes (and thus granular events coincide with events), and 2) non-local conditions become reducible to local ones (since the non-local part is guaranteed by the transitivity). One example is, given a confluent process F of a TSI G and an event T in F, T is postponed at a state $s \in S_F$ that is adjacent to $o(T)$ via transition $s \xrightarrow{a} o(T)$ iff $a \diamond_{f(s)} lb(T)$ in G.

Other related works that transform transition systems into non-interleaving models include the region theory of Petri net [1]. However, it is beyond the scope of this paper for detailed comparison.

5 Unfolding Procedure

In this section we first introduce *granular configuration structures* which is an adaptation of labelled configuration structures [15,14,8]. Granular configuration structures 1) restore the causality relation on events which can greatly simplify the definition of advanced notions like prefixes, immediate conflicts, etc. and 2) improve the expressiveness so that CP/MCP unfoldings can be fully captured. Then we give the *MCP unfolding* procedure to unfold transition systems into granular configuration structures.

5.1 Granular Configuration Structures

Definition 2. *A* granular configuration structure *(or simply GCS) over alphabet Σ is a triple $(E^{\leq}, C, lb)$, where*

– *E is a partially ordered set of* granular events *(or henceforth simply events), where $\leq$ is the well-founded* causality *relation,*
– *lb is a* labelling function *mapping events of E into labels of Σ,*
– *and C is a set of* granular configurations *(or simply configurations), where each configuration $c \in C$ is a finite $\leq$-downward closed subset of E and $e \in E$ implies $[e] \in C$.*

A configuration c can be thought of as representing a state reached after the execution of exactly the set c of granular events. The empty configuration $\{\}$ represents the initial state and is a required member of C in our model.

Below we fix a GCS, $cs = (E^{\leq}, C, lb)$, and introduce some basic notions for GCSes.

– We say cs is *finitely branching* if the Hasse diagram of $E^{\leq}$ is finitely branching. In such a GCS, concurrency and conflict are bounded and *infinite configurations* are derivable from finite ones.
– $[e]$ denotes the $\leq$-downward closure of $\{e\}$ while $[e]^-$ denotes $[e] \setminus \{e\}$.
– Given $X \subseteq E$, $[X]$ and $[X]^-$ denote $\bigcup_{e \in X}[e]$ and $\bigcup_{e \in X}[e]^-$ resp.
– We say a nonempty finite subset $\delta \subseteq E$ is a *consistent set* if there is some $c \in C$ such that $\delta \subseteq c$. Otherwise, δ is an *inconsistent set.*

- A consistent $\leq$-downward closed $\delta \subseteq E$ is called a *pre-configuration.*
- We say an event e is *activated* at pre-configuration δ, or $e \in ac(\delta)$, if $[e]^- \subseteq \delta$ and $c \uplus \{e\}$ is consistent.
- We say *cs* is *well-activated* if $lb(e) = lb(e') \wedge e \,||_{\leq}\, e' \wedge \{e, e'\}$ is consistent implies $[e]^- \,||_{\subseteq}\, [e']^-$.
- We say an activated event e at pre-configuration δ is *pending* at δ, or $e \in ac_{pn}(\delta)$, if $\forall e' \in \delta : e \,||_{\leq}\, e' \implies lb(e) \neq lb(e')$.
- We say an activated event e is *p-pending* at pre-configuration δ, or $e \in ac_{pp}(\delta)$, if $\exists e' \in \delta : e \,||_{\leq}\, e' \wedge lb(e) = lb(e')$. Given a pre-configuration δ, we say δ is *p-pending closed* if $ac_{pp}(\delta) = \{\}$; otherwise we use δ^* to denote the *p-pending closure* of δ.
- We say there is a *transition* from c to c', written $c \xrightarrow{a}_C c'$, if $c \subset c'$ and there exists an event $e \in ac_{pn}(c)$ s.t. $lb(e) = a$ and $\{e\} \subseteq c' \setminus c \subseteq (c \uplus \{e\})^* \setminus c^*$.

The definition of transitions here is unconventional, esp. in comparison with configuration structures. A transition from c to c' may involve multiple events $(c' \setminus c)$. Some of them, those pending events from $ac_{pn}(c)$, are the 'driving events' of the transition while others, those p-pending events from $(c \uplus \{e\})^* \setminus (c^* \cup \{e\})$, are 'auxilary ones' piggybacked on the transition. Note that only those freshed generated p-pending events (due to the driving ones) can be piggybacked, not any old one from $c^* \setminus c$.

Thus a GCS gives rise to an acyclic transition system, and the definitions like 'subsequent to' relation $\sqsubseteq$, (system) execution, etc. carry over. Furthermore, note that $c \subset c'$ does not imply there exists an execution from c to c' in GCSes; this is very different from configuration structures.

- We write $eb(c) = \{a \in \Sigma \mid c \xrightarrow{a}_C\}$ to denote the set of enabled actions at c. $succ(c) = \{c' \in C \mid c \xrightarrow{a}_C c'\}$ denotes its set of successor configurations.
- We say *cs* is *well-connected* if all the configurations in C are $\rightarrow_C$-reachable from $\{\}$ and, for all $c \in C$ and $e \in ac_{pn}(c)$, there exists $c' \in C$ s.t. $c \xrightarrow{lb(e)}_C c' \wedge e \in c'$.

Based on the above, we can say *cs* is *well-formed* if it is finitely branching, well-activated and well-connected. Well-formed GCSes have roughly the same expressiveness as (general) event structures from [17]. We prefer to use GCSes in this paper mainly because of its affinity to transition systems. We give a few basic properties of well-formed GCSes, esp. those in comparison with configuration structures.

- Given a finite $D \subseteq C$, we say D is *downward-closed* (w.r.t. $\sqsubseteq$) if $c \sqsubseteq c' \in D \implies c \in D$. We use $D^{\downarrow}$ to denote the $\sqsubseteq$-downward closure of D.
- We say a finite subset of configurations $D \subseteq C$ are *compatible* if $\bigcup D$ is consistent and disjoint from $\bigcup_{c \in D}(c^* \setminus c)$.
- We say *cs* is *closed under bounded union* if, for all compatible subsets $D \subseteq C$, $\exists c' \in C : \bigcup D \subseteq c' \subseteq (\bigcup D)^* \setminus \bigcup_{c \in D}(c^* \setminus c)$.
- We say *cs* is *free of auto-concurrency* if $lb(e) = lb(e')$ and $e \,||_{\leq}\, e'$ implies $\nexists c \in C : [e']^- \cup [e] \subseteq c \wedge e' \in ac(c)$. [5]

[5] GCSes with auto-concurrency can be useful, on the other hand, for unfolding systems like general Petri Net.

Lemma 11. *cs is free of auto-concurrency and closed under bounded union.*

Lemma 12. *If there is a non-empty execution from c to c' such that $e \in ac(c)$ and $c' \cup \{e\}$ is consistent, then we have either $e \in c'$ or $e \in ac(c')$.*

GCSes are the non-interleaving incarnations of the coalescing of confluent processes: each configuration in a GCS uniquely corresponds to an acyclic confluent transition system.

Lemma 13. *Given any $c \in C$, $cs \rhd \{c\}^{\downarrow}$, i.e. the restriction of cs to $\{c\}^{\downarrow}$, gives rise to an acyclic confluent transition system, i.e. $CP(c) = (\{c\}^{\downarrow}, \Sigma, \rightarrow_{\{c\}^{\downarrow}}, \{\})$, where $CP()$ is an injective function.*

For the rest of this paper we only consider well-formed GCSes and simply call them GCSes. Advanced notions of GCSes can be easily defined by using the restored causality relation:

- Given a finite $\leq$-downward closed subset $X \subseteq E$, we say X is *p-pending event closed* (or simply pp-event closed) if, for all pre-configuration $\delta \subseteq X$ and event $e \in X$, $e \in ac_{pn}(\delta)$ implies $(\delta \cup \{e\})^* \subseteq X$. We denote the *pp-event closure* of X as $X^{\star}$.
- A finite subset $X \subseteq E$ is a *prefix* if X is both $\leq$-downward closed and pp-event closed.
- A finite $\leq$-antichain $K \subseteq E$ is an *immediate conflict* (IC) if $[K]^{\star}$ is a minimal prefix that is not conflict-free.

Lemma 14. *K is an IC implies $K \subseteq ac_{pn}([K]^{-})$.*

Based on these notions, we can recover a purely event-based definition of GCS-like structures (i.e. without resorting to the use of configurations):

Granular Event Structures (GESes): A granular configuration structure (say cs) can be re-formulated (say using transformation $\mathcal{CE}(cs)$) into a *granular event structure*: a granular event structure is a triple, $es = (E^{\leq}, IC, lb)$, where IC is a set of immediate conflicts (IC). An immediate conflict $K \in IC$ is a finite $\leq$-antichain of events satisfying that, for all $K \in IC$, $[K]$ contains no IC other than K and, for all $e \in E$, $[e]$ contains no IC.

Conversely, we can also recover a granular configuration structure from es (say using transformation $\mathcal{EC}(es)$). Given cs, we say a finite subset $X \subseteq E$ is *consistent* if $[X]$ contains no IC. This enables us to recover the definition of *pre-configuration*, *activated/pending/p-pending* events and *well-activatedness*. On well-activated es, we say a pre-configuration $\delta \subseteq E$ is a *configuration* if there exists another pre-configuration $\delta' \supseteq \delta$ s.t. $(ac(\delta') \cup \delta') \cap ac_{pp}(\delta) = \{\}$. Finally we say es is *well-formed* if it is well-activated, finite-branching and satisfying $e \in E \implies [e]$ is a configuration.

We can show that well-formed granular event structures correspond exactly to well-formed granular configuration structures:

Theorem 2. *$cs = \mathcal{EC}(\mathcal{CE}(cs))$ and $es = \mathcal{CE}(\mathcal{EC}(es))$.*

5.2 Unfolding TSes into GCSes

The aim of our procedure is to construct the Hasse diagram of configurations in a roughly bottom up fashion. Starting from the empty configuration, we move up step by step, deriving larger configurations from smaller ones. Each configuration generated corresponds to a finite MCP prefix (up to isomorphism).

However, since transitions between configurations follow 'big-step semantics' (i.e. firing multiple events), a simpler and more elegant approach is to first build the Hasse diagram of pre-configurations, where the transitions follow 'small-step semantics'. Each pre-configuration corresponds to a finite PCP prefix. Then, we remove all pre-configurations that are not configurations (called *nonstable pre-configurations*) and re-connect what are left, i.e. configurations, by big steps.

The search for new PCP prefixes is guided by a key sub-procedure of extending one PCP (say F) into another PCP (say F'). Firstly, we calculate the maximal back-propagation of actions for F and, if there is no corresponding events existing for those points, we create new ones accordingly. Then we extend F by those events to generate F' (i.e. elongate or relax F depending on the events being pending or p-pending), which is a PCP according to Theorem 1. Note that in generating F' there is a 'prefix-closure' effect. That is, F' might have more than one immediate prefix and some of them (other than F) might be non-PCP and, therefore, not generated yet. Thus in generating F' we also need to generate some of its prefixes.

Now let us formalise the unfolding procedure and define the notions of *MCP unfolding* and *MCP branching processes*:

- Given a TS G, a *labelled GCS* over G is a tuple $lcs = (E^{\leq}, C, lb, st)$, where $(E^{\leq}, C, lb)$ constitutes a GCS over Σ and $st : C \to S_G$ is a function mapping configurations to states.
- Given lcs over G, we say lcs is an *MCP unfolding* of G if $(C, \Sigma, \to_C, \{\})$ is an MCP unfolding of G via homomorphism $f = st$.
- Given lcs over G, we say $lcs = (E^{\leq}, C, lb, st)$ is an *MCP branching process* of G if there exists an *MCP unfolding* $lcs' = (E'^{\leq}, C', lb', st')$ of G s.t. E is a prefix of $(E'^{\leq}, C', lb')$, $C = C' \triangleright E$, $lb = lb' \triangleright E$ and $st = st' \triangleright C$.

Our procedure, $lcs = \mathcal{U}(G)$ (c.f. Figure 6), unfolds TSes in a maximally concurrent fashion into a labelled GCS over G.

The basic data structure is $(E, preC, lb, st)$. E and $preC$ store resp. the set of generated events and the set of generated pre-configurations. The back-propagation information for each pre-configuration is stored in function bp_{pn} and bp_{pp} resp. We create new events based on such information, which are then passed on from smaller pre-configurations to larger ones and stored in function ac_{pn} and ac_{pp} resp. as activated events.

Then, adding activated events to existing pre-configurations produces the set of potential *extensions*, EXT. Some of the extensions are PCPs, i.e. those in the set PCP. The definition of PCP is recursive, utilising Theorem 1 and starting from the empty pre-configuration. But, due to the 'prefix closure' effect mentioned above, PCP extensions cannot be *realised* 'eagerly' by immediately throwing them into *preC*. Rather, the realisation proceeds in steps by first realising the prefixes (i.e. sub-configuration[6]) of the PCP extensions, so that the expansion of *preC* will preserve the $\preceq$-downward closedness. We say a PCP prefix is *ready for realisation* if it is unrealised but all its sub-configurations are realised. The set of ready-for-realisation PCP prefixes is given in NXT.

Thus, starting from an empty *preC* we pick one pre-configuration (say c') a time from NXT and realise by adding c' to *preC* (line 3-5 of function UNFOLD in Figure 6). In the mean time we calculate (c.f. line 8-9 of function REALISE) the set of activated events inheritable by c' from its immediate sub-configurations (i.e. ${}^\bullet c'$), and also derive (line 4-7 of function REALISE) the new bp_{pn} and bp_{pp} functions based on those of ${}^\bullet c'$. Some useful notations used in such calculation/derivation are given below.

Given any $c, c' = c \uplus \{e\} \in preC$, we define

- $prp_{pn}(c', c, a) =$
  - $\{[e]\}$ if $e \in ac_{pn}(c) \wedge \neg lb(e) \diamond_{st(c)} a \vee e \in ac_{pp}(c) \wedge mbp_{pn}(c, a) = \{c\}$, or
  - $min(mbp_{pn}(c, a) \cup \{[e]\})$ if otherwise;
- $prp_{pp}(c', c, a) =$
  - $bp_{pp}(c, a)$ if $lb(e) \neq a$,
  - $\{c_0 \in bp_{pp}(c, a) \mid c_0 \,||_{\sqsubseteq}\, [e]^-\}$ if $e \in ac_{pp}(c) \wedge lb(e) = a$, or
  - $\{c_0 \in bp_{pn}(c, a) \cup bp_{pp}(c, a) \mid c_0 \,||_{\sqsubseteq}\, [e]^-\}$ if $e \in ac_{pn}(c) \wedge lb(e) = a$;
- $hrt_{pn}(c', c) = \{e' \in ac_{pn}(c) \mid e \in ac_{pp}(c) \vee (e \in ac_{pn}(c) \wedge lb(e) \diamond_{st(c)} lb(e'))\}$
- $hrt_{pp}(c', c) = \{e' \in (ac_{pn}(c) \cup ac_{pp}(c)) \mid (e' \in ac_{pn}(c) \implies e \in ac_{pn}(c) \wedge lb(e) = lb(e')) \wedge (lb(e) = lb(e') \implies [e']^- \,||_{\sqsubseteq}\, [e]^-)\}$.

Note that $hrt_{pn}(c', c)$ produces the set of pending events c' can inherit from c while $hrt_{pp}(c', c)$ produces the set of p-pending events c' can inherit from c. On the other hand, $prp_{pn}(c', c, a)$ produces the information about how action a can be back-propagated through edge (c, c') into the sub-configurations while $prp_{pp}(c', c, a)$ about the p-pending points of a through edge (c, c').

Then, if c' is a PCP and has maximal back-propagation not covered by activated events inherited from ${}^\bullet c'$, we create new ones to cover them (c.f. procedure GENEVENT). An event (say e) can be created only if its origin (say c) is a *configuration* according to c': the condition is implemented as the predicate c ISCFGIN c'. After e is created at c, we propagate e upward to its super-configurations (c.f. line 4-8 of procedure GENEVENT).

Finally, after the set of pre-configurations fully generated, we filter out non-stable pre-configurations and produce the set of configurations (c.f. line 6 of function UNFOLD). The intuition is that a PCP pre-configuration is a MRP if $ac_{pp}(c') = \{\}$ and the prefixes of MRP pre-configurations are configurations.

We can illustrate the procedure by unfolding the broken cube originally from Figure 1. In the step 0 of Figure 7 the initial state is mapped to configuration $\{\}$. The set of activated events at $\{\}$, i.e. $ac_{pn}(\{\})$ and $ac_{pp}(\{\})$, is initialised

[6] Actually it should be sub-pre-configurations, i.e. pre-configurations which are subset of the original pre-configurations.

```
Data Structure:
    $(E, preC, lb, st) = (\{\}, \{\{\}\}, \{\}, \{(\{\}, \hat{s})\})$;
    $ac_{pn} = \{\}$; $ac_{pp} = \{\}$; $bp_{pn} = \{\}$; $bp_{pp} = \{\}$;

Derived Values, Functions and Predicates:
$\textsc{ext} = \{c \cup \{e\} \mid c \in preC \wedge e \in ac_{pn}(c) \cup ac_{pp}(c)\}$
$\textsc{pcp} = \{c \cup \{e\} \mid c \in \textsc{pcp} \wedge (\, e \in ac_{pn}(c) \wedge [e]^- \in mbp_{pn}(c, lb(e))$
                                        $\vee e \in ac_{pp}(c) \wedge [e]^- \in mbp_{pp}(c, lb(e))\,)\}$
$\textsc{nxt} = \{c' \in \textsc{ext} \setminus preC \mid \exists c'' \in \textsc{pcp} : c' \subseteq c'' \wedge \forall c \in \textsc{ext} : c \subset c' \implies c \in preC\}$
$c$ isCFGin $c' = \forall e \in c' \setminus c : [e]^- \notin bp_{pp}(c, lb(e)) \wedge \forall a \in \Sigma : bp_{pp}(c, a) \cap bp_{pp}(c', a) = \{\}$
$^\bullet c' = \{c \mid c \in preC \wedge c \uplus \{e\} = c'\}$

Function Unfold($(S, \Sigma, \Delta, \hat{s})$)
Begin
 1  Foreach $a \in eb(\hat{s})$ do Set $bp_{pn}(\{\}, a) = \{\{\}\}$;
 2  GenEvent($\{\}$);
 3  While $\textsc{nxt} \neq \{\}$
 4      Pick any $c' \in \textsc{nxt}$ and Realise($c'$);
 5      If $c' \in \textsc{pcp}$ Then GenEvent($c'$);
 6  Set $C = \{c \in preC \mid \exists c' \in \textsc{pcp} : c \subseteq c' \wedge ac_{pp}(c) \cap c' = \{\} = ac_{pp}(c')\}$;
 7  Return $(E, C, lb, st)$
End

Procedure Realise($c'$)
Begin
 1  Assume $c' = c \uplus \{e\}$ for some $c \in preC$;
 2  If $e \in ac_{pn}(c)$ Then $s' = \Delta(st(c), lb(e))$ Else $s' = st(c)$;
 3  Add $c'$ to $preC$ and $(c', s')$ to $st()$ ;
 4  Foreach $a \in eb(st(c'))$ do
 5      Set $bp_{pn}(c', a) = \bigcap_{c \in {}^\bullet c'} \{c_1 : preC \mid \exists c_0 \in prp_{pn}(c', c, a) : c_0 \subseteq c_1 \subseteq c'\}$;
 6  Foreach $a \in \Sigma$ s.t. $D = \bigcup_{c \in {}^\bullet c'} prp_{pp}(c', c, a) \neq \{\}$ do
 7      Set $bp_{pp}(c', a) = \{c_0 \in D \mid \forall c \in {}^\bullet c' : c_0 \subseteq c \implies c_0 \in prp_{pp}(c', c, a)\}$;
 8  Set $ac_{pn}(c') = \{e' \in \bigcup_{c \in {}^\bullet c'} hrt_{pn}(c', c) \mid \forall c \in {}^\bullet c' : [e']^- \subseteq c \implies e' \in hrt_{pn}(c', c)\}$;
 9  Set $ac_{pp}(c') = \{e' \in \bigcup_{c \in {}^\bullet c'} hrt_{pp}(c', c) \mid \forall c \in {}^\bullet c' : [e']^- \subseteq c \implies e' \in hrt_{pp}(c', c)\}$
End

Procedure GenEvent($c'$)
Begin
 1  Foreach $(a, c) \in mbp_{pp}(c') \cup mbp_{pn}(c')$
             s.t. $\nexists e \in E : (lb(e), [e]^-) = (a, c) \wedge c$ isCFGin $c'$ do
 2      Create a new event $e$, and add $e$ to $E$ and $(e, a)$ to $lb()$;
 3      Add $e$ to $ac_{pn}(c)$ and set $D = \{c\}$;
 4      While $M = min(\{x \in preC \mid x \supset c\} \setminus D) \neq \{\}$ do
 5          Pick any $c'' \in M$ and add $c''$ to $D$;
 6          If $\forall c \in {}^\bullet c'' : [e]^- \subseteq c \implies e \in hrt_{pn}(c'', c)$ Then Add $e$ to $ac_{pn}(c'')$
 7          Else If $\forall c \in {}^\bullet c'' : [e]^- \subseteq c \implies e \in hrt_{pp}(c'', c)$
 8                Then Add $e$ to $ac_{pp}(c'')$ Else Add $\{c \in preC \mid c \supseteq c''\}$ to $D$
End
```

Fig. 6. An unfolding procedure

to be empty (i.e. no inheritance). Since the three enabled actions at the initial state are maximally back-propagated to $\{\}$ but there is no activated events at $\{\}$ to match them, we create three new events $e1$, $e2$ and $e3$ at $\{\}$ (note the use of symbol !) and add them to $ac_{pn}(\{\})$. Thus the initial state is labelled as $\{\}/\{e1, e2, e3\}$ (in the style of $c/ac_{pn}(c)$). Firing one of the generated events say $e1$ leads to a new extension, say $\{e1\}$, which is a new member of NXT.

In the step 1, we pick a member say $\{e1\}$ of NXT and realise it. $\{e1\}$ is mapped to s_1 and $ac_{pn}(\{e1\})$ inherits $\{e2, e3\}$ from $\{\}$, which fully covers the maximal back-propagation of the two actions enabled at s_1. Thus, although $\{e1\}$ is a PCP and procedure GenEvent is called, no new event will be created. Similarly we can also realise $\{e2\}$ and $\{e3\}$.

Now $\{e1, e3\}$ is a member of NXT, which we can pick in the step 2 to realise. Note that $\{e1, e3\}$ can inherit $e2$ from $\{e1\}$ but not from $\{e3\}$ (since $e2$ and $e1$ do not form a diamond at $\{e3\}$). This inconsistency leads to $e2$ not being added to $ac_{pn}(\{e1, e3\})$. $\{e1, e3\}$ is a PCP and in calling GenEvent, however, a new event $e2'$ is created at $\{e1\}$ to cover the maximal back-propagation of the enabled b action at s_5 (mapped to $\{e1, e3\}$). $e2'$ can be inherited and added to $ac_{pn}(\{e1, e3\})$. Similarly, the back-propagation from $\{e1, e2\}$ and from $\{e2, e3\}$ leads to the creation of $e3'$ at $\{e1\}$ and $e1'$ at $\{e2, e3\}$ resp.

$e1'$ at $\{e2, e3\}$ can lead to a PCP extension but not so for $e2'$ and $e3'$ at $\{e1\}$. Instead, they are PCP prefixes since $\{e1, e2, e3'\}$ and $\{e1, e2', e3\}$ are PCP extensions. Therefore, all the possible extensions so far are inside NXT. In realising these extensions, $\{e1, e3'\}$ and $\{e1, e2'\}$ need to be realised before $\{e1, e2, e3'\}$ and $\{e1, e2', e3\}$ resp. to preserve the $\preceq$-downward closedness of $preC$.

In the step 3, obviously $\{e1, e2'\}$ inherits $e3$ and $e3'$ from $\{e1\}$; and $\{e1, e3'\}$ inherits $e2$ and $e2'$. But $e3'$ at $\{e1, e2'\}$ and $e2'$ at $\{e1, e3'\}$ will lead to extensions outside NXT. So the outcome is a GCS with three maximal configurations. Two of them, $\{e1, e3, e2'\}$ and $\{e1, e2, e3'\}$, are mapped to a same terminating state $s7$. (In contrast, state $s5$ is split into $\{e1, e3\}$ and $\{e1, e3'\}$ while $s6$ into $\{e1, e2\}$ and $\{e1, e2'\}$.)

The broken cube example does not have or-causality; thus the part of the procedure related to non-stable pre-configuration and p-pending points/events has not been utilised. We give a second example in Figure 8 to do so.

The original transition system is given as the top-left graph in the figure. The top-right graph is its MCP unfolding, which is the coalescing of two MCPs ($\{a1, b1, c1, c2, d2\}$ and $\{a1, b1, c2, d1, d2\}$). The MCP of $\{a1, b1, c1, c2, d2\}$ is drawn with thick-line edges and strong-colored configurations.

The bottom graph is the Hasse diagram of pre-configurations (with some edges missing). The faint-colored pre-configurations in bottom graph are nonstable pre-configurations; they will be removed in order to produce the Hasse diagram of configurations. Note further that $\{a1, b1, c2\}$ is a configuration in the MCP of $\{a1, b1, c1, c2, d2\}$ but not so in that of $\{a1, b1, c2, d1, d2\}$, even though it is a pre-configuration being used in the generation of both MCPs. Note that the set of pre-configurations being used to generate the MCP of $\{a1, b1, c1, c2, d2\}$ are those connected up by edges in the graph.

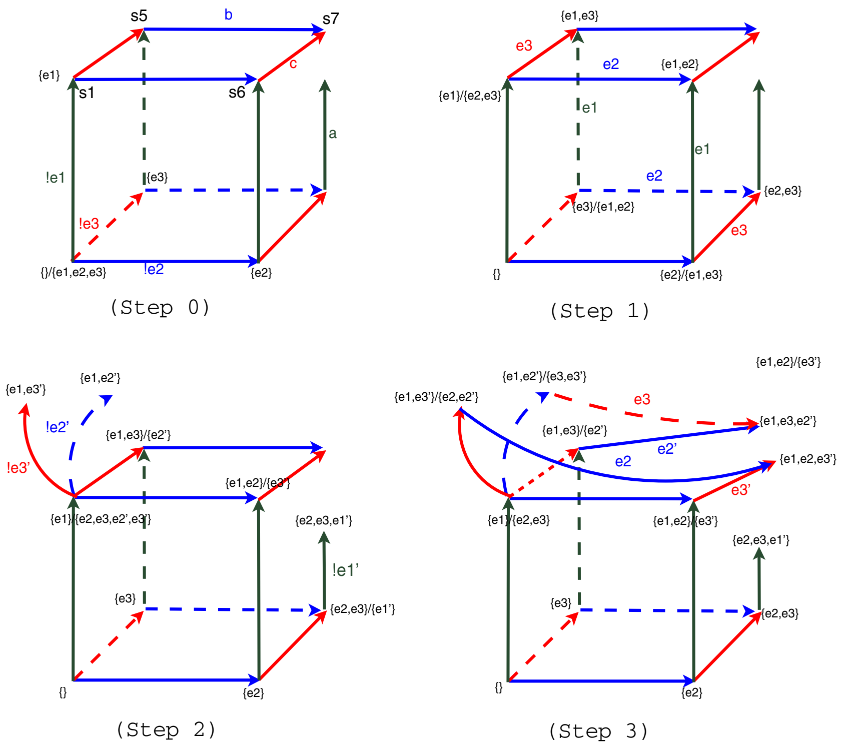


Fig. 7. Unfolding of the broken cube

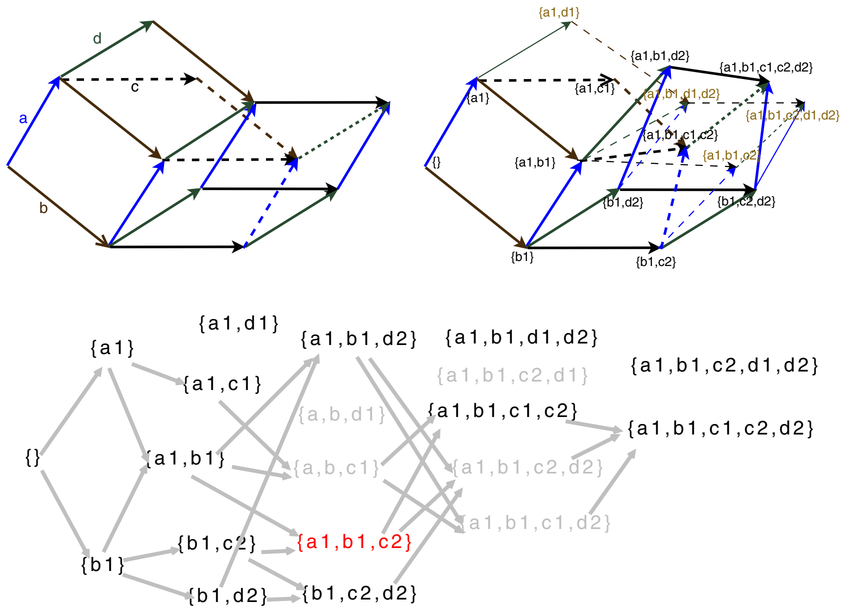


Fig. 8. P-pending event and nonstable pre-configuration

We can show that the output of function UNFOLD indeed is the MCP unfolding of G:

Theorem 3. *$\mathcal{U}(G)$ is an MCP unfolding of G.*

Acknowledgements. We thank Xiaoquan Wu for implementing a prototype unfolder based on the algorithm of this paper. Bill Roscoe and Henri Hansen commented on the earlier ideas of the paper. We acknowledge support from the Macau FDCT project 041/2007/A3 and EU FP7 project IST 231167.

References

1. Badouel, E., Bernardinello, L., Darondeau, P.: The synthesis problem for elementary net systems is np-complete. Theor. Comput. Sci. 186(1-2), 107–134 (1997)
2. Best, E., Devillers, R.R.: Sequential and concurrent behaviour in petri net theory. Theor. Comput. Sci. 55(1), 87–136 (1987)
3. Best, E., Fernández, C.: Nonsequential processes: a Petri net view. EATCS Monographs on TCS, vol. 13. Springer (1988)
4. Godefroid, P.: Partial-order Methods for the Verification of Concurrent Systems: an Approach to the State-explosion Problem. LNCS, vol. 1032. Springer, Heidelberg (1996)
5. Goltz, U., Reisig, W.: The non-sequential behavior of petri nets. Information and Control 57(2/3), 125–147 (1983)
6. Groote, J.F., Sellink, M.P.A.: Confluence for process verification. Theoretical Computer Science 170(1-2), 47–81 (1996)
7. Gunawardena, J.: Causal automata. TCS 101(2), 265–288 (1992)
8. Hansen, H., Wang, X.: On the Origin of Events: Branching Cells as Stubborn Sets. In: Kristensen, L.M., Petrucci, L. (eds.) PETRI NETS 2011. LNCS, vol. 6709, pp. 248–267. Springer, Heidelberg (2011)
9. Liu, X., Walker, D.: Partial confluence of proceses and systems of objects. TCS 206(1-2), 127–162 (1998)
10. Milner, R.: Concurrency and Communication. Prentice-Hall (1988)
11. Peled, D.A.: All From One, One For All: On Model Checking Using Representatives. In: Courcoubetis, C. (ed.) CAV 1993. LNCS, vol. 697, pp. 409–423. Springer, Heidelberg (1993)
12. Sassone, V., Nielsen, M., Winskel, G.: Models for concurrency: Towards a classification. Theor. Comput. Sci. 170(1-2), 297–348 (1996)
13. Valmari, A.: Stubborn Sets for Reduced State Space Generation. In: Rozenberg, G. (ed.) APN 1990. LNCS, vol. 483, pp. 491–515. Springer, Heidelberg (1991)
14. van Glabbeek, R.J., Plotkin, G.D.: Configuration structures, event structures and petri nets. Theoretical Computer Science 410(41), 4111–4159 (2009)
15. van Glabbeek, R., Goltz, U.: Refinement of actions and equivalence notions for concurrent systems. Acta Informatica 37(4), 229–327 (2001)
16. van Glabbeek, R.J., Plotkin, G.D.: Event Structures for Resolvable Conflict. In: Fiala, J., Koubek, V., Kratochvíl, J. (eds.) MFCS 2004. LNCS, vol. 3153, pp. 550–561. Springer, Heidelberg (2004)
17. Winskel, G.: Event Structures. In: Brauer, W., Reisig, W., Rozenberg, G. (eds.) APN 1986. LNCS, vol. 255, pp. 325–392. Springer, Heidelberg (1987)
18. Winskel, G., Nielsen, M.: Models for concurrency. In: Handbook of Logic in Computer Science, vol. 4. Clarendon Press (1995)
19. Yakovlev, A., Kishinevsky, M., Kondratyev, A., Lavagno, L., Pietkiewicz-Koutny, M.: On the models for asynchronous circuit behaviour with or causality. FMSD 9(3), 189–233 (1996)

Old and New Algorithms for Minimal Coverability Sets

Antti Valmari and Henri Hansen

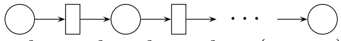
Department of Software Systems, Tampere University of Technology
P.O. Box 553, FI-33101 Tampere, Finland
{antti.valmari,henri.hansen}@tut.fi

Abstract. Many algorithms for computing minimal coverability sets for Petri nets *prune futures*. That is, if a new marking strictly covers an old one, then not just the old marking but also some subset of its subsequent markings is discarded from search. In this publication, a simpler algorithm that lacks future pruning is presented and proven correct. Then its performance is compared with future pruning. It is demonstrated, using examples, that neither approach is systematically better than the other. However, the simple algorithm has some attractive features. It never needs to re-construct pruned parts of the minimal coverability set. If the minimal coverability set is constructed in depth-first or most tokens first order, and if so-called *history merging* is applied, then most of the advantage of future pruning is automatic. Some implementation aspects of minimal coverability set construction are also discussed.

1 Introduction

The set of reachable markings of a finite Petri net is not necessarily finite. However, Karp and Miller [4] showed that an abstracted version of it, the *coverability set*, is always finite. The coverability set constructed by the algorithm of Karp and Miller is not unique. Finkel defined a unique *minimal coverability set* and presented an algorithm for constructing it [1]. Surprisingly, more than a decade later an error was found in his algorithm [2]. This inspired new interest in coverability set algorithms.

Proposals for solving the problem correctly, and more efficiently than the original, have been made [3,6]. Some recent work also exists on incremental construction of coverability graphs, that is, mapping transformations of a Petri net to transformations of its coverability graph [5].

Minimal coverability sets may be huge. For instance, the "linear" Petri net  with n places, $n-1$ tokens in the first place, and no tokens elsewhere, has $(2n-2)!/(n-1)!^2 \approx 2^{2n-2}/\sqrt{\pi(n-1)}$ maximal markings.

The algorithms of [4,1,6] are all based on the idea of building a tree of markings with acceleration or ω-addition, that is, replacing unbounded markings of a place with an ω, based on the history of a newly discovered marking. The two latter algorithms also *prune* the tree by excluding the already constructed descendants of the covered nodes from future exploration, in an attempt to make the computation more efficient. With this pruning, correctness or termination of the algorithm is jeopardised, and proving either becomes very hard, as is evidenced by the fact that the algorithm of [1] was long thought to be correct.

Pruning takes place when a sequence M_0, t_1, $\ldots$, t_n, M_n of markings and transitions has been found, where each M_i is obtained by firing t_i from M_{i-1} and then possibly adding ω-symbols, and then an M_0' such that $M_0 < M_0'$ is found. Then M_0 is clearly unnecessary in the coverability set. If $M_0 \; [t_1 \cdots t_n\rangle \; M_n$ and $M_0 < M_0'$, then there is an M_n' such that $M_0' \; [t_1 \cdots t_n\rangle \; M_n'$ and $M_n < M_n'$. Inspired by this, future pruning passivates also $M_1, \ldots, M_n$ simultaneously with M_0. However, when ω-symbols are added along the path from M_0 to M_n, it is possible that $M_n = M_n'$. Then it is possible that after M_n has been pruned, it may have to be activated anew. We say that the pruning of M_n is *overeager* if and only if $M_0' \; [t_1 \cdots t_n\rangle \; M_n$.

To provide a firm foundation for the discussion in this publication, in Section 2 we recall the basic facts of minimal coverability sets. Then, in Section 3, we propose an algorithm for creating minimal coverability sets that does not prune futures, and show that it is correct. We also comment on some implementation issues. Section 4 is devoted to a discussion of the order in which the set is constructed.

The simple approach is compared with future pruning in Section 5. We demonstrate that future pruning is vulnerable to having to construct large subsets more than once. Then we prove that if the minimal coverability set is constructed in depth-first order or what we call most tokens first order, and if what we call history merging is applied, then, even without explicit pruning, the algorithm automatically avoids the investigation of transitions from those markings that future pruning would prune but not in the overeager way. The last section presents our conclusions and some measurements.

2 Minimal Coverablity Sets

The basic facts of coverability sets are more or less widely known, but their published proofs tend to be unclear and tied to individual algorithms. Therefore, we prove them anew in this section, independently of even the notion of transition. The set of natural numbers (including 0) is denoted with $\mathbb{N}$. Let P be any finite set. Its elements are called *places.* We start with the notion of markings that may also use ω as the marking of a place. Intuitively, ω denotes unbounded.

Definition 1. *An* ω-marking *is a function from* P *to* $\mathbb{N} \cup \{\omega\}$. *If* $n \in \mathbb{N}$, *we define* $n < \omega$ *and* $\omega + n = \omega - n = \omega$. *Let* M *and* M' *be* ω*-markings. We define that* M' covers M *and write* $M \leq M'$ *if and only if* $M(p) \leq M'(p)$ *for every* $p \in P$. *Furthermore,* M' covers M strictly *if and only if* $M < M'$, *that is,* $M \leq M'$ *and* $M \neq M'$. *A (finite or infinite) sequence* $M_1, M_2, \ldots$ *of* ω*-markings is* growing *if and only if* $M_1 \leq M_2 \leq \ldots$ *and* strictly growing *if and only if* $M_1 < M_2 < \ldots$.

Note that (ordinary) markings are ω-markings. An ω-marking is a marking if and only if it does not contain ω-symbols.

Given any infinite growing sequence M_i of ω-markings and any place p, $M_i(p)$ either reaches a maximum value and stays there or grows without limit. The latter means that for each $n \in \mathbb{N}$, there is an i such that $M_i(p) \geq n$. Therefore, the following notion of the limit of the sequence is well-defined.

Definition 2. *Let M_1, M_2, $\ldots$ be ω-markings such that $M_1 \leq M_2 \leq \ldots$. Their* limit *is the ω-marking $M = \lim\limits_{i\to\infty} M_i$ such that for each $p \in P$, either $M(p) = M_i(p) = M_{i+1}(p) = M_{i+2}(p) = \ldots$ for some i, or $M(p) = \omega$ and $M_i(p)$ grows without limit as i grows.*

Clearly $M_i \leq \lim\limits_{i\to\infty} M_i$ for each i. The ω-markings in a sequence need not be different from each other. So also the sequence $M, M, M, \ldots$ has a limit. It is M. It is also worth pointing out that in this publication, the either-part of the definition does not require that $M(p) < \omega$. Therefore, $M(p) = \omega$ is possible in two ways: either $M_i(p) = \omega$ from some i on, or $M_i(p)$ grows without limit.

The following lemma is an immediate consequence of Definition 2. It says that given an arbitrary finite value, the places marked with ω in the limit may *simultaneously* get at least that value, while the other places get their limit values. In this publication, $M_i(p)$ may also be ω, but then trivially $M_i(p) \geq n$.

Lemma 1. *If $M = \lim\limits_{i\to\infty} M_i$, then for every $n \in \mathbb{N}$ there is an i such that for each $p \in P$, either $M_i(p) = M(p) < \omega$ or $M_i(p) \geq n$ and $M(p) = \omega$.*

For convenience, we also define a limit of a set of ω-markings as any limit of any infinite growing sequence of elements of the set. The limit of a set is not necessarily unique. Actually, each element of the set is its limit.

Definition 3. *Let $\mathcal{M}$ be a set of ω-markings. Then M is a* limit *of $\mathcal{M}$ if and only if there are $M_1 \in \mathcal{M}$, $M_2 \in \mathcal{M}$, $\ldots$ such that $M_1 \leq M_2 \leq \ldots$ and $M = \lim\limits_{i\to\infty} M_i$.*

The next lemma follows from Dickson's lemma, but is easier to prove directly.

Lemma 2. *Every infinite sequence of ω-markings has an infinite growing subsequence.*

Proof. Let M_i be the sequence and $P = \{p_1, p_2, \ldots, p_{|P|}\}$. We show that for each $0 \leq j \leq |P|$, M_i has an infinite subsequence $M_{j,i}$ such that $M_{j,1}(p_k) \leq M_{j,2}(p_k) \leq \ldots$ for $1 \leq k \leq j$.

Clearly M_i qualifies as the $M_{0,i}$. Let $j > 0$. If $M_{j-1,i}(p_j)$ only gets a finite number of different values for $i \in \mathbb{N}$, then some value v occurs infinitely many times. We let $M_{j,i}$ be the infinite subsequence obtained by picking those $M_{j-1,i}$ that have $M_{j-1,i}(p_j) = v$. Otherwise $M_{j-1,i}(p_j)$ gets infinitely many different values. Then $M_{j-1,i}$ has an infinite subsequence where $M_{j-1,i}(p_j)$ grows. It qualifies as the $M_{j,i}$.

Finally $M_{|P|,i}$ qualifies as the sequence in the claim of the lemma. $\square$

We are ready to define coverability sets. The idea is that for a given set $\mathcal{M}$ of markings, the ω-markings in its coverability set $\mathcal{M}'$ cover every marking in $\mathcal{M}$ without using bigger ω-markings than necessary. The goal is to get finite coverability sets. However, if $M(p)$ obtains infinitely many different values in $\mathcal{M}$, then to cover them all with finitely many ω-markings it is necessary to let $M'(p) = \omega$ in at least one $M' \in \mathcal{M}'$. More generally, it may be that many places must simultaneously have $M'(p) = \omega$ to cover some subset of $\mathcal{M}$ with finitely many ω-markings. Part 2 of the definition says that this is the only justification for the introduction of ω-symbols. The concept of antichain is important, because we will later show that a coverability set is minimal if and only if it is an antichain.

Definition 4. *Let $\mathcal{M}$ be a set of markings and $\mathcal{M}'$ a set of ω-markings. We define that $\mathcal{M}'$ is a* coverability set *for $\mathcal{M}$, if and only if*

1. *For every $M \in \mathcal{M}$, there is an $M' \in \mathcal{M}'$ such that $M \leq M'$.*
2. *Each $M' \in \mathcal{M}'$ is a limit of $\mathcal{M}$.*

A coverability set is an antichain*, if and only if it does not contain two ω-markings M_1 and M_2 such that $M_1 < M_2$.*

Every $M \in \mathcal{M}$ is the limit of the infinite growing sequence $M, M, M, \ldots$. Thus $\mathcal{M}$ is its own coverability set. However, it is not necessarily finite. To prove that each set of markings has a finite coverability set, we first show that the limit of an infinite growing sequence of limits is a limit of the original set.

Lemma 3. *Let $\mathcal{M}$ be a set of markings. For each $i > 0$, let M_i be any limit of $\mathcal{M}$ such that $M_1 \leq M_2 \leq \ldots$. Then also $\lim\limits_{i\to\infty} M_i$ is a limit of $\mathcal{M}$.*

Proof. For each i, let $M_{j,i}$ be an infinite growing sequence of elements of $\mathcal{M}$ such that $\lim\limits_{j\to\infty} M_{j,i} = M_i$. Let $M'_1 = M_{1,1}$. When $i > 1$, let M'_i be the first element of $M_{j,i}$ such that for each $p \in P$, either $M'_i(p) = M_i(p) < \omega$ or $M'_{i-1}(p) \leq M'_i(p) \geq i$ and $M_i(p) = \omega$. It exists by Lemma 1. Furthermore, $M'_{i-1}(p) \leq M_{i-1}(p) \leq M_i(p)$, so if $M'_i(p) = M_i(p)$, then $M'_{i-1}(p) \leq M'_i(p)$. Thus M'_i is an infinite growing sequence of elements of $\mathcal{M}$.

If $M_i(p) < \omega$ for each i, then $M'_i(p) = M_i(p)$ for each $i > 1$, so $M'_i(p)$ has the same limit as $M_i(p)$. Otherwise, from some value of i on, $M_i(p) = \omega$ and $M'_i(p) \geq i$. Then the limit of $M'_i(p)$ is ω, which is also the limit of $M_i(p)$. As a consequence, $\lim\limits_{i\to\infty} M'_i = \lim\limits_{i\to\infty} M_i$. □

We say that an element a of a set A is *maximal*, if and only if there is no $b \in A$ such that $a < b$. Let $[\mathcal{M}]$ denote the set of all limits of $\mathcal{M}$, and let $\lceil \mathcal{M} \rceil$ denote the set of the maximal elements of $[\mathcal{M}]$. We are ready to prove the central result of this section.

Theorem 1. *Each set $\mathcal{M}$ of markings has a coverability set that is an antichain. It is finite and unique. It consists of the maximal elements of the limits of $\mathcal{M}$.*

Proof. Obviously $[\mathcal{M}]$ satisfies part 2 of Definition 4. It also satisfies part 1, because each $M \in \mathcal{M}$ is the limit of $M, M, M, \ldots$.

We prove next that for every $M \in [\mathcal{M}]$, there is an $M' \in \lceil \mathcal{M} \rceil$ such that $M \leq M'$. Let $M_{0,1} = M$ and $j = 0$. For each $i > 1$ such that $M_{j,i-1}$ is not maximal in $[\mathcal{M}]$, there is an $M_{j,i} \in [\mathcal{M}]$ such that $M_{j,i-1} < M_{j,i}$. If this sequence ends, then the last $M_{j,i}$ qualifies as the M'. Otherwise, let $M_{j+1,1} = \lim_{i \to \infty} M_{j,i}$. Clearly $M_{j+1,1} \geq M_{j,1} \geq M$. By Lemma 3, $M_{j+1,1} \in [\mathcal{M}]$. We repeat this reasoning with $j = 1$, $j = 2$, and so on as long as possible. Because $M_{j,i}$ is strictly growing as i grows, $M_{j+1,1}$ has more ω-symbols than $M_{j,1}$. Therefore, $M_{j,i}$ has at least j ω-symbols. This implies that j cannot grow beyond $|P|$. So a maximal element is eventually encountered.

Thanks to this, $\lceil \mathcal{M} \rceil$ satisfies part 1 of Definition 4. It clearly also satisfies the rest of Definition 4. So an antichain coverability set exists.

If $\mathcal{M}'$ is an infinite coverability set of $\mathcal{M}$, then it is possible to pick an infinite sequence of distinct elements from $\mathcal{M}'$. By Lemma 2, it has an infinite growing subsequence M_i. Because all its elements are distinct, we have $M_1 < M_2$. Therefore, infinite coverability sets are not antichains.

It remains to be proven that there are no other antichain coverability sets. We have already ruled out infinite sets, so let $\mathcal{M}'$ be any finite coverability set of $\mathcal{M}$. Part 2 of Definition 4 yields $\mathcal{M}' \subseteq [\mathcal{M}]$.

For any $M \in \lceil \mathcal{M} \rceil$, let M_i be a sequence of elements of $\mathcal{M}$ whose limit is M. Because $\mathcal{M}'$ is finite, it must cover every M_i only using a finite number of ω-markings. Thus $\mathcal{M}'$ must contain an ω-marking M' that covers infinitely many of M_i. Definition 2 implies that $M \leq M'$. We have $M' \in \mathcal{M}' \subseteq [\mathcal{M}]$, and $M \in \lceil \mathcal{M} \rceil$ makes $M < M'$ is impossible. Therefore, $M' = M$.

We have proven that every finite coverability set $\mathcal{M}'$ satisfies $\lceil \mathcal{M} \rceil \subseteq \mathcal{M}' \subseteq [\mathcal{M}]$. If there is an M such that $M \in \mathcal{M}' \setminus \lceil \mathcal{M} \rceil$, then there is an $M' \in \lceil \mathcal{M} \rceil \subseteq \mathcal{M}'$ such that $M < M'$, so $\mathcal{M}'$ is not an antichain. As a conclusion, there is only one antichain coverability set. □

A coverability set is *minimal* if and only if none of its proper subsets is a coverability set. Next we will show that a coverability set is minimal if and only if it is an antichain. This will immediately yield the existence, finiteness, and uniqueness of minimal coverability sets.

Corollary 1. *Each set of markings has precisely one minimal coverability set. It is finite. It is the antichain coverability set.*

Proof. The claims follow by Theorem 1, if we prove that a coverability set is minimal if and only if it is an antichain. It is clear that a coverability set that is not an antichain has a proper subset that is a coverability set, because M_1 could be left out. Consider the antichain coverability set $\lceil \mathcal{M} \rceil$. Because it is finite, it cannot have any infinite set as a proper subset. By the proof of Theorem 1, every finite coverability set has $\lceil \mathcal{M} \rceil$ as a subset. So $\lceil \mathcal{M} \rceil$ cannot have a proper subset that is a coverability set. □

```
$F := \{\hat{M}\}$; $A := \{\hat{M}\}$; $W := \{\hat{M}\} \times T$; $\hat{M}.B :=$ nil
**while** $W \neq \emptyset$ **do**
    $(M, t) :=$ any element of $W$; $W := W \setminus \{(M, t)\}$
    **if** $\neg M[t\rangle$ **then** continue
    $M' :=$ the $\omega$-marking such that $M[t\rangle M'$
    **if** $M' \in F$ **then** continue
    Add-$\omega(M, M')$
    **if** $\omega$ was added **then if** $M' \in F$ **then** continue
    Cover-check$(M')$      // may update $A$ and $W$
    **if** $M'$ is covered **then** continue
    $F := F \cup \{M'\}$; $A := A \cup \{M'\}$; $W := W \cup (\{M'\} \times T)$; $M'.B := M$
```

Fig. 1. The basic coverability set algorithm

3 Basic Algorithm

In this section we present, discuss, and prove correct the simplest of the algorithms in this paper, and its variant that uses what we call history merging. We believe that they are new.

A *Petri net* is a tuple $(P, T, W, \hat{M})$ such that $P \cap T = \emptyset$, $\hat{M}$ is a function from P to $\mathbb{N}$, and W is a function from $(P \times T) \cup (T \times P)$ to $\mathbb{N}$. For this publication, we assume that P and T are always finite. The elements of P, T, W, and $\hat{M}$ are called *places*, *transitions*, *weights*, and *initial marking* respectively. The firing rule for ω-markings is the same as with markings: $M[t\rangle M'$ if and only if for each $p \in P$, $M(p) \geq W(p, t)$ and $M'(p) = M(p) - W(p, t) + W(t, p)$. Whether or not $M[t\rangle$ holds, is determined by the places for which $M(p) < \omega$. If $M(p) < \omega$, then $M'(p) = M(p) - W(p, t) + W(t, p) < \omega$. If $M(p) = \omega$, then also $M'(p) = \omega$. We define $M\,[t_1 t_2 \cdots t_n\rangle\, M'$ in the usual way.

Overview. The algorithm is shown in Fig. 1. The ω-markings that have been generated and taken into consideration, are stored in the set F. We call these *found* ω-markings. In our test implementation, F is presented as a hash table. There is a base table of pointers to ω-markings that is indexed by the hash value of the ω-marking. Each ω-marking has a pointer to the next ω-marking in the hash list.

The set of found ω-markings is divided to sets of *active* and *passive* ω-markings. The set of active ω-markings is denoted with A, and passive are those that are in F but not in A. In our test implementation, A is represented by a linked list, maintained by another pointer in the ω-marking structure.

Each ω-marking M' has a *back pointer* $M'.B$ that points to the ω-marking M such that M' was first found by firing a transition from M, except that it points to nowhere in the case of the initial marking. Using the back pointers one can scan the history of M' up to the initial marking.

Finally, W is a *workset* that keeps track of the work to be done. The minimal coverability set can be constructed in many different orders, including breadth-first, depth-first, and what we call "most tokens first". To model this generality,

```
Add-$\omega(M, M')$
last := $M$; now := $M$; added := False
repeat
    if now $< M' \wedge \exists p \in P$ : now$(p) < M'(p) < \omega$ then
        added := True; last := now
        for each $p \in P$ such that now$(p) < M'(p) < \omega$ do
            $M'(p) := \omega$
    if now.$B$ = nil then now := $M$ else now := now.$B$
until now = last
```

Fig. 2. The basic version of ω-addition

in Fig. 1, W contains pairs consisting of an ω-marking and a transition. In practice it suffices to store ω-markings instead of pairs. In our test implementation, the workset is a queue, stack, or heap containing pointers to ω-markings, and each ω-marking has an integer attribute *next_tr* containing a number of a transition. If M is in the workset of the implementation, the pairs (M, t) in the W of Fig. 1 are the ones where the number of t is at least $M.\mathit{next\_tr}$. When we say that M is in the workset, we mean that $(M, t) \in W$ for some $t \in T$.

Initially the initial marking $\hat{M}$ has been found and is active, and the workset contains $\hat{M}$ paired with every transition. The algorithm runs until the workset is empty. Intuitively, the workset contains the ω-markings which still may contain something of interest for the minimal coverability set. In each iteration of the main loop, the algorithm selects and removes one pair (M, t) from the workset. Then it tries to fire t from M. If t cannot be fired from M, then the main loop rejects the pair and goes to the next pair. This is shown in the figure with the word "continue" that means a jump to the test of the **while**-loop.

If the firing of t from M succeeds, the algorithm checks whether the resulting ω-marking M' has already been found. If found, M' is rejected. Otherwise the operation Add-ω is applied to M'. It adds ω-symbols to M' as justified by the history of M'. We will discuss the operation in more details soon.

If M' was changed by Add-ω, then the algorithm tests again whether the resulting ω-marking has already been found and rejects it if it is. If M' survived or avoided this test, one more test is applied to it. Cover-check(M') finds out if M' is covered by any ω-marking in A. It also removes from A those ω-markings that M' covers strictly. Therefore, A is always the set of maximal found ω-markings. We will soon discuss this in more details. Cover-check also removes from W each pair whose first component was removed from A.

If M' passes all these tests, it is added to the found ω-markings and made active. It is also added to the workset paired with every transition. Its back pointer is made to point to the previous ω-marking.

Add-$\boldsymbol{\omega}$. Add-ω is shown in Fig. 2. It scans the history of M towards the initial marking at least once. It tries to find an ω-marking $M'' = \mathit{now}$ that is strictly covered by M', in such a way that M' does not yet have ω in all those places where $M''(p) < M'(p)$. Whether such a covered ω-marking was found is recorded

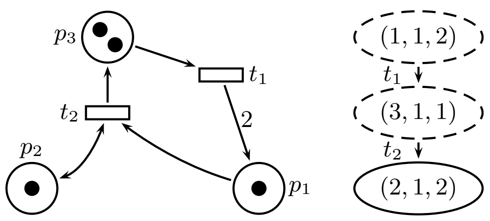


Fig. 3. The advantage of repeated scanning of history

in the boolean variable *added*. The operation then sets $M'(p) = \omega$ for places where previously $M''(p) < M'(p) < \omega$. Then it moves to the next (or perhaps one should say previous) ω-marking, and so on. The operation terminates when it has fully tested the history without being able to add new ω-symbols.

The purpose of repeated scanning of the history is to add as many ω-symbols as possible to M'. For example (see Fig. 3), let $(1,1,2)\ [t_1\rangle\ (3,1,1)\ [t_2\rangle\ (2,1,2)$. For each $n \in \mathbb{N}$ except 0 and 1 we have $(1,1,2)\ [(t_1t_2)^{2n-3}t_2^{n-2}\rangle\ (n,1,n)$, where σ^i means σ repeated i times. So $(\omega,1,\omega)$ is a limit of reachable markings. Add-ω checks whether $(2,1,2)$ covers $(3,1,1)$. It does not. Then it checks whether $(2,1,2)$ covers $(1,1,2)$. It does, so Add-ω converts $(2,1,2)$ to $(\omega,1,2)$. Then it has found the end (or beginning) of the history. If it terminated there, the result would be $(\omega,1,2)$. However, Add-ω starts anew at $(3,1,1)$ and sees that $(\omega,1,2)$ covers it. Therefore, it converts $(\omega,1,2)$ to $(\omega,1,\omega)$.

We will later see that inserting an ω-marking to the data structures is an expensive operation. By Corollary 1 and Theorem 1, the algorithm only has to maintain *maximal* ω-markings. Therefore, first inserting $(\omega,1,2)$ and later removing it and inserting $(\omega,1,\omega)$ is disadvantageous compared to just inserting $(\omega,1,\omega)$. In the worst case, fully testing the history after the last addition of ω doubles the running time of Add-ω, which is a relatively small price.

After each addition of an ω-symbol, scanning is continued from where it was instead of starting anew at M, because intuition suggests that the least recently tried ω-markings have the best chance of success. However, this is not a theorem but a heuristic.

We write $M[t\rangle^{\omega}M'$ to denote that M' is obtained by firing t from M and then executing Add-$\omega(M, M')$. This notion depends on not only M and t, but also on the history of M.

Cover-Check. The purpose of Cover-check(M') is to ensure that A always consists of the maximal ω-markings in F. In our test implementation, A is represented as a linked list, which is scanned by Cover-check. If it finds an element M''' that is strictly covered by M', it removes M''' from the list and removes (M''', t) from the workset for every $t \in T$. In our test implementation, the latter is done simply by assigning to M'''.*next_tr* a number that is greater than the number of any transition. Then Cover-check continues scanning.

If Cover-check finds that M' is covered by an element M'' of the list, it terminates immediately, because then it is not possible that M' covers strictly any element in the list. This is because if $M' > M'''$ and M''' is in the list, then $M'' \geq M' > M'''$, so the list does not consist of only maximal elements.

While the test whether a given ω-marking has already been found can be performed very efficiently with hash tables, testing whether a given ω-marking strictly covers any found ω-marking seems much more difficult. We are not aware of essentially better approaches than comparing the new ω-marking one by one to each old ω-marking, with some heuristics to speed the procedure up a little. Therefore, it makes sense to try to reduce the number of times Cover-check is called. It also makes sense to try to keep A small, because the cost of the call is often and at most proportional to the size of A. For both goals, it seems advantageous to get as many ω-symbols as possible to the ω-markings as early as possible. However, this is not a theorem but an intuitive heuristic.

This is also the reason for the presence of F in the algorithm. Correctness does not require it, because Cover-check and A suffice for filtering out later instances of any found ω-marking M. If the earlier instance of M has been removed from A, it happened in favor of some M' that strictly covers M, so the filtering effect remains. However, detecting repeated instances of the same ω-marking with a hash table costs next to nothing, while the cost of the coverability check is significant. Repeated instances of the same ω-marking are common with Petri nets, because if $M[t_1 t_2\rangle M_{12}$ and $M[t_2 t_1\rangle M_{21}$, then $M_{12} = M_{21}$. Therefore, it makes sense to implement a special mechanism for them that is much faster than the general mechanism.

Also Add-ω is costly compared to the test whether $M' \in F$. Performing the test before Add-ω and after each addition of ω-symbols inside Add-ω would speed Add-ω up, but would also globally slow down the adding of ω-symbols, because the new instance of the same ω-marking usually has a different history and may thus introduce ω-symbols to different places. We believe that most calls of Add-ω do not lead to the addition of ω-symbols, and therefore we believe that it is advantageous to test $M' \in F$ before calling Add-ω. On the other hand, if Add-ω has already succeeded in adding an ω-symbol, then the chances of finding a new maximal ω-marking are improved, so it seems better to let it continue. Again, this is a heuristic argument, and we do not really know the actual effect.

History Merging. History merging is a variant of the basic algorithm. In it, $M.B$ is a set of (pointers to) any number of predecessor ω-markings, instead of being a single ω-marking. This mirrors the fact that the same ω-marking may be reached in multiple ways, any of which may justify the addition of ω-symbols.

If M' is rejected on line 6 or 8 of Fig. 1, then it already has a representation in F. In history merging, the program inserts M to the predecessor set of M'. Thus the predecessor set collects pointers to all ω-markings from which M' was reached by firing one transition and possibly executing Add-ω.

Consider the example in Fig. 4. Suppose the algorithm proceeds in a breadth-first manner. It finds the ω-markings $(0, 1, 0, 0)$ and $(0, 0, 1, 0)$, by firing t_1 and t_2 from $(1, 0, 0, 0)$. It then generates $(0, 0, 0, 1)$ by firing t_3 from $(0, 1, 0, 0)$.

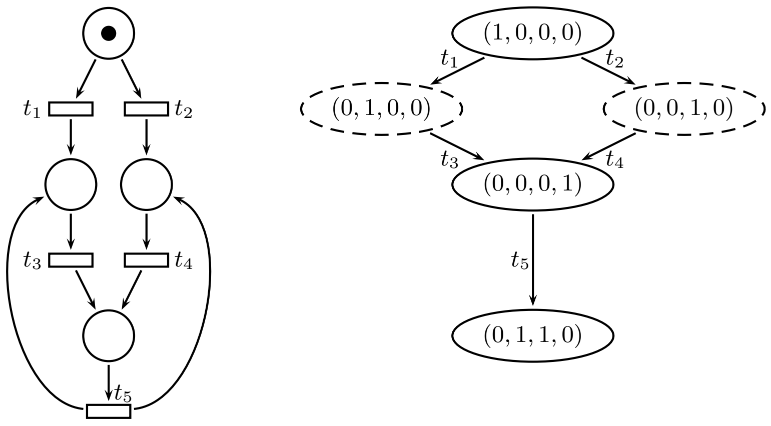


Fig. 4. How history merging helps

The same ω-marking is also found by firing t_4 from $(0,0,1,0)$, at which time we can update the history of $(0,0,0,1)$ to contain both ω-markings. The benefit comes when we add ω-symbols to $(0,1,1,0)$: as both these ω-markings are in its history, we can add two omegas, giving $(0,\omega,\omega,0)$.

History merging may also be applied on line 10 of Fig. 1, if for each $p \in P$ we have $M''(p) = M'(p)$ or $M''(p) = \omega$, where M'' is the ω-marking that covers M'. If the condition holds in the opposite direction, the histories of those ω-markings that M' is found to strictly cover on line 9, can be merged into the history of M'. In these cases, the changes in the numbers of tokens in those places whose marking is not ω are represented correctly along any path despite the mergings. For the remaining places, the changes do not matter, because their ω-marking is ω also in all later ω-markings. Our test implementation does not yet have this feature.

With history merging, the history of an ω-marking forms a directed graph that is partially shared by the histories of other ω-markings. It can be scanned in time that is linear in its size. After each addition of ω-symbols, our test implementation starts a new scan where it was and continues it at M analogously to Fig. 2, to guarantee that at termination no ω-marking in the history can justify the addition of more ω-symbols to M'.

Repeated scans require repeated resetting of "found" information, which may become a performance problem if implemented naïvely. In our test implementation, each ω-marking has an attribute *search_nr* and there is a global variable *search_now*. When an ω-marking is found, *search_now* is assigned to its *search_nr*. In the beginning of each search, *search_now* is incremented, and if it overflows its type, it is set to 1 and the *search_nr* of every found ω-marking is set to 0 by scanning the hash table that implements F.

Correctness. The correctness of the algorithm consists of four issues, three of which correspond to the three conditions in Definition 4 and the fourth is the termination of the algorithm. We present a lemma for each. In the proofs, we will use the following obvious fact: if $M[t\rangle M'$ and $M \leq M_1$, then there is an M_1' such that $M_1[t\rangle M_1'$ and $M' \leq M_1'$.

Lemma 4. *After termination, for every reachable marking M of the Petri net, A contains an ω-marking M' such that $M \leq M'$.*

Proof. Each time when an ω-marking is inserted to F, it is also inserted to A. Each time when an M is removed from A, an M' such that $M < M'$ is inserted to A. Therefore, the algorithm maintains the following invariant:

I1: For each $M \in F$, there is an $M' \in A$ such that $M \leq M'$.

Each time when an M is added to A, (M, t) is added to W for every $t \in T$. Each time when a pair (M, t) is removed from W, either $\neg M[t\rangle$, or there is an M' such that $M[t\rangle M'$. In the latter case, the set F either contains an ω-marking M'' such that $M' \leq M''$, or such an M'' is inserted to F. Therefore, the algorithm also maintains the following invariant:

I2: For each $M \in A$ and $t \in T$, either $(M, t) \in W$, $\neg M[t\rangle$, or for the M' such that $M[t\rangle M'$ there is an $M'' \in F$ such that $M' \leq M''$.

Let R be the set of the reachable markings of the Petri net. If $M \in R$, then there is a sequence $M_0 \,[t_1\rangle\, M_1 \,[t_2\rangle\, \cdots \,[t_n\rangle\, M_n$ such that $M_0 = \hat{M}$ and $M_n = M$. We prove by induction that for each $0 \leq i \leq n$ there is an $M_i' \in A$ such that $M_i \leq M_i'$. The claim holds for $i = 0$ by I1, because $\hat{M}$ is found initially. After termination $(M_{i-1}', t_i) \notin W$, because then $W = \emptyset$. We also cannot have $\neg M_{i-1}'[t_i\rangle$, because $M_{i-1}[t_i\rangle M_i$ and $M_{i-1} \leq M_{i-1}'$. Let M_i''' be such that $M_{i-1}'[t_i\rangle M_i'''$. By I2 there is an $M_i'' \in F$ and by I1 an $M_i' \in A$ such that $M_i \leq M_i''' \leq M_i'' \leq M_i'$. □

Lemma 5. *Every element of F (and thus of A) is a limit of the set of the reachable markings of the Petri net.*

Proof. Let R be the set of the reachable markings of the Petri net.

We show first that if $M[t\rangle M'$ and M is a limit of R, then also M' is a limit of R. We have $M(p) = \omega$ if and only if $M'(p) = \omega$. Let $P_\omega = \{p \in P \mid M(p) = \omega\} = \{p \in P \mid M'(p) = \omega\}$. Let d be the minimum of $W(t,p) - W(p,t)$ over $p \in P_\omega$. By Lemma 1, for every $n \in \mathbb{N}$, there is an $M_i \in R$ such that $W(p,t) \leq M_i(p) \geq n - d$ for every $p \in P_\omega$ and $M_i(p) = M(p)$ for every $p \in P \setminus P_\omega$. We have $M_i[t\rangle$. If M_i' is such that $M_i[t\rangle M_i'$, then $M_i'(p) = M_i(p) - W(p,t) + W(t,p) \geq M_i(p) + d \geq n$ for every $p \in P_\omega$ and $M_i'(p) = M(p) - W(p,t) + W(t,p) = M'(p)$ for every $p \in P \setminus P_\omega$. So M' is the limit of M_i' and thus a limit of R.

We show next that if M' is a limit of R and M'' is the result of applying Add-ω to it, then M'' is a limit of R. Consider any ω-marking $M = \mathit{now}$ that triggers addition of ω-symbols to M'. Let $t_1, \ldots, t_k$ be the transitions from M to M'. Let

d be the minimum of $\sum_{i=1}^{k} W(t_i, p) - W(p, t_i)$ over $p \in P$. Let e be the maximum of $\sum_{i=1}^{k} W(p, t_i)$ over $p \in P$. For each $p \in P$ we have either (1) $M'(p) = M''(p) = \omega$, (2) $M(p) = M'(p) = M''(p) < \omega$, or (3) $M(p) < M'(p) < M''(p) = \omega$.

By Lemma 1, for every $n \in \mathbb{N}$ there is an $M_i \in R$ such that $M_i(p) \geq n(1 - d)$ and $M_i(p) \geq ne$ for every p of kind 1, and $M_i(p) = M'(p)$ for the remaining places. For places of kind 1, ne suffices for firing $t_1 \cdots t_k$ n times in a row starting at M_i, and the result satisfies $M'_i(p) \geq M_i(p) + nd \geq n$. For places of kinds 2 and 3, $t_1 \cdots t_k$ can be fired once from M_i because it was possible to fire it from M. For kind 2, $M_i(p) = M(p) = M'(p) < \omega$, so $t_1 \cdots t_k$ can be fired n times and the result is $M'_i(p) = M_i(p) = M''(p)$. For kind 3, $M(p) < M_i(p) = M'(p) < \omega$, so $t_1 \cdots t_k$ can be fired repeatedly, each time adding at least one token to p. After n repetitions, $M'_i(p) \geq n$. We conclude that $M'' = \lim_{i \to \infty} M'_i$. Furthermore, $M_i \left[(t_1 \cdots t_k)^n\right\rangle M'_i$, so $M'_i \in R$ and M'' is a limit of R.

We have shown that each operation of the algorithm that introduces or modifies ω-markings yields a limit of R, if its input ω-markings are limits of R. Originally there is only the initial marking $\hat{M}$. It is obviously reachable and the limit of $\hat{M}, \hat{M}, \hat{M}, \ldots$. So all ω-markings found by the algorithm are limits of R. □

Lemma 6. *The set A is always an antichain.*

Proof. This is trivial, because it is explicitly ensured by lines 9 to 11 of Fig. 1, $\{\hat{M}\}$ is an antichain, and no other operation modifies the contents of A. □

Lemma 7. *The algorithm terminates.*

Proof. Termination of loops other than the main loop of the algorithm and Add-ω are obvious. Add-ω stops adding ω-symbols to M' at the latest when $M'(p) = \omega$ for every $p \in P$, so each call of Add-ω terminates.

The only way in which the main loop of the algorithm gets new work to do is that a new ω-marking is found that is different from all earlier ones. Each old ω-marking and each transition give rise to at most one new ω-marking. Therefore, if the longest acyclic history of any found ω-marking is of length ℓ, then at most $1 + |T| + |T|^2 + \ldots + |T|^\ell$ ω-markings are found. This is a finite number.

We conclude that failure of termination requires the existence of an infinite sequence $\hat{M} = M_0 \; [t_1\rangle^\omega \; M_1 \; [t_2\rangle^\omega \; M_2 \; [t_3\rangle^\omega \cdots$ such that each M_i is first found by firing t_i from M_{i-1}, and then possibly adding ω-symbols with Add-ω. The M_i are distinct because of the tests $M' \in F$. By Lemma 2, $M_0, M_1, M_2, \ldots$ has an infinite strictly growing subsequence $M'_0 < M'_1 < M'_2 < \cdots$. Thanks to Add-$\omega$, each M'_{i+1} has at least one ω-symbol more than M'_i. However, there are only $|P|$ places, so we run out of places where to add new ω-symbols at $M'_{|P|+1}$, if not earlier. This is a contradiction. So failure of termination is impossible. □

Together Lemmas 4 to 7 yield the following theorem.

Theorem 2. *The algorithm in Fig. 1 terminates, and then A contains the minimal coverability set of the Petri net.*

```
$F := \{\hat{M}\}$; $A := \{\hat{M}\}$; $Q := \{\hat{M}\}$; $\hat{M}.B :=$ nil
while $Q \neq \emptyset$ do
    $M :=$ the first element of $Q$; $Q := Q \setminus \{M\}$
    for $t \in T$ do
        if $M \notin A$ then continue
        lines 4, ..., 10 of Fig 1
        $F := F \cup \{M'\}$; $A := A \cup \{M'\}$; add $M'$ to the end of $Q$; $M'.B := M$
```

Fig. 5. Breadth-first discipline

4 Construction Order

The algorithm in the previous section does not specify the order in which the pairs (M, t) are picked from W, and the correctness of the algorithm does not depend on it. In this section we discuss some possible orderings.

The *age* of a pair is defined as the time (for example, the number of iterations of the main loop) elapsed since the pair was inserted to W. The age of (M, t) is determined by M, because the pairs (M, t) for every $t \in T$ are inserted simultaneously to W.

Many state space verification algorithms require traversing the state space in a specific order. With ordinary state spaces, it is customary to construct the state space in that order, so as to enable running the verification algorithm on-the-fly. With coverability sets, however, due to the high cost of unnecessary construction of non-maximal ω-markings, it may be better to construct the set in the order best suited for coverability sets and then, using the minimal coverability set as a starting point, re-generate the transitions in the order required by the verification algorithm.

Breadth-First. Breadth-first discipline is obtained by always picking one of the oldest pairs from W. One possible implementation is described in Fig. 5, where Q (to reflect the fact that it is a queue) is used instead of W of Fig. 1. With this implementation, the attribute *next_tr* is not needed. To save more memory, A and Q can actually be in the same linked list. The list contains first those elements of A that are not in Q, and then the elements of Q. This implementation automatically retains the property $Q \subseteq A$ when an element is removed from A. There are three common pointers: to the beginning, to the beginning of Q, and to the end.

New ω-markings are added to the end, and $Q := Q \setminus \{M\}$ is implemented by moving the middle pointer (the beginning of Q) one step forward.

Let $M.next\_A$ denote the pointer of M that points to the next ω-marking in the list. The test $M \notin A$ can be done in constant time: $M.next\_A$ is made to point to M after M is removed from A. Actually, it would be correct to skip the test, but then the algorithm would unnecessarily fire transitions from ω-markings that are no longer maximal.

From the above we deduce that the breadth-first discipline has a simple and memory-saving implementation. Furthermore, breadth-first is usually more amenable to parallel implementation than other common disciplines. However, it seems intuitively that it typically adds ω-markings later and thus should have longer running time than other common disciplines. In our measurements (see Section 6), breadth-first was never clearly the fastest but was often clearly the slowest. So we do not recommend breadth-first. Like in many other arguments in this publication, this is a heuristic and not a theorem.

Indeed, it is possible to construct a situation where breadth-first works better than any other approach. Consider an arbitrary Petri net $(P, T, W, \hat{M})$, for which the construction of the minimal coverability set takes some considerable time to finish. We add to it one place p_1 and two transitions t and t' in the following way: $W(p,t) = 0$ and $W(t,p) = 1$ for every $p \in P$. $W(p,t') = 0$ and $W(t',p) = \hat{M}(p)$ for $p \in P$, and $W(t',p_1) = 0$. $W(p_1,t) = W(p_1,t') = W(t,p_1) = 1$. A new initial marking $\hat{M}'$ is such that $\hat{M}'(p_1) = 1$, and all other places are empty. The ordering of transitions is such that t' is the first transition to be fired, and t is the second.

Now, breadth-first fires t as the second transition from the initial marking, resulting in $(\omega, \ldots, \omega)$ and quick termination of the algorithm. With many other disciplines, such as depth-first, the algorithm fires t', which "primes" the original Petri net, after which the algorithm runs its course exactly as with the original Petri net. It explores t only as the very last transition.

Depth-First. Depth-first discipline is obtained by always picking a youngest pair from W. It can be implemented by storing the ω-markings of the pairs in a stack; returning $(M, M.\mathit{next\_tr})$ as the pair, where M is the top element of the stack; and popping the top element when it has no more unused transitions.

Depth-first also has a well-known recursive implementation. It has the advantage that *next_tr* is not needed, making it conceptually simpler. On the other hand, each recursion level consumes some memory, and the recursive calls consume some time. Therefore, the recursive implementation is likely to be at least marginally less efficient.

Intuitively, depth-first typically adds ω-markings early on, because of the following result. Thus it should have a good running time. In our measurements it was seldom the fastest and seldom much worse than the fastest.

We say that M' is a *successor* of M if and only if there is a t such that the algorithm at some point fires $M[t\rangle^{\omega} M'$ and either puts M' into F or detects that it is there already. A *descendant* of an ω-marking is the ω-marking itself, its successor, or a descendant of its successor.

Lemma 8. *If the construction order is depth-first and M has more ω-symbols than the ω-marking via which it was first found, then the algorithm will not backtrack from M before it has investigated all descendants of M.*

Proof. Let an ω-marking be *black*, if it has been backtracked from; *grey*, if it has been found but not yet backtracked from; and *white*, if it has not been found. Depth-first search has the property that the set of grey ω-markings and the transitions via which they were first found, constitute a path from the initial marking to the current ω-marking. We call any contiguous sub-path of this path a *grey path*.

Each black ω-marking has been removed from W. So the successors of any black ω-marking are grey or black. A black ω-marking may have white descendants, but each path to any of them goes via at least one grey ω-marking.

If M_2 is a descendant of M_1, then M_2 has ω-symbols in at least the same places as M_1. Assume that the algorithm is about to backtrack from M to M', where M has more ω-symbols than M'. Then no descendant of M can be along the grey path from $\hat{M}$ to M'. Thus none of them can be grey, implying that none of them can be white either. Hence, they are all black. □

Most Tokens First. The desire to add ω-symbols as early as possible naturally leads to the heuristic of always trying next the ω-marking that has the most tokens. The ω-marking with the maximal number of ω-symbols is preferred, and if it is not unique, then the total number of tokens in the places whose marking is not ω is used as the criterion. Like before, only ω-markings are stored in the workset, and *next_tr* is used to get the transition component of the pair (M, t). If the workset is implemented as a heap and contains w ω-markings, then each operation on it takes $O(\log w)$ time.

In our measurements, this discipline was often both the fastest and constructed the smallest number of ω-markings. It lost to depth-first a small number of times, and often there was no clear difference. It may be remarkable that it lost in the biggest example. However, our set of measurements is far too small for firm conclusions.

5 To Prune or Not to Prune

In this section we discuss pruning of active ω-markings and whether it is better than the algorithm in Section 3.

Pumping Cycle Passivation. Consider M_0 in the history of M_n such that M_0 triggered the addition of at least one ω-symbol to M_n along the path $M_0\ [t_1\rangle^\omega\ M_1\ [t_2\rangle^\omega\ \ldots\ [t_n\rangle^\omega\ M_n$. Then $M_0 < M_n$ and $M_n\,[t_1 \cdots t_n\rangle$. When $0 \leq i \leq n$, let M'_i be the ω-marking such that $M_n\ [t_1 \cdots t_i\rangle\ M'_i$. Clearly $M_i < M'_i$ for each $0 \leq i < n$. So eventually $M_0, \ldots, M_{n-1}$ will not be maximal. The firing of those transitions from them that have not already been fired seems wasted work.

Therefore, it seems a good idea to passivate or remove $M_0, \ldots, M_{n-1}$ altogether, when ω-symbols are added to M_n. By *passivation* we mean the removal from W and A, but not from F. (The removal of M from W means the removal of (M, t) from W for every $t \in T$.) By *removal* we mean the removal from all data structures. The algorithm in [1] removes $M_0, \ldots, M_{n-1}$.

We argue that the removal of $M_0, \ldots, M_{n-1}$ is not a good idea in general. Firstly, keeping them in F costs very little, but prevents the algorithm from constructing their futures, if they are constructed anew. As we have pointed out, reaching the same marking many times is common with Petri nets.

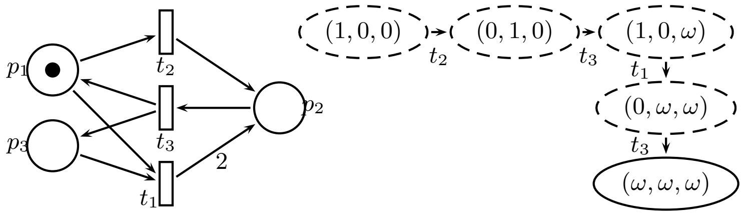


Fig. 6. A pumping ω example

Secondly, the removal may slow down the addition of ω-symbols. Consider the example in Fig. 6, assuming depth-first discipline. The algorithm in Fig. 1 fires $(1,0,0)$ $[t_2\rangle$ $(0,1,0)$ $[t_3\rangle$ $(1,0,1)$ and converts $(1,0,1)$ to $(1,0,\omega)$. Then it fires $(1,0,\omega)$ $[t_1\rangle$ $(0,2,\omega)$ and converts $(0,2,\omega)$ to $(0,\omega,\omega)$, because it covers $(0,1,0)$. Finally $(0,\omega,\omega)$ $[t_3\rangle$ $(1,\omega,\omega) > (0,\omega,\omega)$, yielding (ω,ω,ω). Transitions were fired altogether four times. However, if M_0, …, M_{n-1} are removed, then $(1,0,0)$ and $(0,1,0)$ are removed after constructing $(1,0,\omega)$. Next $(1,0,\omega)$ $[t_1\rangle$ $(0,2,\omega)$ $[t_3\rangle$ $(1,1,\omega)$ are fired, yielding $(1,\omega,\omega)$. Again, all other ω-markings are removed. Firing t_1 and t_2 from $(1,\omega,\omega)$ do not yield new maximal ω-markings, but t_3 yields (ω,ω,ω). So seven transition firings were needed.

Future Pruning. Pruning of futures refers to the passivation or removal of some or all found ω-markings whose histories contain an ω-marking that was strictly covered by a newly found ω-marking. Pumping cycle passivation can be considered as a special case of future pruning. The algorithms in [1] and [6] both perform some more general form of future pruning.

Correctness of future pruning is tricky, and not all forms are correct. The counter-example presented in [2] reveals a flaw in the future pruning of the algorithm in [1]. In the counter-example, an ω-marking M_1 first triggers pumping cycle removal. Then another ω-marking M_2 with a different history is found, and its successor ω-markings are covered by M_1. Therefore, M_2 remains active but does not lead to any new ω-markings. An ω-marking in the (removed) pumping cycle is covered by M_2, but the algorithm fails to notice this, since the cycle's ω-markings have been removed. Finally, a third ω-marking M_3 is found that covers strictly some ω-marking in the history of M_1, and M_1 is removed. The firing of transitions from M_3 leads to an ω-marking that is covered by M_2, and exploration stops short of finding an ω-marking that covers M_1.

In [6], the algorithm never removes, only passivates ω-markings. The presence of these passive ω-markings in the histories of active ω-markings means that pruning happens differently from [1]. When a pumping cycle is found, the intermediate ω-markings are passivated, but they remain in the history of the new ω-marking. Whenever a new ω-marking M covers some ω-marking not in its own history, the whole branch starting from that ω-marking is passivated, even if the covered ω-marking is passive.

This avoids the behaviour described above. When M_1 in the counter-example triggers pumping cycle passivation, the intermediate ω-markings remain in a tree. On the way to M_2 the algorithm encounters another ω-marking, M, that covers a passive ancestor of M_1. The algorithm passivates the branch of M_1 when adding M, so by the time it gets to M_2, M_1 is no longer active, and the search will continue from M_2.

Unfortunately, this technique requires checking whether the new ω-marking M strictly covers any element in F (excluding the history of M). This is a disadvantage, because otherwise checking coverage against A would suffice, A may be much smaller than F, and checking coverage is expensive.

Is It Worth the Effort? Our first observation is that the running time may depend heavily on finding a certain ω-marking early on. By exploiting this, it is possible to design Petri nets so that either algorithm is faster in the particular case. This is illustrated by the tables below. The arc weights are shown in the table in the format $-W(p,t), W(t,p)$. The initial marking is $(1,0,0)$. We consider the most tokens first order, and at the same ω-marking, transitions are tried in the numeric order. (We have designed a similar example for depth-first order.) Like in [6] and unlike in Section 3, we assume that only active ω-markings are taken into account in Add-ω.

	t_1	t_2	t_3	t_4	t_5
p_1	$-1,2$	$-1,0$	$-0,1$	$-0,0$	$-0,1$
p_2	$-0,1$	$-0,2$	$-2,0$	$-1,1$	$-0,0$
p_3	$-1,0$	$-0,0$	$-1,0$	$-0,1$	$-1,1$

	t_1	t_2	t_3	t_4	t_5
p_1	$-1,2$	$-1,0$	$-0,1$	$-0,1$	$-0,0$
p_2	$-0,0$	$-0,2$	$-2,0$	$-1,2$	$-1,1$
p_3	$-1,0$	$-0,0$	$-1,0$	$-1,0$	$-0,1$

With the first Petri net, both algorithms fire first $(1,0,0)$ $[t_2\rangle^\omega$ $(0,2,0)$ $[t_4\rangle^\omega$ $(0,2,\omega)$ $[t_3\rangle^\omega$ $(1,0,\omega)$ and passivate at least $(0,2,0)$ and $(1,0,0)$. The pruning algorithm also passivates $(0,2,\omega)$ simultaneously with $(1,0,0)$. Then it fires $(1,0,\omega)$ $[t_1\rangle^\omega$ (ω,ω,ω), passivates all other ω-markings, fires (ω,ω,ω) $[t_i\rangle^\omega$ (ω,ω,ω) for $1 \le i \le 5$, and terminates. The non-pruning algorithm continues with $(0,2,\omega)$, because it has more tokens than $(1,0,\omega)$. It fires $(0,2,\omega)$ $[t_4\rangle^\omega$ $(0,2,\omega)$ and $(0,2,\omega)$ $[t_5\rangle^\omega$ $(\omega,2,\omega)$ $[t_1\rangle^\omega$ (ω,ω,ω), fires each t_i, and terminates. So the pruning algorithm is faster. (Thanks to $(1,0,0)$, the Add-ω in Section 3 would have yielded $(0,2,\omega)$ $[t_5\rangle^\omega$ (ω,ω,ω).)

With the second Petri net, both algorithms fire first $(1,0,0)$ $[t_2\rangle^\omega$ $(0,2,0)$ $[t_5\rangle^\omega$ $(0,2,\omega)$ $[t_3\rangle^\omega$ $(1,0,\omega)$. The non-pruning algorithm continues $(0,2,\omega)$ $[t_4\rangle^\omega$ (ω,ω,ω), while the pruning algorithm fires $(1,0,\omega)$ $[t_1\rangle^\omega$ $(\omega,0,\omega)$ $[t_1\rangle^\omega$ $(\omega,0,\omega)$ $[t_2\rangle^\omega$ (ω,ω,ω). So with this Petri net, the non-pruning algorithm is faster.

Our second observation is that the pruning algorithm may activate the same ω-marking more than once, leading to repeated work. To illustrate this, let the t_1 of the second Petri net be replaced by a transition that takes two tokens from each of p_2 and p_3, and puts three tokens to a new place p_4. There is also a transition t_6 that moves a token from p_4 to p_5. After constructing $(0,2,\omega,0,0)$, the algorithm fires $(0,2,\omega,0,0)$ $[t_1t_6t_6t_6\rangle^\omega$ $(0,0,\omega,0,3)$. Then it fires $(0,2,\omega,0,0)$ $[t_3\rangle^\omega$ $(1,0,\omega,0,0)$, notices that $(1,0,0,0,0)$ is covered, and passivates all ω-markings other than $(1, 0, \omega, 0, 0)$. Next it fires $[t_2\rangle^\omega$, activating $(0, 2, \omega, 0, 0)$ again. Then it fires $(0, 2, \omega, 0, 0)$ $[t_1 t_6 t_6 t_6\rangle^\omega$ $(0, 0, \omega, 0, 3)$ for a second time.

The goal of pruning is to avoid unnecessarily investigating ω-markings that will later be strictly covered by other ω-markings. Fortunately, the following theorem says that if the construction order is depth-first and history merging is applied, this happens automatically, without any explicit future pruning.

Theorem 3. *Let the construction order be depth-first and history merging be applied. Assume that M_0 $[t_1 \cdots t_n\rangle^\omega$ M_n and $M_0 < M'_0$. Assume that all transitions along the path M_0 $[t_1 \cdots t_n\rangle^\omega$ M_n were found before M'_0. After finding M'_0, the algorithm will not fire transitions from M_n, unless M'_0 $[t_1 \cdots t_n\rangle$ M_n.*

Proof. Let $M_1, \ldots, M_{n-1}$ be defined in the obvious way. Consider the moment when M'_0 has just been found. If M_n is black, then it has no more transitions to fire. From now on we assume that M_n is grey.

There is a grey path from M_n to the newest ω-marking, that is, M'_0. We denote its transitions and ω-markings with $t_{n+1}, \ldots, t_m$ and $M_{n+1}, \ldots, M_m$, where $M_m = M'_0$. We have M_0 $[t_1 \cdots t_n\rangle^\omega$ M_n $[t_{n+1} \cdots t_m\rangle^\omega$ M'_0, and $M_0 < M'_0$. Thanks to Add-ω, for each $p \in P$, $M'_0(p) = M(p)$ or $M'_0(p) = \omega$. In particular, M'_0 has more ω-symbols than M_0.

Along any path, the marking of any place may change from finite to ω but not vice versa. Let M'' be the first ω-marking along the grey path that has ω-symbols in precisely the same places as M'_0. By Lemma 8, the algorithm will not backtrack from M'' before it has investigated all descendants of M''. Therefore, currently the only investigated transition from any non-descendant of M'' to any descendant of M'' is the transition via which M'' was first found. Because also the path M_0 $[t_1 \cdots t_m\rangle^\omega$ M'_0 has such a transition, it must be the same transition. So $M'' = M_h$ for some $0 < h \leq m$.

Let M''_n be the ω-marking such that M'_0 $[t_1 \cdots t_n\rangle$ M''_n. The algorithm will not backtrack from M_h before it has found an M'_n that covers M''_n. If $n < h$, then M'_n is found before backtracking to M_n. Furthermore, $M_n < M'_n$, because M'_0 has ω-symbols in the same places as M_n and in at least one more place. So the algorithm passivates M_n by direct coverage before backtracking to it.

If $n \geq h$, then M_n has ω-symbols in precisely the same places as M'_0. Also M''_n has ω-symbols in precisely the same places, because it was defined using "$[\cdots\rangle$" instead of "$[\cdots\rangle^\omega$". For the remaining places, $M_0(p) = M'_0(p)$, so also $M_n(p) = M''_n(p)$. We conclude that $M''_n = M_n$, implying M'_0 $[t_1 \cdots t_n\rangle$ M_n. □

We prove a similar theorem for most tokens first search.

Theorem 4. *Let the construction order be most tokens first and history merging be applied. Assume that M_0 $[t_1 \cdots t_n\rangle^\omega$ M_n and $M_0 < M'_0$. Assume that all transitions along the path M_0 $[t_1 \cdots t_n\rangle^\omega$ M_n were found before M'_0. After finding M'_0, the algorithm will not fire transitions from M_n, unless M'_0 $[t_1 \cdots t_n\rangle$ M_n.*

Proof. Let $M_1, \ldots, M_{n-1}$ be defined in the obvious way. Let $M \prec M'$ denote that M has fewer ω-places than M', or the same number of ω-places but altogether fewer tokens in the remaining places than M'. Then $M < M'$ implies

$M \prec M'$. Also note that along any path, ω-symbols may be introduced but cannot disappear. Let $A(M)$ denote any M'' such that $M \preceq M''$ and $M'' \in A$.

If $M_n \preceq M_i$ for each $0 \leq i \leq n$, then M_n has ω-symbols in precisely the same places as M_0. Furthermore, $M_n \preceq M_0 \prec M'_0 \preceq A(M'_0)$, so $A(M'_0)[t_1\rangle^\omega M'_1$ is fired before firing transitions from M_n. Because $M_0 < M'_0$ and ω-symbols were not added during $M_0\ [t_1\rangle^\omega\ M_1$, we have $M_n \preceq M_1 < M'_1$. The reasoning continues until we get $M_n < M'_n$. Then M_n is passivated by coverage.

In the opposite case, let i be maximal such that $M_{i-1} \prec M_n$. So $M_n \preceq M_j$ for $i \leq j \leq n$. If any of $M_i, \ldots, M_n$ has been found before $M_{i-1}[t_i\rangle^\omega M_i$ is fired, then all transitions from such an ω-marking and eventually all transitions from M_n are fired before $M_{i-1}[t_i\rangle^\omega M_i$. In that case, the algorithm never fires any transitions from M_n after finding M'_0, simply because it already has fired them all. The same happens if any M that is investigated after firing $M_{i-1}[t_i\rangle^\omega M_i$ but before M'_0 is found has $M \prec M_n$.

The case remains where $M_{i-1} \prec M_n \preceq M_j$ for $i \leq j \leq n$, none of the M_j is found before firing $M_{i-1}[t_i\rangle^\omega M_i$, and (1) from then on every ω-marking M investigated had $M_n \preceq M$ until M'_0 is found.

Because M'_0 is not found before completing the path, the finding history of M'_0 has some $M'_{h-1}\ [t'_h\rangle^\omega\ M'_h$ (where $h \leq 0$) such that M'_{h-1} (but not M'_h) has been found when $M_{i-1}\ [t_i\rangle^\omega\ M_i$ is fired. This implies $M'_{h-1} \preceq M_{i-1} \prec M_n$. By (1), $M'_{h-1}\ [t'_h\rangle^\omega\ M'_h$ cannot be fired after $M_{i-1}\ [t_i\rangle^\omega\ M_i$ until M'_0 is found. The remaining possibility is that $M'_{h-1}\ [t'_h\rangle^\omega\ M'_h$ is the same transition as $M_{i-1}\ [t_i\rangle^\omega\ M_i$. This implies that $M_0\ [t_1 \cdots t_i\rangle^\omega\ M_i\ [t'_{h+1} \cdots t'_0\rangle^\omega\ M'_0$. Add-$\omega$ guarantees that for each $p \in P$, either $M_0(p) = M'_0(p)$ or $M'_0(p) = \omega$.

By $M_n \preceq M_i$, M_n and M_i have ω-symbols in precisely the same places. Therefore, M'_0 has ω-symbols in at least the same places as M_n.

If M'_0 has more ω-symbols than M_n, then the same holds for all M'_i along the path $M'_0\ [t_1 \cdots t_n\rangle\ M'_n$, and we have $M_n \prec M'_i \preceq A(M'_i)$. Therefore, all the transitions corresponding to $A(M'_0)\ [t_1 \cdots t_n\rangle^\omega\ A(M'_n)$ are fired before firing any transition from M_n, and M_n is passivated before being investigated further.

Otherwise, M'_0 and M_n have ω-symbols in precisely the same places. This implies $M'_0\ [t_1 \cdots t_n\rangle\ M_n$, because $M_0(p) = M'_0(p)$ if $M'_0(p) < \omega$. □

6 Conclusion

We have given a simple algorithm for calculating minimal coverability sets. Furthermore, we have given arguments that lead us to believe that published more complicated algorithms are in general no more efficient.

Using examples, we have demonstrated that by tailoring the incoming Petri net in a suitable way, almost any algorithm can be made terminate much quicker for that particular Petri net than its competitors. Therefore, there probably cannot be any theorem that one algorithm is *systematically* better, or even as good as, another. However, we proved two theorems saying that certain versions of the simple non-pruning algorithm automatically have the benefits that pruning tries to achieve. At the same time, the simple algorithm does not run the risk of repeating work on identical ω-markings, like pruning algorithms do. We also

Table 1. Some measurements with test data from [3]

model	$\|A\|$	most tokens f.		depth-first		breadth-first		[6]
fms	24	63	53	110	56	421	139	809
kanban	1	12	12	12	12	12	12	114
mesh2x2	256	479	465	774	455	10733	2977	6241
mesh3x2	6400	11495	11485	8573	10394			
multipoll	220	245	234	244	244	507	507	2004
pncsacover	80	215	246	284	325	7122	5804	1604

pointed out that it may be advantageous to add as many ω-symbols as early as possible, and presented techniques towards such a goal.

Table 1 shows results of the six biggest test runs that we have made with the test set from [3]. The second column shows the size of the minimal coverability set. The other numbers are the total numbers of constructed distinct ω-markings, that is, $|F|$. The running time was always below 0.1 s except with mesh3x2 (30 s, 29 s, 8 s, 9 s) and, in the case of breadth-first search also mesh2x2 (0.7 s, 0.1 s) and pncsacover (0.3 s, 0.3 s). These times should not be compared to those in [3,6], because we used a different computer and programming language (C++).

We ran each experiment with transitions tried in the order that they were given in the input and in the opposite order. As the table shows, this low-level difference had sometimes a dramatic impact on the result. This acts as a warning that numbers like the ones in the table are much less reliable than we would like.

History merging was applied on lines 6 and 8 of Fig. 1. Switching it off had very little effect on $|F|$ except with breadth-first search.

We leave further analysis and measurements as potential future work.

Acknowledgements. We would like to thank Prof. Mikko Tiusanen and the reviewers for their help, and the authors of [6] for providing help for the experiments.

References

1. Finkel, A.: The Minimal Coverability Graph for Petri Nets. In: Rozenberg, G. (ed.) APN 1993. LNCS, vol. 674, pp. 210–243. Springer, Heidelberg (1993)
2. Finkel, A., Geeraerts, G., Raskin, J.-F., Van Begin, L.: A counter-example to the minimal coverability tree algorithm. Technical Report 535, Universite Libre de Bruxelles (2005)
3. Geeraerts, G., Raskin, J.-F., Van Begin, L.: On the efficient computation of the minimal coverability set of Petri nets. International Journal of Foundations of Computer Science 21(2), 135–165 (2010)
4. Karp, R.M., Miller, R.E.: Parallel program schemata. Journal of Computer and System Sciences 3(2), 147–195 (1969)
5. König, B., Koziura, V.: Incremental construction of coverability graphs. Information Processing Letters 103(5), 203–209 (2007)
6. Reynier, P.-A., Servais, F.: Minimal Coverability Set for Petri Nets: Karp and Miller Algorithm with Pruning. In: Kristensen, L.M., Petrucci, L. (eds.) PETRI NETS 2011. LNCS, vol. 6709, pp. 69–88. Springer, Heidelberg (2011)

Stubborn Sets for Simple Linear Time Properties

Andreas Lehmann, Niels Lohmann, and Karsten Wolf

Universität Rostock, Institut für Informatik

Abstract. We call a linear time property *simple* if counterexamples are accepted by a Büchi automaton that has only singleton strongly connected components. This class contains interesting properties such as LTL formulas $\mathbf{G}(\varphi \implies \mathbf{F}\,\psi)$ or $\varphi\,\mathbf{U}\,\psi$ which have not yet received support beyond general LTL preserving approaches.

We contribute a stubborn set approach to simple properties with the following ingredients. First, we decompose the verification problem into finitely many simpler problems that can be independently executed. Second, we propose a stubborn set method for the resulting problems that does neither require cycle detection, nor stuttering invariance, nor existence of transitions that are invisible to *all* atomic propositions. This means that our approach is applicable in cases where traditional approaches fail. Third, we show that sufficient potential is left in existing implementations of the proposed conditions by exploiting all the available nondeterminism in these procedures. We employ a translation to integer linear programming (ILP) for supporting this claim.

1 Introduction

There are two main approaches to the verification of temporal properties, both concerned with the alleviation of the state explosion. In symbolic model checking [1,2], sophisticated data structures represent a set S of states or paths such that the memory consumption does not strongly correlate to the number of elements contained in S. In explicit model checking, the original transition system is replaced by a smaller one that, by construction, is equivalent with respect to the investigated property. This paper is concerned with explicit model checking. Here, the partial order reduction [16,7,11] appears to be the most powerful reduction technique.

There is a broad spectrum of approaches to partial order reduction. On one end of the spectrum, there are approaches that preserve properties of a full temporal logic such as LTL [11,17] or CTL* [6] or a process algebraic semantics [18]. On the other end of the spectrum, singular properties or distinguished classes of properties are preserved such as deadlocks [16], reachability [10] or other standard properties [13]. Between these extremal techniques, there are approaches to smaller but nontrivial fragments of temporal logic [14].

In this paper, we generalize the technique used in [13] for singular properties to a class of properties that can be defined in terms of the structure of a Büchi automaton. Büchi automata are an established tool for specifying linear time temporal properties and are closely related to the temporal logic LTL. We study Büchi automata where all loops are self loops; that is, the automaton has no strongly connected component with more than one element. For such properties, we decompose the verification problem into a finite number of even simpler verification problems. We present a new stubborn set method that preserves the resulting class of properties. The resulting technique complements existing approaches. In difference to traditional LTL preserving methods, it does not require the detection of cycles (at least not for stubborn set calculation), it tolerates a limited amount of stuttering in the investigated property, and it requires fewer transitions to be invisible with respect to the property.

For increasing the power of the required stubborn set calculation, we further study the impact of nondeterministic choices in a particular procedure for stubborn set calculation. To this end, we give a translation of the stubborn set calculation problem to an integer linear programming (ILP) problem that reflects all possible choices. This way, we import the mature heuristics that ILP offers for resolving nondeterminism. Experiments show that this translation pays off, even when the costly ILP procedure is involved.

The paper is organized as follows. We first introduce Petri nets (Sect. 2) and Büchi automata (Sect. 3). Then we introduce our notion of simple properties and discuss consequences (Sect. 4). We continue with a presentation of traditional stubborn set reduction for liner time properties (Sect. 5) and our new approach (Sect. 6). In Sect. 7, we present a translation of stubborn set calculation to integer linear programming. We conclude with a discussion of related work.

2 Petri Nets

For a set M, denote 2^M its power set. For sets M and N, let M^N be the set of functions $f : N \to M$. We use the following notation for Petri nets.

Definition 1 (Petri net). *A Petri net $N = [P, T, F, W, m_0]$ consists of two finite, nonempty, and disjoint sets P (of places) and T (of transitions), a flow relation $F \subseteq (P \times T) \cup (T \times P)$, a multiplicity assignment $W : F \to \mathbb{N} \setminus \{0\}$, and an initial marking $m_0 \in \mathbb{N}^P$.*

Places and transitions are collectively called nodes. For a node x, let $\bullet x = \{y \mid [y, x] \in F\}$ denote its pre-nodes while $x\bullet = \{y \mid [x, y] \in F\}$ denotes its post-nodes.

Definition 2 (Behavior). *Transition $t \in T$ is enabled in marking $m \in \mathbb{N}^P$ iff, for all $p \in \bullet t$, $m(p) \geq W(p, t)$. We denote this by $m \xrightarrow{t}$. If t is enabled, t can fire in m, yielding marking m' where, for all $p \in P$, $m'(p) = m(p) - W(p, t) + W(t, p)$ under the assumption that $W(x, y)$ is set to 0 for $[x, y] \notin F$. This relation is denoted by $m \xrightarrow{t} m'$.*

We investigate properties that are evaluated in marking sequences of N.

Definition 3 (Marking sequence). *A finite or infinite sequence of markings $m_0m_1m_2\ldots$ ($m_i \in \mathbb{N}^P$) is a marking sequence of the Petri net N iff m_0 is the initial marking of N and, for all i, either there are no enabled transitions in m_i and $m_{i+1} = m_i$, or there is a transition t_i such that $m_i \xrightarrow{t_i} m_{i+1}$. We say that the corresponding transition sequence $t_0t_1\ldots$ produces the marking sequence $m_0m_1m_2\ldots$. We denote executability of a transition sequence starting at m by $m \xrightarrow{t_it_{i+1}\cdots}$ or, if the reached marking is relevant, by $m \xrightarrow{t_it_{i+1}\cdots} m'$.*

3 Model Checking with Büchi Automata

In this paper, we consider *linear time* properties. A linear time property is one that can be evaluated in each single (typically infinite) run of the system in isolation. Such a property is true of the given system if all runs of the system satisfy it. A popular way to verify linear time properties is to construct a Büchi automaton that accepts all those infinite sequences which violate the property. Using standard product automaton construction, a new Büchi automaton is constructed that accepts all runs which are possible in the system *and* violate the property. If the given property is satisfied in the system, the resulting automaton accepts the empty language, otherwise any accepted run yields a counterexample.

Properties are built upon *propositions*. We postulate a countable set of *atomic* propositions which can combined using Boolean operators.

Definition 4 (Proposition). *Let $\mathcal{A}$ be some countable set, elements of which are called* atomic proposition. *Atomic propositions are* propositions. *If φ and ψ are propositions, so are $(\varphi \wedge \psi)$ and $(\varphi \vee \psi)$. Let $\mathcal{P}_{\mathcal{A}}$ the set of all propositions over $\mathcal{A}$.*

Subsequently, we omit parentheses relying on the usual precedence of the $\wedge$ and $\vee$ operators. We assume that $\mathcal{A}$ is arbitrary but fixed throughout the paper. In examples, we shall use atomic propositions of the form $p \geq k$ and $p < k$ where p is a place of the investigated Petri net and k is some natural number. Propositions are evaluated in markings. For the atomic propositions, we assume that their interpretation is somehow given by a relation $\models \subseteq \mathbb{N}^P \times \mathcal{A}$. In our example set of atomic propositions, let $m \models p \geq k$ iff $m(p) \geq k$ and $m \models p < k$ iff $m(p) < k$. Relation $\models$ is canonically lifted to arbitrary propositions by stating $m \models (\varphi \wedge \psi)$ ($m \models (\varphi \vee \psi)$) iff $m \models \varphi$ and (or) $m \models \psi$.

We call $\mathcal{A}$ *closed under negation* if, for every $a \in \mathcal{A}$, there is a $b \in \mathcal{A}$ such that, for all m, $m \not\models a$ if and only if $m \models b$. Throughout the paper, we assume $\mathcal{A}$ to be closed under negation which justifies the absense of negation in Def. 4: negation can easily be removed using de Morgan's laws. Our example set of atomic propositions is obviously closed under negation. We furthermore assume that there is a proposition *tt* that is true of all markings and a proposition *ff* that is false of all markings. In our examples, *tt* $:= p \geq 0$ and *ff* $:= p < 0$ satisfy these requirements.

We define Büchi automata such that they are directly controlled by propositions. For defining accepting runs of the automaton, we need to observe that

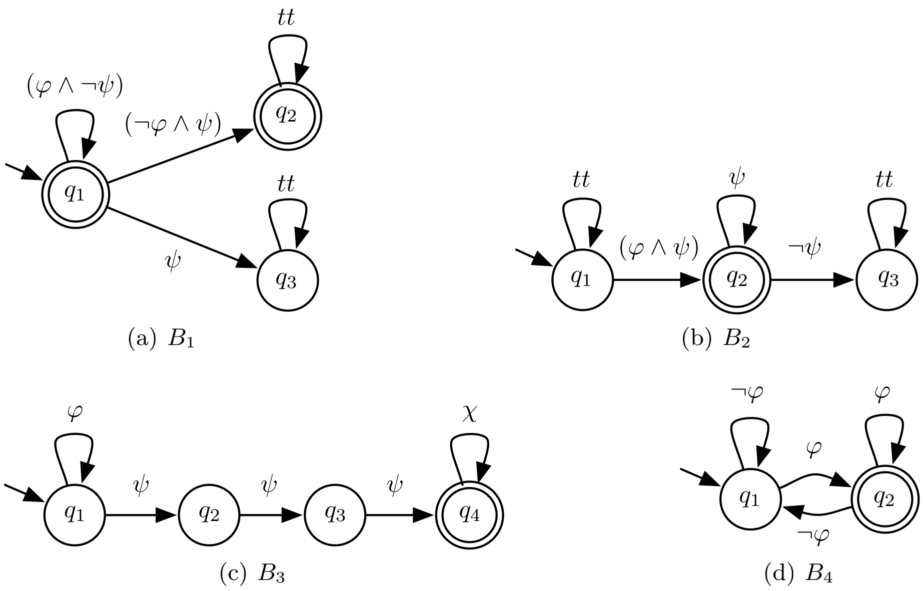


Fig. 1. Examples of Büchi automata

the Büchi automaton attaches propositions to transitions while, in the system, propositions are related to markings.

Definition 5 (Büchi automaton). *A Büchi automaton $B = [Q, q_0, \delta, \lambda, F]$ consists of*

- *a finite set Q of states;*
- *an initial state $q_0 \in Q$;*
- *a transition relation $\delta \subseteq Q \times Q$;*
- *a labeling function $\lambda : \delta \to \mathcal{P}_{\mathcal{A}}$;*
- *a set $F \subseteq Q$ of finite states.*

B accepts an infinite sequence of markings $m_1 m_2 m_3 \ldots$ if and only if there is an infinite sequence $q_0 q_1 q_2 q_3 \ldots$ such that

- *for all i, $[q_i, q_{i+1}] \in \delta$;*
- *for all $i > 0$, $m_i \models \lambda([q_{i-1}, q_i])$;*
- *for infinitely many i, $q_i \in F$.*

Figure 1 shows a few examples of Büchi automata.

The first state of a state sequence is of course the designated initial state of B. Acceptance of a Büchi automaton as defined above is in fact nondeterministic. For some state q and marking m, there may exist several q_1, q_2 such that $q_1 \neq q_2$, $m \models \lambda([q, q_1])$, and $m \models \lambda([q, q_2])$.

The product system $N \cap B$ of a Petri net N and a Büchi automaton B is a Büchi automaton again.

Definition 6 (Product System). *Let $N = [P, T, F, W, m_0]$ be a Petri net and $B = [Q, q_0, \delta, \lambda, F]$ be a Büchi automaton. The product system $N \cap B$ is a Büchi automaton $B^* = [Q^*, q_0^*, \delta^*, \lambda^*, F^*]$ where*

- *q_0^* is some element not contained in $\mathbb{N}^P \times Q$*
- *$Q^* = \{q_0^*\} \cup (\mathbb{N}^P \times Q)$;*
- *$[q_0^*, [m, q]] \in \delta^*$ iff $[q_0, q] \in \delta$, $m_0 \models \lambda([q_0, q])$, and $m = m_0$;*
- *$[[m, q], [m', q']] \in \delta^*$ iff $m \to m'$, $[q, q'] \in \delta$, $m' \models \lambda([q, q'])$;*
- *for all $[q^*, q^{*\prime}] \in \delta^*$, $\lambda([q^*, q^{*\prime}]) = tt$;*
- *$[m, q] \in F^*$ iff $q \in F$.*

The resulting Büchi automaton labels all transition with tt. That is, the actual input sequence is irrelevant and the product system can be seen as a closed system. Nevertheless, standard theory asserts:

Proposition 1 (Emptiness). *There is an infinite sequence of markings realizable in N and accepted by B if and only if the product system $N \cap B$ has an accepting run.*

For bounded Petri nets, the product system has an accepting run if and only if it has a strongly connected component (w.r.t. δ) that is reachable from the initial state and contains a final state.

There is a close link between Büchi automata and the linear time temporal logic LTL. In our setting, LTL formulas are interpreted on infinite sequences of markings. Sloppily introduced, LTL comprises propositions (true of a sequence if satisfied in the first marking), Boolean operations (with the usual meaning), and temporal operators **X** ("holds in the next state"), **F** ("holds eventually"), **G** ("holds invariantly"), and **U** ("one property holds until another one is satisfied"). For LTL, we know that:

Proposition 2 (LTL). *For every LTL formula φ, there is a Büchi automaton that accepts exactly those sequences of markings which violate φ.*

Büchi automaton B_1 (Fig. 1) accepts all sequences that violate the LTL property $\varphi \, \mathbf{U} \, \psi$. B_2 accepts all sequences that violate $\mathbf{G}(\varphi \implies \mathbf{F} \, \psi)$.

In this paper, all results rely on Büchi automata as such. We use LTL only for the purpose of linking our results to related work. For this reason, we skip a formal definition of syntax and semantics of LTL.

4 Simple Büchi Automata

For a binary relation R, let R^* denote its reflexive and transitive closure.

Definition 7 (Simple Property). *A Büchi automaton is* simple *iff, for all $q, q' \in Q$, $[q, q'] \in \delta^*$ and $[q', q] \in \delta^*$ implies $q = q'$. A linear time property is simple iff the set of infinite runs violating the property is accepted by a simple Büchi automaton.*

In a simple Büchi automaton, the only occurring cycles in the transition relation are self-loops at states. In other words, all strongly connected components reachable from the initial state are singletons. In Fig. 1, automata B_1, B_2, and B_3 are simple whereas B_4 is not.

The class of simple properties contains several relevant properties that are expressible in LTL. Among them, there is the formula $\mathbf{G}(\varphi \implies \mathbf{F}\,\neg\psi)$ (for arbitrary propositions φ and ψ, see B_2 in Fig. 1) which has been shown to be central in the verification of distributed algorithms in [12]. $\mathbf{F}\,\varphi$, $\mathbf{G}\,\varphi$, and $\varphi\,\mathbf{U}\,\psi$ are other examples of simple LTL properties. In contrast, $\mathbf{FG}\,\varphi$ is not simple. Marking sequences that violate $\mathbf{FG}\,\varphi$ contain infinitely many markings where φ is not satisfied. For these sequences, the set of states of the Büchi automaton that are entered infinitely often must contain both a final and a non-final state since we need to leave the final state when φ is violated and to enter it when φ is satisfied. In this case, however, these two states would form a cycle contradicting simplicity. Automaton B_4 in Fig. 1 accepts all sequences that violate $\mathbf{FG}\,\neg\varphi$.

As self loops (states q with $[q, q] \in \delta$) are the only cycles in simple Büchi automata, accepting runs have only one state that occurs infinitely often. This must obviously be a final state. In addition, there is no state re-entered after having been left. This can be exploited for replacing simple Büchi automata by even simpler automata.

Definition 8 (Elementary Büchi automaton). *A simple Büchi automaton is elementary iff*

- *for each state q there is at most one state q' such that $[q, q'] \in \delta$ and $q \neq q'$;*
- *if $q \in F$ then $[q, q'] \in \delta$ implies $q = q'$.*

That is, if unreachable states are removed, an elementary Büchi automaton consists of a nonbranching sequence of states which may or may not have self loops and the last of which is a final state.

It is not difficult to see

Lemma 1. *For each simple Büchi automaton B, there is a finite set of elementary Büchi automata $\mathcal{M}$ such that a marking sequence is accepted by B if and only if it is accepted by one automaton in $\mathcal{M}$.*

$\mathcal{M}$ is obtained by enumerating all self-loop free paths in B that end in a final state of B. By the simplicity property, there exist only finitely many such paths. For each path, its set of states together with transitions between subsequent states and self loops (as far as present in B) form one automaton to be included in $\mathcal{M}$. Only the last state in the path is final in the resulting automaton.

Figure 3 shows the two elementary Büchi automata that are the result of decomposing B_1 in Fig. 1. Decomposition of B_2 in the same figure yields the automaton depicted in Fig: 3.

If a non-elementary simple Büchi automaton B is decomposed into a set of elementary Büchi automata $\mathcal{M}$, it is clear that each element $B^* \in \mathcal{M}$ has a simpler structure than B. In particular, its structure is embedded in B. That is, we can immediately observe

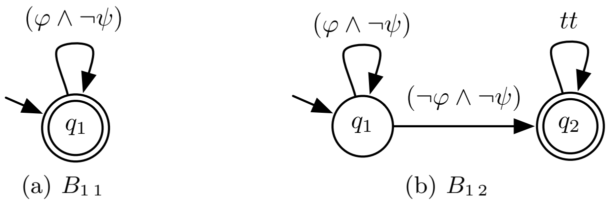

$(\varphi \wedge \neg\psi)$
q_1
(a) $B_{1\,1}$
$(\varphi \wedge \neg\psi)$
tt
$(\neg\varphi \wedge \neg\psi)$
q_1
q_2
(b) $B_{1\,2}$

Fig. 2. Decomposition of B_1 (Fig. 1) into elementary automata

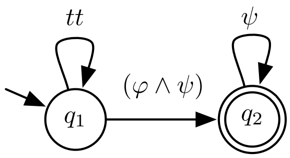


Fig. 3. Decomposition of B_2 (Fig. 1) into elementary automata

Corollary 1. *Let N be a Petri net, B a simple Büchi automaton, and $\mathcal{M}$ its decomposition into elementary Büchi automaton. For every $B^* \in \mathcal{M}$, the number of states of $N \cap B^*$ is less than or equal to the number of states of $N \cap B$.*

Of course, the decomposition of B may contain a large number of elementary automata (in fact, up to exponentially many). Moreover, as some of the resulting automata share common prefixes, the accumulated run time for the product systems $N \cap B^*$ for all $B^* \in \mathcal{M}$ may take more time than producing $N \cap B$. However, the limiting resource for model checking is still the available memory and a large number of state spaces each of which fit into memory is more valuable than a single state space that does not.

Another advantage of decomposition is that each resulting automaton B^* operates only on some of the proposition that occur in B. This means that more transitions of N become invisible which is favorable for traditional LTL preserving stubborn set methods (see next section).

Using this argument, the remainder of this paper is concerned with elementary Büchi automata only. Hence, we conclude this section with the introduction of some terminology. Without loss of generality, we represent an elementary Büchi automaton using a set $\{q_0, \ldots, q_n\}$ such that q_0 is the initial state and $\delta = \{[q_i, q_{i+1}] \mid i < n\} \cup \{[q_i, q_i] \mid i \leq n\}$. Consequently, $F = \{q_n\}$. Every elementary automaton can be expressed this way by renaming its states. In particular, missing self loops can be added by labeling them with formula *ff*.

For state q_i, we call $\varphi_i := \lambda([q_i, q_{i+1}])$ the *progressing* formula at q_i as its satisfaction is necessary for leaving q_i and getting closer to the final state. Likewise, call $\psi_i := \lambda([q_i, q_i])$ the *retarding* formula at q_i.

5 Traditional LTL Preserving Stubborn Sets

For self-containedness, we sketch the core concepts of the classical LTL preserving stubborn set technique. Stubborn set reduction aims at replacing a state space with a smaller one that is equivalent to the original one with respect to the original one. For linear time properties, equivalence means that the reduced state space has an accepting run if and only if the original does. Reduction is obtained by considering, in each state only some transitions. The set of transitions to be considered is controlled by a function $\mathit{stub} : \mathbb{N}^P \to 2^T$. In each marking m, only transitions in $\mathit{stub}(m)$ are considered. $\mathit{stub}(m)$ is called the stubborn set in m. For a given function stub, the reduced state space is obtained by replacing the condition $m \to m'$ in Def. 6 with the condition *there is a transition t in $\mathit{stub}(m)$ such that $m \xrightarrow{t} m'$*. While only enabled transitions in $\mathit{stub}(m)$ are considered for producing successor states, it is convenient to permit disabled transitions as members of $\mathit{stub}(m)$, too.

Equivalence of original and reduced transition system can only be established if certain conditions are met by the stubborn set generator stub. For preserving a linear time property, $\mathit{stub}(m)$ must contain an enabled transition if there is any in m and meet the requirements of *commutation*, *invisibility*, and *non-ignorance* each of which is discussed subsequently. Furthermore, it is required that the investigated property is *stuttering invariant*. A linear time property is stuttering invariant iff, for all finite marking sequences π_1, all markings m, and all infinite marking sequences π_2, the property is satisfied in the path $\pi_1 m \pi_2$ if and only if it is satisfied in the path $\pi_1 m m \pi_2$. For an LTL formula, absence of the $\mathbf{X}$ operator is sufficient but not necessary for stuttering invariance. For instance, $\mathbf{F}(\varphi \wedge \mathbf{X}\,\psi)$ is stuttering invariant if ψ is the negation of φ.

Properties represented by B_1, B_2, and B_4 in Fig. 1 are stuttering invariant, whereas the one represented by B_3 is not.

5.1 Commutation

The requirement of commutation is the heart of any stubborn set method.

Definition 9 (Commutation). *A stubborn set generator stub satisfies the commutation requirement at marking m iff, for at least one $t^* \in \mathit{stub}(m)$, all sequences w of transitions outside $\mathit{stub}(m)$ and all $t \in \mathit{stub}(m)$,*

- *$m \xrightarrow{wt} m'$ implies $m \xrightarrow{tw} m'$;*
- *m does not enable any transition or $m \xrightarrow{t^*}$ and $m \xrightarrow{w}$ implies $m \xrightarrow{wt^*}$.*

For preserving linear time properties, the traditional stubborn set method requires that the commutation requirement is satisfied in all markings.

For every marking m and every set $U \subseteq T$ of transitions, there exists a set U' of transitions such that $U \subseteq U'$ and U' meets the commutation requirement at m. This statement is trivial as $U' := T$ always satisfies the commutation requirement. It is nevertheless useful to postulate the existence of a function *closure* : $\mathbb{N}^P \times 2^T \to 2^T$ such that, for al markings m and all sets U of transitions, $closure(m, U)$ is a — desirably small — set of transitions that includes U and meets the commutation requirement at m.

Several procedures for implementing *closure* have been discussed in the literature [20,18]. A very simple procedure (implemented in our tool LoLA) transitively includes transitions according to the following requirements: (E) if t is contained and enabled in m, then $(\bullet t)\bullet$ must be included, too, and (D) if t is included and disabled in m then $\bullet p$ must be included, too, for some insufficiently marked place p (i.e., $m(p) < W([p, t])$).

5.2 Invisibility

Invisibility is the only requirement in the traditional LTL preserving stubborn set method that depends on the verified property. For space reasons, we present a rather coarse version of the requirement. A finer version of the requirement can be found in [17].

Definition 10 (Invisibility). *Transition t is invisible for proposition φ iff, for all $m, m' \in \mathbb{N}^P$ with $m \xrightarrow{t} m'$, $m \models \varphi$ if and only if $m' \models \varphi$. A stubborn set generator obeys the invisibility requirement if, for all markings m, it either returns all transitions or only transitions that, if enabled, are invisible to all propositions occurring in the involved Büchi automaton.*

The purpose of the invisibility requirement is to make sure that the sequences wt and tw considered for the commutation requirement produce marking sequences that differ only by stuttering.

5.3 Non-ignorance

Preservation of a linear time property can be proved by gradually transforming an accepting run of the original system into an accepting run of the reduced system. The main tool for transformation is the commutation requirement, as follows. A path $\pi_1 w t \pi_2$ of the original system where π_1 is executable in the reduced system, $m_0 \xrightarrow{\pi_1} m$, w being a sequence of transitions not using transitions in $stub(m)$, and $t \in stub(m)$, is transformed into $\pi_1 t w \pi_2$ which, by the commutation requirement, is equally well executable in the original. Moreover, as $t \in stub(m)$, at least $\pi_1 t$ is now executable in the reduced system as well. Repeated application of this argument produces the accepting run in the reduced system.

The invisibility property ensures that $\pi_1 w t \pi_2$ and $\pi_1 t w \pi_2$ produce marking sequences that differ only by stuttering. The same is true for any finite number of modifications of the path. Unfortunately, the argument cannot be continued to an infinite number of transformations. Assume that m does not satisfy φ but the occurrence of w makes φ hold. That is, the original path satisfies $\mathbf{F}\,\varphi$. If, however, an infinite number of invisible transitions is shifted in front of w, the resulting path is an infinite sequence of markings none of which satisfies φ, so the resulting path does not satisfy $\mathbf{F}\,\varphi$. The non-ignorance requirement repairs this problem.

Definition 11 (Non-ignorance). *A stubborn set generator stub obeys the non-ignorance requirement for a Petri net N if, in every cycle of the reduced product system, there is at least one marking where $stub(m) = T$.*

By this condition, only finitely many transformations of the original path are necessary before the first transition of w can be shown to be executable in the reduced system as well. So, all (especially all visible) transitions of the original path can be executed in the reduced system thus ensuring the desired preservation of the verified property.

Non-ignorance is typically implemented by starting with a stubborn set generator *stub* that does not care about ignorance. Whenever a cycle is detected in the reduced state space, $stub(m)$ is augmented to the whole T. In order to safely detect all cycles, the generation of the reduced system is usually done according to the depth first strategy. Here, every cycle contains a state that has a successor on the depth first search stack which is a very simple procedure for cycle detection. This approach is, however, quite problematic especially for distributed state space generation where a strict depth first order blocks the parallel exploration of states. In addition, it is widely believed that the non-ignorance requirement is to a large extent responsible for unsatisfactory reduction results.

6 Our Approach to Stubborn Sets for Simple Properties

In our approach, we would like to use the stubborn sets for navigating the generation of the product system through the Büchi automaton. By the results of Sect. 4, we only need to consider elementary Büchi automata. For being able to tinvolve the current state of the Büchi automaton, we modify the signature of the stubborn set generator. Instead of $stub : \mathbb{N}^P \to 2^T$, we use $stub : \mathbb{N}^P \times Q \to 2^T$. In a non-final Büchi state q_i, two formulas are important. First, there is the (unique) progressing formula φ_i. Its satisfaction can take us to the next Büchi state and hence closer to acceptance. Second, there is the retarding formula ψ_i. This formula must not be violated until the progressing formula becomes true. Otherwise, the Büchi automaton would not accept either. In the unique final state of the Büchi automaton, our goal is to stay there forever. This state has only a retarding formula to be considered. Consequently, we have two objectives in stubborn set generation: we would like to "switch on" the progressing formula while, at the same time, avoid "switching off" the retarding formula. In the sequel, we first study these two objectives in isolation. Then we combine them to our stubborn set approach for elementary Büchi automata, and finally we compare the approach to the traditional LTL preserving method.

6.1 Stubborn Sets for Switching on Formulas

For this part, we import the ideas from [13] and present them just for being self-contained. The main concept in his subsection are up-sets.

Definition 12 (up-set). *Consider a marking m and a proposition φ such that $m \not\models \varphi$. A set U of transitions is an up-set for m and φ if every transition sequence w such that $m \xrightarrow{w} m'$ and $m' \models \varphi$ contains at least one element of U.*

In other words, φ remains violated as long as only transitions in $T \setminus U$ are fired. Let $UP(m, \varphi)$ the family of all up-sets for m and φ and observe that $UP(m, \varphi)$ is not defined if $m \models \varphi$.

For our stubborn set approach, we are interested in small up-sets for given m and φ. Calculation of up-sets can typically be done by just considering m, the structure of φ, and the topology of N. A reasonable up-set for atomic proposition $p \geq k$ is $\bullet p$ while $p\bullet$ can be used for $p < k$. If $m \not\models \varphi$ any up-set of φ is an up-set for $\varphi \wedge \psi$. Symmetrically, if $m \not\models \psi$, any up-set for ψ is an up-set for $\varphi \wedge \psi$ as well. In cases where neither φ nor ψ are satisfied, we can actually choose between an up-set for φ and an up-set for ψ when trying to find one for $\varphi \wedge \psi$. For $\varphi \vee \psi$, an up-set can be constructed as the union of some up-set for φ and an up-set for ψ. Both $UP(m, \varphi)$ and $UP(m, \psi)$ are defined since violation of $\varphi \vee \psi$ at m implies violation of both φ and ψ at m.

The stubborn set approach of [13] just relies on up-sets and the commutation requirement. The following results is concerned with the state space of a Petri net rather than a product system.

Lemma 2 ([13]). *Let N be a Petri net and φ a proposition. Assume that, at every marking m where $m \not\models \varphi$, stub(m) includes U for some $U \in UP(m, \varphi)$ and satisfies the commutation requirement. Then the full state space contains some marking that satisfies φ if and only if the reduced system does.*

If the original state space does not reach a marking that satisfies φ, then such a state cannot be reachable in the reduced state space as we never fire disabled transitions. Assume that the full state space contains a marking m^* satisfying φ but the reduced state space does not. Let $\pi_1\pi_2$ be a transition sequence that is executable in the full state space such that $m_0 \xrightarrow{\pi_1\pi_2} m^*$, π_1 can be executed in the reduced system, and the length of π_2 is minimal. The minimum exists since some m satisfying φ is reachable in the full state space and m_0 is part of the reduced state space (so π_1 is in worst case the empty sequence). Let m be the marking where $m_0 \xrightarrow{\pi_1} m$, so m occurs in the reduced system. By our assumptions, $m \not\models \varphi$, so *stub*(m) includes an up-set for m and φ. One element of this up-set must occur in π_2 since otherwise φ would be violated in m^*. That is, π_2 contains elements of *stub*(m) and can thus be separated into $\pi_2 = w_1tw_2$ where w_1 contains only transitions in $T \setminus$ *stub*(m) and $t \in$ *stub*(m). By the commutation requirement, tw_1w_2 can be executed in m as well and still reaches m^*. However, this means that we have $\pi_1' = \pi_1 t$ and $\pi_2' = w_1w_2$ such that $m_0 \xrightarrow{\pi_1'\pi_2'} m^*$, π_1' can be executed in the reduced system, and the length of π_2' is shorter than the length of π_2. This contradicts the required minimality for the length of π_2.

Stubborn set calculation using up-sets is implemented in the Petri net based verification tool LoLA [21]. In case studies [4,15] and a model checking contest [9], the technique turned out to perform very well. In nets where a marking satisfying φ is indeed reachable, LoLA often computes only little more than the path from m_0 to a target marking. The computed path tends to be only little longer than the shortest possible one. If a marking that satisfies φ is not reachable, the stubborn set method proposed in [10] produces smaller state spaces but does not have such a strong goal orientation on satisfiable instances.

6.2 Preventing Formulas from Being Switched Off

For preserving the value of a retarding formula, we propose a stubborn set that contains only invisible enabled transitions. In difference to the traditional stubborn set technique for LTL, invisibility refers only to the retarding formula at the current state (and not for all propositions). Another difference is that we completely drop the non-ignorance requirement. For non-final Büchi states, the up-sets replace non-ignorance. For the final Büchi state, infinite stuttering is rather welcome, so the absence of a non-stuttering requirement does not harm.

6.3 Stubborn Sets for Elementary Büchi Automata

Let B be an elementary Büchi automaton and N a Petri net. We aim at constructing a reduced product system that accepts some path if and only if $N \cap B$ does. Besides the ideas presented above, we only need two more ingredients. First, there are states where the above ideas cannot be applied. In these situations, the fallback solution is always to include *all* transitions in the stubborn set. Second, we would like to encapsulate the up-set based arguments such that the considered portion of a path does not change the Büchi state. To do that, we will require an up-set that does not contain enabled transitions. This way, the definition of up-sets assures that the progressing formula remains false by firing any transition in the stubborn set. Hence, we stay in the same Büchi state.

Definition 13 (Stubborn set for elementary Büchi automata). *Let B be an elementary Büchi automaton, q_i a state with progressing formula φ_i and retarding formula ψ_i, and m a marking of Petri net N. $stub(m, q_i)$ is any set of transitions of N that satisfies the commutation requirement, the invisibility requirement with respect to ψ_i and:*

- *If q_i is the final state of B then $stub(m, q_i)$ contains at least one enabled transition if m has one;*
- *If q_i is a non-final state of B then either $m \not\models \varphi_i$ and $stub(m, q_i)$ contains an up-set U for m and the progressing formula φ_i such that U does not contain any transition enabled at m, or $stub(m, q_i) = T$.*

This definition brings us immediately to the main theorem of this paper.

Theorem 1 (Preservation of linear time properties). *Let B be an elementary Büchi automaton and N a Petri net. Assume that a reduced state space is constructed using stubborn set $stub(m, q)$ in state $[m, q]$. Assume further that stub meets the conditions of Def. 13. Then $N \cap B$ accepts some path if and only if the reduced product system does.*

Proof. As the reduced system is a subsystem of the full one, acceptance in the reduced system trivially implies acceptance of the same path in the full system. Assume that the full product system accepts some marking sequence while the reduced system does not accept any marking sequence. Let π be an accepting run in $N \cap B$, that is, π is a sequence of elements in $\mathbb{N}^P \times Q$. Separate π into the finite sequence π_1 and the infinite sequence π_2 such that $\pi = \pi_1\pi_2$ and π_1 is the largest prefix of π that is also executable in the reduced system. Assume without loss of generality that π is selected such that the number of elements in π_2 with non-final Büchi states is minimal. Let $[m, q]$ be the last element of π_1. As π_1 is the largest prefix executable in the reduced state space, $\mathit{stub}(m, q)$ does not contain all enabled transitions. We consider two cases.

In the first case, assume that q is the final state of the Büchi automaton. Then, by construction of *stub*, $\mathit{stub}(m, q)$ contains at least one enabled transition if m has one, all enabled transitions in $\mathit{stub}(m, q)$ are invisible with respect to the retarding formula ψ at q, and the commutation requirement is satisfied. Let $t_1t_2\ldots$ be the transition sequence that produces the infinite marking sequence $[m, q]\pi_2$ (i.e. t_1 is fired in m). We separate two sub-cases. In case 1.1, there is a i such that $t_j \in \mathit{stub}(m, q_i)$, then let j be the smallest such i. This means that $t_1 \ldots t_{j-1}$ consists of transitions in $T \setminus \mathit{stub}(m, q)$. By the commutation requirement, sequence $t_jt_1t_2 \ldots t_{j-1}t_{j+1}t_{j+2} \ldots$ is executable at m. As t_j is invisible for ψ, the marking sequence produced by the modified transition sequence is an accepting one in the full product system but has a larger prefix executable in the reduced system. In case 1.2, the whole sequence $t_1t_2\ldots$ consists only of transitions not in $\mathit{stub}(m)$. Then either m does not have enabled transitions in which case acceptance is trivial, or $\mathit{stub}(m)$ contains a transition t^* that satisfies the second condition of the commutation requirement. For this t^*, combining the two conditions of the commutation requirement, we have that $t^*t_1t_2\ldots$ can be executed at m. The produced marking sequence is accepted in the full product system since the invisibility condition applies to t^*. Again, a larger prefix is executable in the reduced system. As we do not leave the final Büchi state, the arguments of the first case can be repeated, finally yielding an infinite accepting run in the reduced system.

In the second case, q is a non-final state of the Büchi automaton. Separate π_2 into $\pi_2 = \pi_{2,1}\pi_{2,2}$ such that all elements of $\pi_{2,1}$ have q as their Büchi state while no element of $\pi_{2,2}$ has q as its Büchi state. This separation is possible since we consider an elementary Büchi automaton. Let $[m^*, q^*]$ be the first element of $\pi_{2,2}$. Let $t_1 \ldots t_n$ be the marking sequence that produces the markings occurring in $[m, q]\pi_{2,1}[m^*, q^*]$. In particular, we have $m \xrightarrow{t_1\ldots t_n} m^*$. Let φ be the progressing formula at q and ψ the retarding formula at q. Since $\mathit{stub}(m, q) \neq T$, we have that $m \not\models \varphi$ and there is an up-set U for m and φ that is contained in $\mathit{stub}(m, q)$ and does not contain transitions enabled at m. On the other hand, since $[m^*, q^*]$ is the first state in $\pi_{2,2}$, we have that $q^* \neq q$, so $m^* \models \varphi$. By the property of up-sets, some element of U occurs in $t_1 \ldots t_n$. Since U is included in $\mathit{stub}(m, q)$, we can rephrase this statement into: some element of $\mathit{stub}(m, q)$ occurs in $t_1 \ldots t_n$. Let i be the smallest index such that $t_i \in \mathit{stub}(m, q)$. Let m' be the marking where $m \xrightarrow{t_1 \ldots t_i} m'$. By the commutation requirement, t_i is enabled in m, so $t_i \notin U$. Hence, $m' \not\models \varphi$ and, in particular, $m' \neq m^*$. Arguing once more with the commutation requirement, we have $m \xrightarrow{t_i t_1 t_2 \ldots t_{i-1}} m'$. Since none of the involved markings satisfies φ and the invisibility condition for ψ applies to t_i, this sequence produces a run in the full product system that ends in state $[m', q]$. From there, we can continue as in the original path thus having exhibited another accepting run but with smaller distance to the final Büchi state. This contradicts an initial assumption on the choice of π.

6.4 Comparison

Our stubborn set approach differs in various regards from the traditional LTL preserving method.

First, our stubborn sets can be determined purely from N, m, q_i, φ_i, and ψ_i. In particular, we do not need to check for cycles in the reduced state space (except for checking the Büchi acceptance condition). This simplifies reduced state space generation with state space exploration techniques other than depth-first order. Hence, our approach permits flexibility in the search strategy for a accepted run and enables distributed state space generation. We have completely dropped the non-ignorance requirement.

Second, we require invisibility only for retarding formulas, and only one at a time. For progressing formulas not a single invisible transition is formally required. That is, we can expect reduction for properties where progress towards the accepting state depends on non-local events. For the retarding formulas, consideration of one at a time increases the odds that a stubborn set containing only invisible enabled transitions can be found. On the other hand, we require to have a complete up-set included in the stubborn set which may decrease the odds of finding a stubborn set with invisible enabled transitions only. Experimentation beyond this paper will be necessary for finding out whether the overall effect is positive or negative.

Third, we would like to emphasize that we did not require stuttering invariance of the verified property. Stuttering invariance can be violated if some Büchi state q has *ff* as its retarding formula. Our stubborn set method will yield $stub(m, q) = T$. However, if other states do have retarding formulas different from *ff*, the reduced product system can still be smaller than the full one. If we verify using automaton B_3 in Fig. 1, reduction can at least be obtained for states of the product system where the Büchi state is q_1 or q_4.

7 Calculation of Stubborn Sets Using ILP Solving

In the previous section, we learned that the main thread for the size of the stubborn set is fact that two requirements may interfere: the inclusion of an up-set and the invisibility condition. A bad up-set may inevitably cause the insertion of a visible transition while another up-set may not cause this problem. Fortunately, for a fixed marking m and a fixed proposition φ, there may be several different up-sets. An existing implementation of the up-set approach to simple reachability queries in the tool LoLA does not compute and compare up-sets — it uses just an arbitrary one. In connection with the invisibility condition which is not required for reachability, we feel that it is necessary to optimize the choice of an up-set. When computing stubborn sets in LoLA, there is another source of nondeterminism that is not fully exploited yet. In establishing the commutation requirement (in particular, condition (D) in Sect. 5.1), a disabled transition may have several insufficiently marked pre-places and we need to consider only one of them.

Consequently, we would like to study the potential gain that can be obtained by considering all instead of just one up-set and by considering all instead of an arbitrarily fixed insufficiently marked pre-place. To avoid too bad run-time penalties, we import a mature mechanism for managing the huge amount of resulting nondeterminism. To this end, we translate the stubborn set calculation used in LoLA into an integer linear programming (or ILP) problem. The LoLA procedure starts with an arbitrary up-set. For each enabled transition, it includes the conflicting ones and for each disabled transition, it includes the pre-transitions of some insufficiently marked pre-place. If, during this procedure, a visible transition becomes prt of the stubborn set, LoLA returns the set of all enabled transitions.

Translation to ILP brings us to the question of the optimality criterion for stubborn sets. Optimality of stubborn sets has been studied in depth [20,19]. Two results are relevant here. First, if there are two stubborn sets U and V such that $U \subseteq V$ then U is always the better choice; that is, yields better reduction. If there are two stubborn sets U and V such that $card(U) < card(V)$ then U may or may not be the better choice, that is, replacing V with U may decrease or increase the size of the overall state space. The reason is that transitions in U may generate successor markings where only poor reduction is obtained while transitions in V may avoid these states. However, the results only state that U *may* be not the best choice. It does not state that it never *is* the best choice. For this reason, we decided to use a target function of the ILP problem where the number of enabled transitions in the stubborn set is minimized.

For our ILP problem, we include three sets of integer variables. The first set is T. We shall construct the inequations such that a value of 1 assigned to t in the solution means that t is an element the stubborn set while a value of 0 means that t is not an element of the stubborn set. The second set of variables is P. A value of 1 assigned to p means that p is the selected insufficiently marked pre-place of some disabled transition in the closure operation sketched in Sect. 5.1. The third set of variables consists of one variable for each up-set we want to consider.

We continue with presenting the actual translation of the stubborn set calculation into an ILP problem under the assumption that a family of up-sets to be considered is given. We continue with an enumeration of reasonable up-sets and conclude with experimental results.

7.1 Specification of the ILP Problem

For stubborn set calculation in the case of a non-final state, the inputs are a net N, a marking m, a family of up-sets $\mathcal{M}$, a progressing formula φ and a retarding formula ψ. The target function of or ILP problem states that we want to minimize the number of enabled transition in the set.

$$\sum_{t \in T \mid m \xrightarrow{t}} t = MIN.$$

For all variables, we want them to have integer values between 0 and 1.

$$\forall t \in T : 0 \leq t \leq 1 \quad \forall p \in P : 0 \leq p \leq 1 \quad \forall M \in \mathcal{M} : 0 \leq M \leq 1$$

Next, we want to have one of the up-sets included in the stubborn set. In the sequel, we shall use the inequation $y + 1 - x > 0$ for modeling the Boolean relation $x \implies y$.

$$\sum_{M \in \mathcal{M}} M = 1 \quad \forall M \forall t \in M : t + 1 - M > 0$$

The invisibility condition can be stated as follows:

$$\sum_{t : t \text{ visible to } \varphi,\ m \xrightarrow{t}} t = 0$$

The commutation requirement for enabled transitions can be coded in the following way.

$$\forall t : t \text{ enabled in } m \ \ \forall t' \in (\bullet t)\bullet : t' + 1 - t > 0$$

For disabled transitions, we first need to pick an insufficiently marked pre-place.

$$\forall t : t \text{ disabled in } m \qquad \sum_{p \in (\bullet t), m(p) < W(p,t)} p > 0$$

For the picked place, all pre-transitions need to be inserted.

$$\forall p \in P \forall t \in \bullet p : t + 1 - p > 0$$

Since the translation is more or less a re-formulation of the requirements of Def. 13, we can state without proof:

Theorem 2. *If the ILP returns a solution, the set of all transitions that have value 1 in this solution form a stubborn set with minimal number of enabled transitions. If the ILP is infeasible, T is the stubborn set with minimal number of enabled transitions.*

For stubborn sets in final Büchi states, the approach is analogous.

7.2 Enumeration of Up-Sets

The remaining problem is to enumerate reasonable up-sets. Input is a marking m and a formula φ. Output is a family $UP(m, \varphi)$ of up-sets. By Def. 13, we are only interested in up-sets which do not contain enabled transitions.

We follow the discussion in Sect. 6.1 and proceed by induction on the structure of the formula. For atomic propositions, up-sets depend on the particular nature of the propositions. In our example language, we have $UP(m, p \geq k) = \{\bullet p\}$ if $\bullet p$ does not contain enabled transitions, otherwise $UP(m, p \geq k) = \emptyset$. Likewise, $UP(m, p < k)$ is $\{p\bullet\}$ or $\emptyset$.

For the Boolean operations, we naturally derive $UP(m, \varphi_1 \wedge \varphi_2) = UP(m, \varphi_1)$ if $m \models \varphi_2$, $UP(m, \varphi_1 \wedge \varphi_2) = UP(m, \varphi_2)$ if $m \models \varphi_1$, $UP(m, \varphi_1 \wedge \varphi_2) = UP(m, \varphi_1) \cup UP(m, \varphi_2)$, else. Remember that $m \not\models \varphi_1 \wedge \varphi_2$ when we compute up-sets. For disjunction, we get $UP(m, \varphi_1 \vee \varphi_2) = \{M_1 \cup M_2 \mid M_1 \in UP(m, \varphi_1), M_2 \in UP(m, \varphi_1)\}$.

If c_φ denotes the number of elements in $UP(m, \varphi)$, we can give the following constraints for the values c_φ.

$$\begin{aligned} c_{p \geq k} &= 1 \\ c_{p < k} &= 1 \\ c_{\varphi_1 \wedge \varphi_2} &\leq c_{\varphi_1} + c_{\varphi_2} \\ c_{\varphi_1 \vee \varphi_2} &= c_{\varphi_1} \cdot c_{\varphi_2} \end{aligned}$$

In principle, this number can grow exponentially. We believe, however, that most properties occurring in practice will have a limited alternation between conjunction and disjunction, so the number should not be a problem.

The returned up-sets are the smallest ones we can exhibit without further knowledge of the net. If there is no up-set without enabled transitions, or there is one beyond the ones considered by the recursive procedure sketched above, we will not launch the ILP instance but return T as a stubborn set.

7.3 Experimental Validation

Unfortunately, we do not have experimental results on nontrivial elementary properties yet. We can, however, provide results for the property $\mathbf{G}\,\varphi$. This property is violated if a state m is reachable where $m \not\models \varphi$. Hence, we can evaluate our approach, in particular the effect of minimizing the stubborn sets by enumeration of up-sets. We compare the new, ILP based approach to the existing implementation in our tool LoLA where an arbitrary up-set is picked, and an arbitrary insufficiently marked pre-place is chosen. We compare these two approaches using benchmark examples from the 2011 issue of the model checking contest [9]. Using the existing LoLA implementation for comparison is justified by the excellent overall performance of LoLA in that contest. The reachablity queries in the contest were all global reachability predicates which as such do not leave any invisible transition (which is not a problem for progressing formulas). For traditional stubborn set methods, the absence of invisible transitions would

Table 1. Comparing the original LoLA with the new ILP approach. All numbers are states explored for checking 20 tasks (fomulas) for each family and scale factor. After 5 million states we stopped the check (–).

model		**states with LoLA**			**states with LoLA-ILP**		
family	**scale**	**min**	**avg**	**max**	**min**	**avg**	**max**
FMS	2	11	436	1,230	11	56	203
FMS	5	53,929	147,760	308,435	76	241	665
FMS	10	–	–	–	121	625	1,970
FMS	50	–	–	–	481	9,586	39,410
FMS	100	–	–	–	931	35,524	153,710
FMS	200	–	–	–	1,831	136,527	607,310
FMS	500	–	–	–	4,531	832,534	3,768,110
Kanban	5	54	302,121	860,406	46	304	1,131
Kanban	10	–	–	–	266	2,498	12,832
Kanban	20	–	–	–	516	9,121	51,782
Kanban	50	–	–	–	1,266	52,932	322,232
Kanban	100	–	–	–	2,516	205,749	1,284,982
Kanban	200	–	–	–	5,016	804,110	–
Kanban	500	–	–	–	12,516	2,623,553	–
MAPK	8	132	1,775,353	–	93	32,701	93,948
MAPK	20	–	–	–	539,722	2,206,091	–

require them to produce an unreduced state space. For this reason, we do use any other tool in our experiments.

We checked three family of models: a flexible manufacturing system (FMS), Kanban (a benchmark from the SMART model checker [3]), and a system biological model (MAPK). Each family can be scaled by adjusting the size of the initial marking. The interested reader is referred to [9] for a detailed discussion of the models.

Of course, calculation of stubborn sets using ILP is much slower than classical methods where nondeterminism is resolved arbitrarily but without backtracking. In the experiments, the ILP version of LoLA needed about 50 to 100 times as long as the original LoLA implementation for calculating the same number of states. As the numbers show, this time is well invested, though. First, the investment of more expensive stubborn set calculation pays off by the smaller number of states to be explored. That is, the ILP version typically finishes earlier than the original LoLA. Second, the smaller state spaces increase the odds that a problem instance ca be solved within the given memory. This is actually much more important in the model checking area than run time.

8 Related Work and Conclusion

Most approaches on partial order reduction focus on reduction of the transition system under investigation. To our best knowledge, [8] is the closest approach that also takes care of the Büchi state when calculation stubborn sets. Basically, they relax the original invisibility requirement to those propositions that are ahead of the current Büchi state. In contrast, our approach requires invisibility only for one proposition at a time, and only for retarding formulas. We have to pay for this improvement with the inclusion of a whole up-set in the stubborn set whereas [8] needs to include only an enabled transition. This means that they may be able to find stubborn sets with only invisible enabled transitions more frequently. We address this issue by a more sophisticated calculation procedure where all the available nondeterminism is exploited for the sake of finding a good stubborn set.

This paper lacks experimental evaluation of the actual reduction power. We compensate this problem by collecting other, indirect evidence for the significance of our approach. First, our approach is applicable where other approaches generally fail. We tolerate partial violation of the stuttering invariance, we tolerate lack of invisible transitions w.r.t. the progressing formulas, and we tolerate search strategies where cycles in the product system cannot be found. Furthermore our technique is a proper generalization of [13] which has already shown its merit. That approach has been further explored in [10]. In future work, we have to study whether these ideas can be applied to simple linear time properties as well.

Determining an optimal stubborn set is a question that has received much attention [20,5,19]. With our translation to ILP, we exploit all the nondeterminism that is available in the particular closure operation we rely on, as well as most of the nondeterminism in the choice of an up-set. Experimental results are quite encouraging. Nevertheless, our minimality criterion (minimal number of enabled transitions in the stubborn set) is known not necessarily to be the best choice [20]. It optimizes the size of stubborn sets only locally but cannot guarantee minimal resulting product systems. Hence, optimal stubborn set calculation that takes care of up-sets remains an open question.

Acknowledgement. This work was partially funded by the DFG (German research foundation) in the project WS4Dsec in the priority program Reliably Secure Software Systems (SPP 1496).

References

1. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 10^{20} states and beyond. Inf. Comput. 98(2), 142–170 (1992)
2. Clarke, E.M., Biere, A., Raimi, R., Zhu, Y.: Bounded model checking using satisfiability solving. Formal Methods in System Design 19(1), 7–34 (2001)
3. Ciardo, G., et al.: The smart model checker, `http://www.cs.ucr.edu/~ciardo/SMART`
4. Fahland, D., Favre, C., Koehler, J., Lohmann, N., Völzer, H., Wolf, K.: Analysis on demand: Instantaneous soundness checking of industrial business process models. Data Knowl. Eng. 70(5), 448–466 (2011)

5. Geldenhuys, J., Hansen, H., Valmari, A.: Exploring the Scope for Partial Order Reduction. In: Liu, Z., Ravn, A.P. (eds.) ATVA 2009. LNCS, vol. 5799, pp. 39–53. Springer, Heidelberg (2009)
6. Gerth, R., Kuiper, R., Peled, D., Penczek, W.: A partial order approach to branching time logic model checking. In: IEEE 3rd Israel Symp. on the Theory of Computing and Systems, pp. 130–140 (1995)
7. Godefroid, P., Wolper, P.: A partial approach to model checking. In: 6th IEEE Symp. on Logic in Computer Science, Amsterdam, pp. 406–415 (1991)
8. Kokkarinen, I., Peled, D., Valmari, A.: Relaxed Visibility Enhances Partial Order Reduction. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 328–339. Springer, Heidelberg (1997)
9. Kordon, F., et al.: Report on the model checking contest at Petri Nets, LNCS ToPNoC (2011), more information provided on `http://sumo.lip6.fr/mcc.html` (accepted for publication in January 2012)
10. Kristensen, L.M., Valmari, A.: Improved Question-Guided Stubborn Set Methods for State Properties. In: Nielsen, M., Simpson, D. (eds.) ICATPN 2000. LNCS, vol. 1825, pp. 282–302. Springer, Heidelberg (2000)
11. Peled, D.: All From One, One For All: On Model–Checking Using Representitives. In: Courcoubetis, C. (ed.) CAV 1993. LNCS, vol. 697, pp. 409–423. Springer, Heidelberg (1993)
12. Reisig, W.: Elements Of Distributed Algorithms: Modeling and Analysis with Petri Nets. Springer (September 1998)
13. Schmidt, K.: Stubborn Sets for Standard Properties. In: Donatelli, S., Kleijn, J. (eds.) ICATPN 1999. LNCS, vol. 1639, pp. 46–65. Springer, Heidelberg (1999)
14. Schmidt, K.: Stubborn sets for model checking the EF/AG fragment of CTL. Fundam. Inform. 43(1-4), 331–341 (2000)
15. Stahl, C., Reisig, W., Krstic, M.: Hazard detection in a GALS wrapper: A case study. In: ACSD 2005, pp. 234–243. IEEE Computer Society (2005)
16. Valmari, A.: Error detetction by reduced reachability graph generation. In: 9th European Workshop on Application and Theory of Petri Nets, Venice, Italy, pp. 95–112 (1988)
17. Valmari, A.: A stubborn attack on state explosion. In: Formal Methods in System Design 1, pp. 297–322 (1992)
18. Valmari, A.: Stubborn set methods for process algebras. In: Workshop on Partial Order Methods in Verification, Princeton, pp. 192–210 (1996)
19. Valmari, A., Hansen, H.: Can Stubborn Sets Be Optimal? In: Lilius, J., Penczek, W. (eds.) PETRI NETS 2010. LNCS, vol. 6128, pp. 43–62. Springer, Heidelberg (2010)
20. Varpaaniemi, K.: On the stubborn set method in reduced state space generation. PhD thesis, Helsinki University of Technology (1998)
21. Wolf, K.: Generating Petri Net State Spaces. In: Kleijn, J., Yakovlev, A. (eds.) ICATPN 2007. LNCS, vol. 4546, pp. 29–42. Springer, Heidelberg (2007)

Hybrid On-the-Fly LTL Model Checking with the Sweep-Line Method*

Sami Evangelista[1] and Lars Michael Kristensen[2]

[1] LIPN — Laboratoire d'Informatique de l'Université Paris Nord
99, av. J-B Clément, 93430 Villetaneuse, France
sami.evangelista@lipn.univ-paris13.fr
[2] Department of Computer Engineering, Bergen University College, Norway
Lars.Michael.Kristensen@hib.no

Abstract. The sweep-line method allows explicit state model checkers to delete states from memory on-the-fly during state space exploration thereby lowering the memory demands of the verification procedure. The sweep-line method is based on a least progress-first search order that prohibits the immediate use of standard on-the-fly LTL model checking algorithms that rely on a depth-first search order. This paper proposes and experimentally evaluates an algorithm for LTL model checking compatible with the search order prescribed by the sweep-line method.

1 Introduction

A main paradigm in explicit state model checking is to limit memory requirements by storing only a subset of the visited states in memory at a time. This means that the peak memory usage is reduced. The subsets of the state space stored in memory during state space exploration are chosen in such a way that termination of the exploration is still guaranteed. State caching [14,17] was one of the first methods based on this paradigm and relies on storing only the states on the depth-first search stack in memory. The *sweep-line method* [5,18] and the *to-store-or-not-to-store method* [2] represent more recent methods based on the paradigm of on-the-fly state deletion, and use other conditions for determining the subsets of states that are to be stored in memory.

The basic idea of the sweep-line method is to exploit a notion of *progress* exhibited by many systems. Exploiting progress makes it possible to explore all reachable states while storing only small subsets of the state space in memory at a time. The subsets of states stored are determined via a progress value assigned to each state, and the method explores the states in a progress-first order. The sweep-line method explore all states with a given progress value before progressing to the states with a higher progress value. When the method proceeds to the consider states with a higher progress value, it deletes the subset of states with a lower progress value. The assumption is that the system does not make

* Supported by a Norwegian Research Council (NRC) Yggdrasil grant and NRC project 194521 (FORMGRID).

regress, and hence states with a lower progress value will not be visited again and do not need to be kept in memory. If the system does make regress, then the method will mark states at the end of *regress edges* as persistent (i.e., make them permanently stored in memory) in order to ensure termination. In presence of regress, the sweep-line method may visit some states multiple times. The sweep-line method is in its simplest form [5,18] aimed at on-the-fly verification of *safety properties*, such as determining whether a reachable state exists that satisfies a given state predicate. The theoretical foundation of the sweep-line method has been further developed in several papers [3,11,18,20,19] and the method has been implemented in the ASAP platform [25] and in the LoLA tool [22]. The sweep-line method has been used [12,13,15,23] for the verification of several industrial-sized protocols specified using the CPN modelling language.

An open research question that has not been addressed in the earlier papers on the sweep-line method is how to combine the sweep-line method with on-the-fly model checking of Linear Time Temporal Logic (LTL) properties. The conventional approach to on-the-fly LTL model checking is based on the exploration of a product Büchi automaton: the negation of the LTL formula to be checked is represented as a Büchi automata [24] and the product of this property automaton and the state space (viewed as a Büchi automata) is explored using a nested depth-first traversal [7] in search for an *acceptance cycle*, i.e., a cycle containing an acceptance state. The challenge in the context of the sweep-line method is that the nested depth-first search of LTL model checking is incompatible with the progress-based search order of the sweep-line method as the latter cannot guarantee that states in an acceptance cycle will be present in memory simultaneously. The basic idea of the hybrid approach developed in this paper is to use nested depth-first search to detect acceptance cycles where the states on the cycle all have the same progress value, and use a variation of the MAP algorithm [1,4] to detect acceptance cycles that span multiple progress values.

The rest of this paper is organised as follows. Section 2 introduces the basic concepts underlying LTL model checking and the sweep-line state space exploration algorithm. Section 3 then presents our algorithm for conducting LTL model checking with the sweep-line method. The correctness of this algorithm is proved in Sect. 4 along with its complexity. In Sect. 5, we discuss some possible extensions and variations of our algorithm. Section 6 presents the results from the experimental evaluation that we have performed based on an implementation of the proposed algorithm. Finally, in Sect. 7 we sum up our conclusions and discuss directions for future work. The reader is assumed to be familiar with the basic idea of explicit state space exploration.

2 Background

2.1 LTL Model Checking

LTL model checking is usually performed following the automata-based approach originating from [24] that proceeds in two steps, the first being the translation of the negation of the LTL formula to be checked into a Büchi automata.

In this paper we focus only on the second step of the process that can be reduced to a graph problem [6]: given a graph representing the synchronised product of the Büchi property automaton and the state space of the system, find a cycle containing an accepting state. A state of the synchronised product is an acceptance state if the Büchi property automaton component of the state is an acceptance state. Any such identified cycle determines an infinite execution of the system violating the LTL formula. Acceptance cycles can be detected using nested depth-first search [7] or a variation of Tarjan's algorithm for strongly connected component (SCC) detection [8]. Hence, we will only reason on *automaton graphs* that result from the product of a Büchi property automaton and a state space graph describing the behaviour of the modelled system.

Definition 1. *An* ***automaton graph*** *G is a 4-tuple $(\mathcal{S}, \mathcal{E}, s_0, \mathcal{A})$ where $\mathcal{S}$ is a finite and non-empty set of states; $\mathcal{E} \subseteq \mathcal{S} \times \mathcal{S}$ is a finite set of edges; $s_0 \in \mathcal{S}$ is an initial state; and $\mathcal{A} \subseteq \mathcal{S}$ is a set of accepting states.*

Notation. For an automaton graph $G = (\mathcal{S}, \mathcal{E}, s_0, \mathcal{A})$ we write $s \to s'$ if $(s, s') \in \mathcal{E}$; and $s \to^* s'$ if there exists $s_1, \ldots, s_n$ with $s_1 = s$, $s_n = s'$ and $s_i \to s_{i+1}$ for $1 \leq i \leq n-1$. Acceptance states are graphically represented using double circles.

2.2 The Sweep-Line Method

The sweep-line method [5,18] deletes states on-the-fly by exploiting a particular notion of progress in the system. Progress is formally captured by a *progress measure* that quantifies the progression of a state:

Definition 2. *Let $G = (\mathcal{S}, \mathcal{E}, s_0, \mathcal{A})$ be an automaton graph. A* ***progress measure*** *(or progress mapping) for G is a mapping from $\mathcal{S}$ to Φ, where Φ is a non-empty set of progress values equipped with a total order $\sqsubseteq$.*

A progress mapping determines a partition of edges into *progress edges* corresponding to steps that increase the progress value (i.e., edges (s, s') with $\phi(s) \sqsubset \phi(s')$); *stationary edges* connecting two states with a same progress value; and *regress edges* corresponding to steps that decreases the progress value (i.e., edges (s, s') with $\phi(s') \sqsubset \phi(s)$). Algorithm 1 is the generalised sweep-line algorithm [18] that maintains four data structures:

$\mathcal{H}$ is a hash table used to store states currently in memory;
$\mathcal{G} \subseteq \mathcal{H}$ contains states that will be garbage collected (deleted) when possible;
$\mathcal{R} \subseteq \mathcal{H}$ contains states that will serve as roots during the next sweep (i.e., an iteration of the algorithm at ll. 4–10); and
$\mathcal{Q} \subseteq \mathcal{H}$ contains states that have not yet been processed by the algorithm.

A sweep works basically as a standard state space exploration initiated from a set of root states $\mathcal{R}$ initialised, for the first sweep, with the initial state s_0. States are expanded and their successors that have not been visited so far are put in $\mathcal{Q}$ to be later expanded. This process ends when $\mathcal{Q}$ becomes empty.

Algorithm 1. Sweep, the generalised sweep-line algorithm of [18]

```
1  algorithm Sweep is
2      $s_0.pers$ := false ; $\mathcal{R} := \{s_0\}$ ; $\mathcal{H} := \{s_0\}$
3      while $\mathcal{R} \neq \emptyset$ do
4          $\mathcal{Q} := \mathcal{R}$ ; $\mathcal{R} := \emptyset$
5          while $\mathcal{Q} \neq \emptyset$ do
6              $\mathcal{G} := \emptyset$ ; $\phi_m := \mathcal{Q}.minProgress$ ()
7              while $\mathcal{Q}.minProgress$ () $= \phi_m$ do
8                  $s := \mathcal{Q}.dequeue$ ()
9                  $visit$ ($s$)
10             $\mathcal{H} := \mathcal{H} \setminus \mathcal{G}$
11 procedure $visit(s)$ is
12     for $(s, s') \in \mathcal{E}$ do
13         if $s' \notin \mathcal{H}$ then
14             $s'.pers := \phi(s') \sqsubset \phi(s)$
15             $\mathcal{H} := \mathcal{H} \cup \{s'\}$
16             if $s'.pers$ then
17                 $\mathcal{R} := \mathcal{R} \cup \{s'\}$
18             else
19                 $\mathcal{G} := \mathcal{G} \cup \{s'\}$
20                 $\mathcal{Q} := \mathcal{Q} \cup \{s'\}$
```

The search progresses using a least progress-first policy determined by mapping ϕ: the algorithm proceeds layer by layer, a *layer* being defined as a set of states sharing the same progress value, i.e., connected by stationary edges. Two differences are introduced by the sweep-line method compared to standard state space exploration. First, we perform garbage collection at l. 10 by removing a whole layer of states sharing progress value ϕ_m (ll. 7–9), and before processing states with a higher progress value. Any state can be deleted this way except for *persistent* states for which the detection is described below. Conceptually, there exists a *sweep-line* that separates already visited (and deleted) states from states to be processed present in $\mathcal{Q}$. This line advances to include new states after a whole layer of states with the same progress value has been processed. The only situation the sweep-line can move back is before the start of a new sweep. A second difference is that to guarantee termination, the algorithm identifies regress edges (l. 14), and marks their destination as persistent, indicating that these may not be deleted from memory. Indeed, for any cycle it holds that either all its states have the same progress value or at least two of its states are connected by a regress edge. In the first case, no state of the cycle will be garbage collected as long as at least one of its states remains in $\mathcal{H}$ and in the second case, the destination of the regress edge will always remain in $\mathcal{H}$ after having been detected. Hence, it is guaranteed that the algorithm will not visit the same states over and over since at least one state per cycle will be present in $\mathcal{H}$. Note that the destination of a regress edge is not put in $\mathcal{Q}$ but in $\mathcal{R}$ to serve as a root for the next sweep (ll. 16–17).

The snapshot of the Sweep algorithm at three different stages is presented in Fig. 1 for a simple example. Information on accepting states have not been drawn on this figure as they are not relevant to illustrate the principle of the algorithm. States are ordered left to right according to their progress value. The conceptual sweep-line is represented as a vertical dotted line. After the visit of states 0 and 1 (Stage 1) their successor states 2, 3 and 4 are put the queue. State 0 and 1 can then be deleted since they have been processed and their progress value is strictly smaller than the minimal value found in $\mathcal{Q}$, i.e., $\phi(2)$. Hence, the sweep-line method makes the assumption that they cannot be reached from the set of unprocessed states. At Stage 2, states 7 and 8 have been processed. All states from

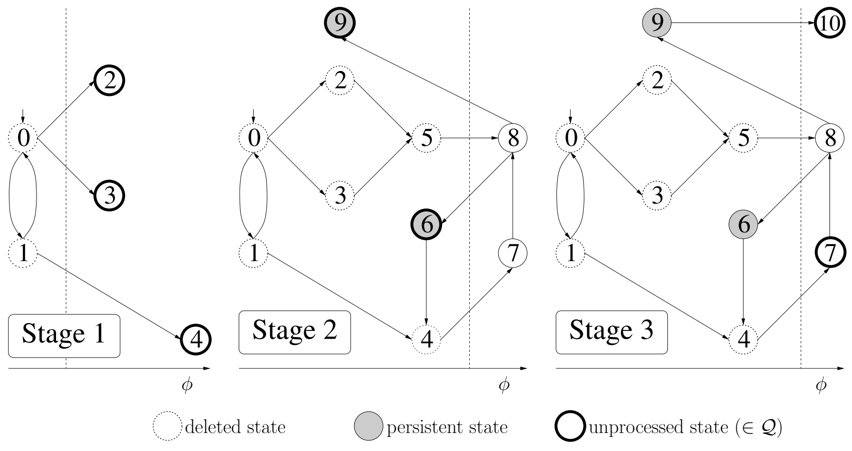


Fig. 1. Snapshot of algorithm Sweep at different stages

0 to 6 have been previously deleted from $\mathcal{H}$. The visit of state 8 then generated state 6, already explored but deleted from memory, and state 9, seen for the first time. Since edges (8,6) and (8,9) are both regress edges, states 6 and 9 are marked as persistent and put in set $\mathcal{R}$ to serve as roots for the next sweep. Once the expansion of states 7 and 8 is finished they are deleted and this sweep terminates. A new sweep starts with states 6 and 9 as roots. The algorithm then visits states in the following order given by ϕ: 9, 6, 4, 10, 7, 8. After the visit of states 9, 6 and 4 (Stage 3), 9 and 6 will not be deleted since they were marked as persistent during the previous sweep. Hence, the cycle 8→6→4→7→8 is detected during the visit of state 8 that generates 6 already in $\mathcal{H}$.

3 A Sweep-Line Algorithm for LTL Model Checking

In this section we introduce our new LTL model checking algorithm. This algorithm consists of two distinct components each dedicated to the detection of specific kinds of accepting cycles. Before introducing these two components, we describe a property of accepting cycles.

Let us suppose that state 1 in Fig 1 is an accepting state. The accepting cycle 1→0→1 could be easily discovered by algorithm Sweep previously introduced since all its states share the same progress value and will therefore be simultaneously present in memory at some stage. If state 6 is an accepting state, then the accepting cycle 6→4→7→8→6 will not be discovered by the sweep-line method since its states are distributed upon several layers. This property of cycles is formalised through the following definition.

Definition 3. *Let $G = (\mathcal{S}, \mathcal{E}, s_0, \mathcal{A})$ be an automaton graph, and ϕ be a progress mapping for G. An accepting cycle $c = s_1 \rightarrow s_2 \rightarrow \ldots \rightarrow s_n \rightarrow s_1$ is a* ***single layer accepting cycle*** *(SLAC) if and only if $\phi(s_i) = \phi(s_j), \forall i, j \in \{1, \ldots, n\}$. Otherwise c is a* ***multiple layers accepting cycle*** *(MLAC).*

Our algorithm separates the detection of SLACs from the detection of MLACs. In order to discover SLACs, the algorithm builds upon the classical NDFS algorithm while MLACs are taken care of by a combination of the sweep-line algorithm and the MAP algorithm [1,4]. Before introducing the LTL model checking algorithm, we specify the property it should have in terms of being compatible with the search order of the sweep-line method. The definition below specify that a sweep-line compliant algorithm may not keep in memory a state behind the sweep-line that would be deleted by algorithm Sweep.

Definition 4. *Let $G = (\mathcal{S}, \mathcal{E}, s_0, \mathcal{A})$ be an automaton graph and ϕ be a progress mapping for G. An LTL model checking algorithm storing states to process in a queue $\mathcal{Q}$ is **sweep-line compliant** if and only if, at any step t of the algorithm:*

$$\forall s \in \mathcal{S}^t : (\exists s' \in \mathcal{S} \text{ with } (s', s) \in \mathcal{E}^t \wedge \phi(s) \sqsubset \phi(s')) \ \vee \ (min_{s' \in \mathcal{Q}}(\phi(s')) \sqsubseteq \phi(s))$$

where $\mathcal{S}^t \subseteq \mathcal{S}$ denotes all states kept in memory by the algorithm at step t, and $\mathcal{E}^t \subseteq \mathcal{E}$ denotes all edges connecting states in $\mathcal{S}^t$.

3.1 Combining NDFS and Sweep to Discover SLACs

Algorithm 2 contains the pseudo-code of procedure LTL-Sweep that is a variation of the sweep-line algorithm equipped with a mechanism that allows the detection of SLACs by performing local nested depth-first searches on layers of states sharing the same progress value. A state has three associated boolean flags, all initialised to *false* (ll. 12–14): *pers* indicating if the state is persistent and must not be garbage collected; *blue* and *red* that indicate if the state has been visited by the first level DFS (the blue DFS) or the second level DFS (the red DFS).

The principle of NDFS is to interleave a blue DFS looking for accepting states, and red DFSs looking for cycles containing accepting states reached by the blue DFS. When the blue DFS backtracks from an accepting state s, it initialises a red DFS rooted in s (called *seed* in the first presentation of the algorithm [7]) to find whether s is reachable from itself. The algorithm works in linear time since the result of a red DFS (i.e., marking visited states as red) can be reused in subsequent red DFSs. In addition to this linear complexity, NDFS also has other appreciable characteristics, among which: its low memory requirements (only 2 bits required per state, the blue and red bits), its ability to report accepting cycles on-the-fly, and its easy combination with partial-order reduction [16].

Starting from the initial state, algorithm LTL-Sweep repeatedly perform sweeps using procedure *findSLAC*, until no new persistent state is found. An iteration of procedure *findSLAC* (ll. 17–22) consists of removing from the priority queue $\mathcal{Q}$ all states with the lowest progress value and performing local NDFSs on theses states. All non-persistent states visited by these NDFSs are then present in $\mathcal{G}$ and can be removed from $\mathcal{H}$ (l. 22). If the main loop fails to find an SLAC, the *findMLAC* is invoked to look for an MLAC. Procedures *dfsBlue* and *dfsRed* follow the same principle as algorithm NDFS with the following modifications to the DFSs:

Algorithm 2. LTL-Sweep, a sweep-line algorithm for LTL model checking

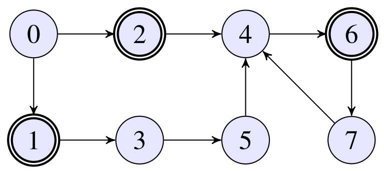
```
1  algorithm LTL-Sweep is
2      $insert\ (s_0)$
3      $\mathcal{R} := \{s_0\}$
4      while $\mathcal{R} \neq \emptyset$ do
5          $\mathcal{Q} := \mathcal{R}$
6          $\mathcal{R} := \emptyset$
7          $findSLAC\ ()$
8      $findMLAC\ (\mathcal{H})$
9      report "no cycle found"
10 procedure $insert(s)$ is
11     $\mathcal{H} := \mathcal{H} \cup \{s\}$
12     $s.pers :=$ false
13     $s.blue :=$ false
14     $s.red :=$ false
15 procedure $findSLAC()$ is
16     while $\mathcal{Q} \neq \emptyset$ do
17         $\mathcal{G} := \emptyset$
18         $\phi_m := \mathcal{Q}.minProgress\ ()$
19         while $\mathcal{Q}.minProgress\ () = \phi_m$ do
20             $s := \mathcal{Q}.dequeue\ ()$
21             $dfsBlue\ (s)$
22         $\mathcal{H} := \mathcal{H} \setminus \mathcal{G}$
23 procedure $dfsBlue(s)$ is
24     if $\neg s.blue$ then
25         $s.blue :=$ true
26         if $\neg s.pers$ then $\mathcal{G} := \mathcal{G} \cup \{s\}$
27         for $(s, s') \in \mathcal{E}$ do
28             if $s' \notin \mathcal{H}$ then
29                 $insert\ (s')$
30             if $\phi(s) = \phi(s')$ then
31                 $dfsBlue\ (s')$
32             else if $\phi(s') \sqsubset \phi(s)$ then
33                 $s'.pers :=$ true
34                 $\mathcal{R} := \mathcal{R} \cup \{s\}$
35             else
36                 $\mathcal{Q} := \mathcal{Q} \cup \{s\}$
37         if $s \in \mathcal{A}$ then $dfsRed\ (s, s)$
38 procedure $dfsRed(s, seed)$ is
39     $s.red :=$ true
40     for $(s, s') \in \mathcal{E}$ with $\phi(s) = \phi(s')$ do
41         if $s' = seed$ then
42             report "SLAC found"
43         else if $\neg s'.red$ then
44             $dfsRed\ (s', seed)$
```

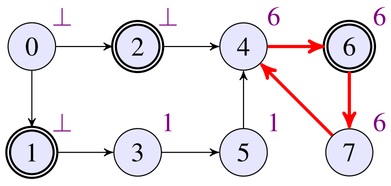
- Any visited state is put in the garbage set $\mathcal{G}$ if not persistent (l. 26).
- DFSs are limited to states sharing the same progress value (ll. 30–31, l. 40).
- Finally, two alternatives arise for any new state s' reached by the first level DFS (*dfsBlue*) from a state s belonging to a different layer: it is put in the root set $\mathcal{R}$ and marked as persistent if it is behind the sweep-line (ll. 32–34); or, if it is in front of the sweep-line (ll. 35–36), it is put in the priority queue $\mathcal{Q}$ to be later visited by a next NDFS during a subsequent iteration of procedure *findSLAC*.

3.2 Combining MAP and Sweep to Discover MLACs

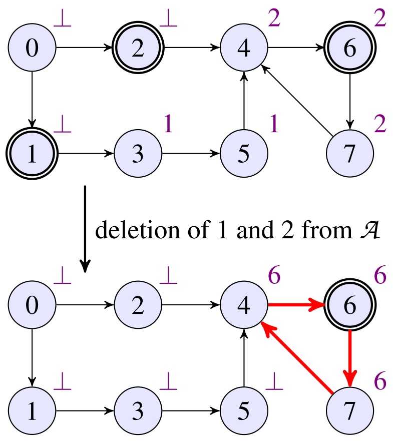
Local nested DFSs are guaranteed to find any SLAC, but the algorithm relies on another procedure to find accepting cycles split upon several layers. This one is based on the principle that any MLAC always contains at least one regress edge and, hence, one persistent state. Thus, if the algorithm has failed to find an SLAC it will launch procedure *findMLAC* to possibly discover an MLAC containing a persistent state (l. 8 of Alg. 2). This procedure is invoked with the hash table $\mathcal{H}$ that contains, at l. 8 of Algorithm 2, all persistent states discovered.

The algorithm we propose to find MLACs is an adaptation of the MAP algorithm initially designed in the context of distributed memory model checking [4], and later adapted for external memory storage [1].


(a) Input graph



(b) MAP with $6 >_{\mathcal{S}} 2 >_{\mathcal{S}} 1$



(c) MAP with $2 >_{\mathcal{S}} 1 >_{\mathcal{S}} 6$ (top) and after deletion (bottom)

Fig. 2. Illustration of the MAP algorithm

Principle of the MAP Algorithm. For an automaton graph $G = (\mathcal{S}, \mathcal{E}, s_0, \mathcal{A})$, MAP assumes a total order relation $>_{\mathcal{S}} \subseteq \mathcal{S} \times \mathcal{S}$ that is used to determine a *maximal accepting predecessor* function $map_G : \mathcal{S} \rightarrow \mathcal{A} \cup \{\perp\}$. Intuitively $map_G(s)$ is the largest accepting state that is backward reachable from s (or $\perp$ if there is no such state). The mapping map_G can be computed using a breadth-first search (MBFS) that propagates forward information on the maximal accepting predecessor in $\mathcal{O}(|\mathcal{S}| \cdot |\mathcal{A}|)$. Trivially, $map_G(s) = s$ implies the existence of an accepting cycle looping on s. Unfortunately the converse does not necessarily hold (see Figure 2(c) for an example) as an accepting state a that is outside a cycle containing an accepting state b will prevent from discovering this cycle if $a \rightarrow^* b \wedge a >_{\mathcal{S}} b$ (which implies that $\Rightarrow map_G(b) = a$). We will then say that b (or the cycle) is *hidden* by a. Hence, MAP alternates between MBFSs used to compute map_G and delete transformations used to remove states from $\mathcal{A}$ that may hide accepting cycles. Deleted states are those that have been propagated along the graph, i.e., the set $\{s \in \mathcal{A} \mid \exists s' \in \mathcal{S}, map_G(s') = s\}$. If there is no such state to delete, then there is no accepting cycle. Otherwise, it is guaranteed that any accepting cycle will be discovered within a finite number of iterations.

The behaviour of MAP is illustrated on the graph of Fig. 2(a) with two different orders. With the first order (see Fig. 2(b)), MAP finds the accepting cycle around 6 during the first iteration (since $map_G(6) = \max_{\{1,2,6\}} = 6$). With the second order (see Fig. 2(c)), the accepting cycle around 6 is not discovered after the first computation of map_G (since $map_G(6) = \max_{\{1,2,6\}} = 2$). States 1 and 2 that have been propagated during this first MBFS are both deleted from $\mathcal{A}$ before a second MBFS is initiated. Since 6 is now the only accepting state, the cycle is discovered. In the absence of edge $(6, 7)$, MAP would have stopped after this second iteration and reported the absence of accepting cycle.

Adaptation of the MAP Algorithm for MLAC Detection. Unlike state-of-the-art algorithms for LTL model checking, MAP is not based on a depth-first search and is as such a good candidate for a combination with the sweep-line method. However, we would like to have an algorithm that is sweep-line compliant (Def. 4) and a straightforward adaptation of the MAP algorithm would not have this property as it has to remember somehow accepting states removed from $\mathcal{A}$ by the delete transformation. We instead exploit the fact that any MLAC we are looking for contains at least one persistent state. This is a direct consequence of the fact that the cycle is distributed upon several layers and, hence, contains at least one regress edge. We therefore switch from the idea of maximal accepting predecessor to the one of maximal persistent predecessor formalised below. Note that the $mpp_G^{\mathcal{P}}$ function defined below associates a pair to each state, the second boolean component giving information on the backwards reachability of an accepting state as explained below.

Definition 5. *Let $G = (\mathcal{S}, \mathcal{E}, s_0, \mathcal{A})$ be a automaton graph, $\mathcal{P} \subseteq \mathcal{S}$ be a set of persistent states, and $>_{\mathcal{S}}$ be a total order relation on $\mathcal{S}$. For a state $s \in \mathcal{S}$, $R(s) = \{p \in \mathcal{P} \mid p \rightarrow^* s\}$ denotes the set of persistent states backwards reachable from s. The* ***maximal persistent predecessor*** *function $mpp_G^{\mathcal{P}} : \mathcal{S} \rightarrow \{\bot\} \cup (\mathcal{P} \times \{false, true\})$ is defined by:*

$$mpp_G^{\mathcal{P}}(s) = \begin{cases} (p, true) & \textit{if } R(s) \neq \emptyset,\ \forall p' \in R(s) \setminus \{p\} : p >_{\mathcal{S}} p' \\ & \textit{and } \exists a \in \mathcal{A} \textit{ such that } p \rightarrow^* a \rightarrow^* s \\ (p, false) & \textit{if } R(s) \neq \emptyset,\ \forall p' \in R(s) \setminus \{p\} : p >_{\mathcal{S}} p' \\ & \textit{and } \nexists a \in \mathcal{A} \textit{ such that } p \rightarrow^* a \rightarrow^* s \\ \bot & \textit{otherwise} \end{cases}$$

Intuitively, if $mpp_G^{\mathcal{P}}(s) = (p, b)$, then p is the largest persistent state that is backward reachable from s and $b = true$ if and only if there is a path from p to s containing an accepting state. Hence, the following proposition is a direct consequence of Def. 5.

Proposition 1. *Let $G = (\mathcal{S}, \mathcal{E}, s_0, \mathcal{A})$ be an automaton graph and $\mathcal{P} \subseteq \mathcal{S}$. If $mpp_G^{\mathcal{P}}(s) = (s, true)$ then G has an accepting cycle containing state s.*

Procedure *findMLAC* (see Algorithm 3) is an adaptation of the MAP algorithm for MLACs detection. Each iteration of the algorithm (ll. 4–7) consists of computing the $mpp_G^{\mathcal{P}}$ function; and then removing set $\mathcal{D}$ from $\mathcal{P}$. This set $\mathcal{D}$ contains states that have been propagated during the computation of $mpp_G^{\mathcal{P}}$ and that may hide some accepting cycle(s). It is initialised to $\mathcal{P}$ before each computation even though procedure *mpp* will discard from it hidden persistent states. In order to optimise the search, we also remove from $\mathcal{P}$ the states s such that $mpp_G^{\mathcal{P}}(s) = (\_, false)$ (l. 7). It follows from Def. 5 that these cannot be part of an accepting cycle. The procedure terminates when set $\mathcal{P}$ has been emptied meaning that all persistent states have been propagated.

Procedure *mpp* is a sweep-line compliant algorithm computing the maximal persistent predecessor function with states behind the sweep-line being deleted

Algorithm 3. Procedure *findMLAC* to discover multiple layers accepting cycles

```
1  procedure $findMLAC(\mathcal{H})$ is
2      $\mathcal{P} := \mathcal{H}$
3      while $\mathcal{P} \neq \emptyset$ do
4          $\mathcal{D} := \mathcal{P}$
5          $mpp\ ()$
6          $\mathcal{P} := \mathcal{P} \setminus \mathcal{D}$
7          $\mathcal{P} := \mathcal{P} \setminus \{s \mid s.mpp = (\_, \mathbf{false})\}$
8  procedure $visit(s, prop)$ is
9      for $(s, s') \in \mathcal{E}$ do
10         if $prop = (s', \mathbf{true})$ then
11             report "MLAC found"
12         if $s' \notin \mathcal{H}$ then
13             $insert\ (s')$ /* see Alg. 2 */
14             $s'.mpp := \perp$
15         if $prop >_{mpp} s'.mpp$ then
16             $\mathcal{Q} := \mathcal{Q} \cup \{s'\}$
17             $s'.mpp := prop$
18 procedure $mpp()$ is
19     for $s \in \mathcal{P}$ do
20         $s.mpp := (s, s \in \mathcal{A})$
21     $\mathcal{Q} := \mathcal{P}$
22     while $\mathcal{Q} \neq \emptyset$ do
23         $\mathcal{G} := \emptyset$
24         $\phi_m := \mathcal{Q}.minProgress\ ()$
25         while $\mathcal{Q}.minProgress\ () = \phi_m$ do
26             $s := \mathcal{Q}.dequeue\ ()$
27             $(p, acc) := s.mpp$
28             $prop := (p, acc \vee s \in \mathcal{A})$
29             if $s \in \mathcal{P}$ and $p >_{\mathcal{S}} s$ then
30                 $\mathcal{D} := \mathcal{D} \setminus \{s\}$
31             $visit\ (s, prop)$
32             if $\neg s.pers$ then
33                 $\mathcal{G} := \mathcal{G} \cup \{s'\}$
34         $\mathcal{H} := \mathcal{H} \setminus \mathcal{G}$
```

at l. 34. Before visiting a state s, we first determine the maximal persistent predecessor *prop* it will propagate to its successors (ll. 27–28). It is $s.mpp$ with the second component set to *true* if s is accepting. Moreover, if s is persistent and hidden by $s.mpp$ (ll. 29–30) it must be removed from set $\mathcal{D}$ as it must not be touched by the deletion transformation operated at l. 6.

The *visit* procedure evaluates whether a maximal predecessor value *prop* computed as explained above should be propagated to the successors s' of a state s. This decision is made according to the result of the comparison of *prop* and $s'.mpp$ using the order relation defined below.

Definition 6. *Let $G = (\mathcal{S}, \mathcal{E}, s_0, \mathcal{A})$ be an automaton graph and $>_{\mathcal{S}}$ be a total order relation on $\mathcal{S}$. We define the total order relation $>_{mpp}$ on $\{\perp\} \cup (\mathcal{S} \times \{false, true\})$ as follows:*

$$m >_{mpp} m' \Leftrightarrow \begin{cases} \quad m = (s, b) \wedge m' = (s', b') \wedge (s >_{\mathcal{S}} s' \vee s = s' \wedge b \wedge \neg b') \\ \vee\ m = (s, b) \wedge m' = \perp \end{cases}$$

The definition states that propagation takes place if we have found for s' a larger persistent predecessor than the previous one, or, starting from the same persistent predecessor, an alternative path containing an accepting state has now been found. State s' then has to be put in the priority queue $\mathcal{Q}$ to be later visited according to the same process. Note that from Def. 6 and ll. 12–14, *prop* is always propagated if s' is a new state. Finally, as stated by Prop. 1 an MLAC is found at ll. 10–11 if one reaches s' with $s'.mpp = (s', true)$.

An example is illustrated by Fig. 3. At the first iteration, we have $\mathcal{P} = \{1, 2, 3\}$ (states in dark gray on the figure). We assume that the order relation on $\mathcal{S}$ is such that $3 >_{\mathcal{S}} 2 >_{\mathcal{S}} 1$. The computation of the maximal persistent

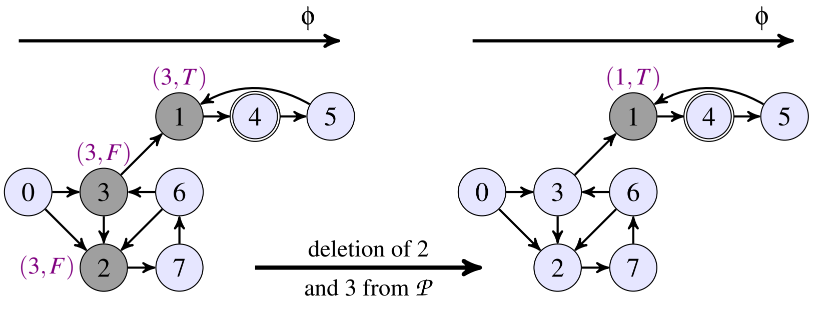


Fig. 3. Detection of an MLAC

predecessor function gives the following results: $mpp_\phi^\mathcal{P}(2) = mpp_\phi^\mathcal{P}(3) = (3, \mathit{false})$ and $mpp_\phi^\mathcal{P}(1) = (3, \mathit{true})$. The set $\mathcal{D}$ of states to remove from $\mathcal{P}$ is the singleton $\{3\}$ because 1 and 2 have been hidden by 3 and, hence, discarded from $\mathcal{D}$. However, we can also delete 2 from $\mathcal{P}$ since, as $mpp_\phi^\mathcal{P}(2) = (3, \mathit{false})$, no path starting from 1, 2, or 3 and leading to 3 can contain an accepting state. Hence, $\mathcal{P} = \{1\}$ after the removal, and after the second iteration, $mpp_\phi^\mathcal{P}(1) = (1, \mathit{true})$ and the MLAC $1 \rightarrow 4 \rightarrow 5 \rightarrow 1$ is discovered.

4 Correctness Proof and Complexity of LTL-Sweep

We first prove the correctness of our algorithm. The proof of Theorem 1 is inspired by the proof of the MAP algorithm [4].

Theorem 1. *Algorithm LTL-Sweep reports an accepting cycle if and only if the automaton graph has an accepting cycle.*

Proof. We prove that if the graph has an accepting cycle then a cycle is necessarily reported by the algorithm. The other direction follows immediately from the computation of $s.mpp$ and Prop. 1. Let $C = \{s_1, \ldots, s_n\}$ be an accepting cycle with $s_1 \rightarrow \ldots \rightarrow s_n \rightarrow s_1$. We consider two cases.

1. C is an SLAC. Since algorithm Sweep visits all states, and $\forall s_i, s_j \in C, \phi(s_i) = \phi(s_j)$, the first NDFS initiated on one of the states $s_i \in C$ will obviously report the cycle.
2. C is an MLAC. We denote by $\mathcal{P}_i$ the content of set $\mathcal{P}$ of states used as roots during the i^{th} call to procedure *mpp*. Let $s^{\max}$ be the largest persistent state of the cycle (i.e. $s^{\max} \in C \cap \mathcal{P}_0$) such that $\forall s_i \in (C \cap \mathcal{P}_0) \setminus \{s^{\max}\}, s^{\max} >_\mathcal{S} s_i$. This state exists since otherwise, we would have $\mathcal{P}_0 = \emptyset$ which would in turn mean that $\forall s_i, s_j \in C, \phi(s_i) = \phi(s_j)$ and, hence, that C is an SLAC. If the cycle C is not reported by the i^{th} call to procedure *mpp* then it necessarily holds that once the procedure has terminated, $s^{\max}.mpp = (m, \mathit{true})$ with $m >_\mathcal{S} s^{\max}$ which implies that $s^{\max} \notin \mathcal{D}$ and $m \in \mathcal{D}$. Now, once *mpp* has

terminated, if $\exists s \in \mathcal{S} \mid s.mpp = (s', b)$ with $s' \neq s$ then $s' \in \mathcal{D}$ and $s \notin \mathcal{D}$. This, combined with the fact that $s^{\max}.mpp = (m, true)$, implies that $s^{\max}$ is not touched by the deletion transformation (ll. 6–7 of Alg. 3) and therefore belongs to $\mathcal{P}_{i+1}$ while m does not. Since $\mathcal{P}$ is finite, C (or another MLAC) is necessarily reported by a j^{th} (with $j > i$) call to *mpp*. $\square$

Theorem 2. *Algorithm LTL-Sweep terminates after having explored at most* $2 \cdot |\mathcal{P}| \cdot |\mathcal{S}| + 2 \cdot |\mathcal{P}|^3 \cdot |\mathcal{S}|$ *states where* $\mathcal{P}$ *denotes the set of persistent states computed by Algorithm Sweep.*

Proof. Algorithm Sweep explores at most $|\mathcal{P}| \cdot |\mathcal{S}|$ states [18]. Therefore the same algorithm combined with NDFS to detect SLACs explores at most $2 \cdot |\mathcal{P}| \cdot |\mathcal{S}|$ states. Procedure *mpp* of Alg. 3 visits each state $s \in \mathcal{P}$ at most $2 \cdot |\mathcal{P}|$ times: for any $s' \in \mathcal{P}$ with $s' >_{\mathcal{S}} s$, it can be visited a first time with $s.mpp = (s', false)$ and a second time with $s.mpp = (s', true)$. Each visit by procedure *mpp* of $s \in \mathcal{P}$ generates at most $|\mathcal{S}|$ visits. Hence, procedure *mpp* terminates after visiting at most $2 \cdot |\mathcal{P}|^2 \cdot |\mathcal{S}|$. Therefore, since procedure *findMLAC* performs at most $|\mathcal{P}|$ iterations and calls to *mpp*, it explores at most $2 \cdot |\mathcal{P}|^3 \cdot |\mathcal{S}|$ states. $\square$

5 Extensions

We propose in this section extensions to the algorithm we introduced in the previous section. The first extension has been implemented in our tool and the experimental section discusses its benefits. The second and third extensions are opportunities for future research directions and have not been implemented yet.

5.1 On-the-Fly MLAC Detection

The algorithm we introduced can detect SLACs on-the-fly, i.e., without the need of exploring the entire graph. Indeed, each time a sweep will encounter a layer containing an SLAC, the use of NDFS guarantees an early termination of the algorithm. However, an MLAC will be discovered only when no SLAC has been discovered and, hence, after a complete visit of the automaton graph. One could however prioritise the discovery of MLACs by interleaving both searches. The modification we propose is to launch procedure *findMLAC* each time a sweep of the first level algorithm (i.e., procedure *findSLAC* of Algorithm 2) looking for SLACs has finished. The search of *findMLAC* is then initiated from states that were used as roots by the first level sweep and is bound to persistent states that have already been visited (i.e., not discovered by the last sweep performed). Let us denote by $\mathcal{R}_i$ the set of root states during the i^{th} sweep of the first level algorithm. After sweep i has terminated, procedure *findMLAC* will be launched to look for an MLAC including at least one persistent state of $\mathcal{R}_i$ and possibly some states of $\cup_{j \in \{0..i-1\}} \mathcal{R}_j$. Procedure *findMLAC* of Alg. 3 has to be modified in such a way that, after sweep i has terminated, $\mathcal{P}$ is initialised at l. 2 with $\mathcal{R}_i$. With this modification, it is guaranteed that an MLAC containing some persistent states $p_1 \in \mathcal{R}_{\hat{p}_1}, \ldots, p_n \in \mathcal{R}_{\hat{p}_n}$ will be reported once the m^{th} sweep has finished where $m = \max_{\{\hat{p}_1,\ldots,\hat{p}_n\}}$. In the following, we refer to $\mathsf{LTL\text{-}Sweep}^{\mathsf{off}}$ as the algorithm where *findMLAC* is followed by *findMLAC* and $\mathsf{LTL\text{-}Sweep}^{\mathsf{on}}$ as the version that interleaves the discovery of the two types of cycles.

5.2 Informed Search for MLACs

For now, there is no interaction between the two search procedures whereas procedure *findMLAC* could benefit from the experience of the previous search for SLACs. For example, in case the automaton graph does not have any accepting state, the search for MLACs could be avoided by just noticing this information during the previous step. More generally, we propose to build, as the search progresses, the progress graph as defined below.

Definition 7. *Let $G = (\mathcal{S}, \mathcal{E}, s_0, \mathcal{A})$ be an automaton graph with a progress mapping $\phi : \mathcal{S} \rightarrow \Phi$. The* progress graph *of G with ϕ is an automaton graph $G^\phi = (\mathcal{S}^\phi, \mathcal{E}^\phi, s_0^\phi, \mathcal{A}^\phi)$ defined by:*

- $\mathcal{S}^\phi = \{s_\alpha \mid \exists s \in \mathcal{S}$ *with* $\phi(s) = \alpha\}$ *;*
- $(s_\alpha, s_\beta) \in \mathcal{E}^\phi \Leftrightarrow \exists (a, b) \in \mathcal{E}$ *with* $\phi(a) = \alpha$ *and* $\phi(b) = \beta$ *;*
- $s_0^\phi = s_{\phi(s_0)}$ *; and*
- $s_\alpha \in \mathcal{A} \Leftrightarrow \exists s \in \mathcal{A}$ *with* $\phi(s) = \alpha$.

This graph provides information on the connectivity between progression layers of the state space graph and can be used to prune the search for MLACs. We can for instance avoid the visit of any state s such that, in the progress graph, $s_{\phi(s)}$ is not in a strongly connected component with an accepting state. It is a direct consequence of Def. 7 that this state can not be part of an accepting cycle.

5.3 Using the Progress Measure to Order States

Another direction we would like to pursue is to study whether the progress mapping could be useful in the definition of the order relation used to calculate the maximal persistent predecessor function. We arbitrarily chose in the current implementation to order states according to the bit vectors they are encoded in before insertion in the hash table. The sweep-line method works well for systems for which it is possible to derive a progress measure clustering the state space into multiple layers with few regress edges. On the basis of this assumption, we would like to experiment with an order relation that considers the progress value of states. Let s_1 and s_2 be two states such that $\phi(s_1) \sqsubset \phi(s_2)$ with s_2 being part of an accepting cycle. If the progress measure has the desired properties, then it is more likely that $s_1 \rightarrow^* s_2$ than $s_2 \rightarrow^* s_1$. In this situation it would then make sense that $s_2 >_\mathcal{S} s_1$ so that if it is indeed the case that $s_1 \rightarrow^* s_2$ and if both states are used as roots during the computation of the maximal persistent predecessor function, then state s_1 would not hide s_2 and the accepting cycle containing that state. It would then not be necessary to perform an iteration of the algorithm in order to delete s_1 to be able at the next iteration to detect the accepting cycle. For instance, with the graph of Fig. 3, this heuristic would imply to order states in such a way that $1 >_{\mathcal{S}} 2$ and $1 >_{\mathcal{S}} 3$ (since $\phi(2) \sqsubset \phi(1)$ and $\phi(3) \sqsubset \phi(1)$). We would then have $mpp(1) = (1, true)$ after the first iteration and the accepting cycle $1 \to 4 \to 5 \to 1$ would be detected without the need of deleting 2 and 3 from $\mathcal{P}$ and reiterating the search.

6 Experiments

We have implemented our algorithm in its two variants LTL-Sweep$^{\mathsf{off}}$ and LTL-Sweep$^{\mathsf{on}}$ on the ASAP verification platform [25], and experimented with it using DVE models from the BEEM database [21]. The 85 instances we selected have a number of states ranging from 100,000 to 10,000,000 states. Out of these 85 instances, 49 had an accepting cycles and the 36 remaining ones did not. We compared our algorithm to two LTL model checking algorithms: the classical NDFS algorithm [7] and the MAP algorithm [4]. We also compared it to the sweep-line algorithm Sweep from [18] designed for checking safety properties. As the full graph must be explored in the absence of an accepting cycle, the performance of Sweep served, in that context, as a baseline to assess the performance of LTL-Sweep: the latter cannot visit (or store) fewer states than Sweep. We used automatically generated progress measures for each model according to the generation process described in [9]. Each measure projects a state vector to some of its components (e.g., local variables, program counters) chosen after a preliminary exploration of the system. For the sake of clarity, we have selected a set of representative instances from our experiments with respect to several parameters (complexity of the model, size of the graph, and performance). However, for completeness, the reader may find all our experimental data in [10].

Our experimental data are reported in Table 1. We have separated instances for which the property analysed holds (top part of the table) from those containing an accepting cycle (bottom part). The first column provides information on each graph we analysed: the instance name, the number (in the BEEM database) of the analysed property and the number of states (st.) and edges (ed.) in the automaton graph[1]. Each entry in the table provides data for a single run, i.e., a triple (instance, property, algorithm): the peak number of states stored (first row) and the number of states visited (second row). For sweep-line based algorithms an entry also reports the number of persistent states once the search has terminated (third row). All these numbers are expressed as fractions of the number of states of the automaton graph. Finally, for instances exhibiting an accepting cycle, a small letter to the left of stored states indicates, for our algorithm, the type of cycle detected (S for an SLAC and M for an MLAC).

Our interpretation of the data first deals with graphs without an accepting cycle. We then discuss the models with falsified properties. Throughout this section all our comments dealing with LTL-Sweep apply to both versions of the algorithm: LTL-Sweep$^{\mathsf{off}}$ and LTL-Sweep$^{\mathsf{on}}$.

[1] We will not detail the models and their properties in this article but we invite the reader to consult this database online at `http://anna.fi.muni.cz/models/`.

Table 1. Experimental data

	NDFS	MAP	Sweep	LTL-Sweep^{on}	LTL-Sweep^{off}
Verified properties					
bopdp.3, prop. 4	1.000	1.000	0.074	0.074	0.106
1,703,192 st.	1.000	1.000	2.021	15.739	8.349
4,619,673 ed.	–	–	0.009	0.009	0.009
leader_filters.5, prop. 2	1.000	1.000	0.086	0.086	0.086
1,572,886 st.	2.000	14.462	1.000	2.000	2.000
4,319,565 ed.	–	–	0.000	0.000	0.000
lifts.6, prop. 2	1.000	1.000	0.012	0.012	0.012
998,570 st.	1.332	16.564	1.552	4.076	3.387
2,864,768 ed.	–	–	0.006	0.006	0.006
lup.3, prop. 2	1.000	1.000	0.330	0.336	0.336
2,346,373 st.	1.170	10.954	3.909	50.486	8.511
4,965,501 ed.	–	–	0.111	0.111	0.111
peterson.4, prop. 4	1.000	1.000	0.184	0.205	0.224
2,239,039 st.	1.500	11.546	5.322	103.883	25.443
11,449,204 ed.	–	–	0.046	0.046	0.046
pgm_protocol.8, prop. 4	1.000	1.000	0.043	0.045	0.143
3,069,399 st.	1.000	5.000	1.127	2.995	5.029
7,125,130 ed.	–	–	0.025	0.025	0.025
rether.6, prop. 2	1.000	1.000	0.069	0.121	0.175
6,046,531 st.	1.001	2.000	1.463	45.663	9.681
7,980,886 ed.	–	–	0.045	0.045	0.045
Falsified properties					
extinction.4, prop. 2	0.408	1.000	0.068	S 0.022	S 0.022
2,001,372 st.	0.817	2.585	1.000	0.115	0.115
6,856,693 ed.	–	–	0.000	0.000	0.000
iprotocol.4, prop. 4	0.006	0.614	0.109	M 0.032	M 0.123
8,214,324 st.	0.006	1.077	1.803	0.938	7.565
30,357,177 ed.	–	–	0.108	0.029	0.108
mcs.6, prop. 4	0.140	1.000	0.339	S 0.045	S 0.045
665,007 st.	0.279	8.289	3.414	0.183	0.183
3,283,155 ed.	–	–	0.003	0.000	0.000
plc.2, prop. 3	0.004	0.005	0.013	M 0.000	S 0.001
130,220 st.	0.004	0.006	1.051	0.009	0.053
210,710 ed.	–	–	0.013	0.000	0.000
rether.3, prop. 6	0.001	0.129	0.255	M 0.056	M 0.476
607,382 st.	0.001	0.193	1.612	0.413	15.660
991,098 ed.	–	–	0.040	0.007	0.040
synapse.1, prop. 3	0.059	0.103	0.393	S 0.219	S 0.219
159,888 st.	0.059	0.081	2.099	1.050	0.270
721,531 ed.	–	–	0.183	0.033	0.033

Instances without Accepting Cycle. On the criterion of explored states, MAP and LTL-Sweep, are in general incomparable although we found that, on the average, MAP have more stable performance. However, this observation is not surprising if we recall that the time complexity of MAP is $\mathcal{O}(|\mathcal{A}|^2 \cdot |\mathcal{S}|)$ while our algorithm works in $\mathcal{O}(|\mathcal{P}|^3 \cdot |\mathcal{S}|)$. The relative performance of both algorithms then depends on the number of accepting states and the quality of the progress measure that has an impact on the number of persistent states. Still, even in the presence of very few persistent states (e.g., instances bopdp.3, rether.6) our algorithm can explore a large number of states: it also depends on how the deletion transformation (ll. 6–7 of Alg. 3) succeeds in removing states from the set of persistent states $\mathcal{P}$ that procedure *findMLAC* will search for cycles on.

If we compare the two variants of our algorithm on the same criterion (explored states) we observe that LTL-Sweep$^{\text{off}}$ is generally faster than LTL-Sweep$^{\text{on}}$. With algorithm LTL-Sweep$^{\text{off}}$, procedure *findMLAC* is invoked only once with all persistent states discovered by the algorithm whereas with algorithm LTL-Sweep$^{\text{on}}$, the procedure is invoked with $\mathcal{R}_0, \mathcal{R}_1, \ldots$ ($\mathcal{R}_i$ being the set of root states during the i^{th} sweep of the top level algorithm looking for SLACs). Hence, the delete transformation is usually more successful in the off-line variant as it can potentially remove more states from $\mathcal{P}$ and perform fewer computations of the maximal persistent predecessor function. Stated in a different manner, it is better to perform a single invocation of *findMLAC* on a set S rather than partitioning S and then performing several invocations on each class of this partition. Averaged over all instances with no accepting cycle, LTL-Sweep$^{\text{on}}$ was 12.6 slower than NDFS while this number goes down to 5.3 with algorithm LTL-Sweep$^{\text{off}}$. If we now compare both algorithms to Sweep these ratios become 7.6 and 3.9.

From a different perspective, calling procedure *findMLAC* once with a large set naturally causes the algorithm to consume more memory (with respect to several calls with smaller sets). This is why the peak number of states stored observed with LTL-Sweep$^{\text{off}}$ is generally larger than with LTL-Sweep$^{\text{on}}$ (e.g., instances bopdp.3, pgm_protocol.8 or rether.6). For the on-line variant, the difference observed in stored states between Sweep and LTL-Sweep$^{\text{on}}$ is due to the way the *mpp* procedure processes: it does not really perform sweeps as algorithm Sweep does (i.e., visiting states layer-by-layer by increasing the progress value and then starting again a sweep from some persistent states) but each time it meets a persistent state s after executing a regress edge, it puts s in the priority queue, and then continues the search normally. Hence, the sweep-line moves back each time a persistent state is met. We plan to implement and experiment with both versions in a future version of our prototype. The negative observations we made on our algorithm regarding visited states must, however, be related to its lower memory usage. In most cases, Table 1 shows that the number of states stored of LTL-Sweep$^{\text{on}}$ and LTL-Sweep$^{\text{off}}$ equalled or at least approached the consumption of Sweep.

In order to compare algorithms on both visited and stored states we measured for each algorithm on a specific instance a score defined as the product of stored and visited states. This score indicates to which extent state revisits

Table 2. Instances for which an algorithm got the best score (i.e., minimised |Visited| · |Stored|)

	MAP	NDFS	LTL-Sweep^{on}	LTL-Sweep^{off}
36 instances without an accepting cycle	11.1%	44.4%	61.1%	36.1%
49 instances with an accepting cycle	34.6%	85.7%	53.06%	53.06%

are compensated by memory reduction, the lower score the better. We computed for each algorithm A the percentage of instances for which algorithm A got the smallest score. This data can be found in Table 2. Note that the sum of these percentages exceeds 100% since several algorithms can obtain the same best score. This for example occurs if the graph does not have any accepting state. The results indicate that the run time increase of LTL-Sweep is usually acceptable in that it is counterbalanced by an effective memory usage. Moreover, it appears that our algorithm obtained bad scores mainly on instances for which the sweep-line method is anyway not adapted. These include models such as peterson.4 or lup.3. Their graphs are composed of a single connected component and do not really exhibit progress. The only exception is model rether.6. As Table 1 shows, algorithm Sweep performs quite well on that model but the performance of LTL-Sweep^{on} and LTL-Sweep^{off} is poor.

Instances with Accepting Cycle(s). On most instances, NDFS is the algorithm performing the best, reporting an accepting cycle faster than its competitors. Even if we consider the number of states stored NDFS, Table 2 shows that NDFS is the clear winner. However, we found out some instances, for which LTL-Sweep outperformed both NDFS and MAP. Two such examples are extinction.4 and mcs.6. In both cases it happened that the SLAC reported by LTL-Sweep contained states close to the initial state (from the progress measure perspective) which possibly explains why the algorithm could terminate relatively early. In contrast, using NDFS, it is likely that this cycle would be discovered later since, by proceeding depth-first, the first states the algorithm backtracks from (launching the search for accepting cycles) are deeper in the graph and usually with a higher progress value. If we compare MAP and LTL-Sweep we again observe very different performances and there is no clear winner between the two. Relying in these two algorithms on an arbitrary order relation (comparison of bit state vectors) can also explain their unpredictable performances.

A comparison of the two variants of our algorithm reveals the impact of the type of accepting cycles found in the graph. If the graph only contains SLACs (or if contains MLACs including persistent states met after an SLAC is reported), then the number of states visited by LTL-Sweep^{off} is guaranteed to be fewer or equal than the one with LTL-Sweep^{on}. Indeed, in that case, LTL-Sweep^{on} will interleave between searches for SLACs and (useless) searches for MLACs whereas

LTL-Sweep$^{\text{off}}$ will postpone the latter and discover SLACs sooner. This, for example, explains the difference observed in visited states for instance synapse.1. If the graph has both types of cycles, looking for MLACs as soon as possible can be fruitful. This explains why LTL-Sweep$^{\text{on}}$ terminated faster on instance plc.2. Algorithm LTL-Sweep$^{\text{off}}$ could report an SLAC only during the last iterations. Finally, if the graph only has MLACs (e.g., for instances iprotocol.4 and rether.3), then we observe that LTL-Sweep$^{\text{on}}$ is usually much faster showing again the benefit of searching for MLACs as soon as possible.

7 Conclusion and Perspectives

We have introduced in this article an LTL model checking algorithm that can be used with the on-the-fly deletion of states performed by the sweep-line method. The major difficulty of designing such a combination stems from the algorithm the sweep-line method relies on. For reachability properties, the search uses a progress-based policy whereas state-of-the-art algorithms for LTL model checking rely on a depth-first search that is best suited for cycle detection.

Our algorithm LTL-Sweep is made of two distinct building blocks each one being dedicated to a specific kind of accepting cycle. For accepting cycles containing states with the same progress value (i.e., SLACs), we simply adapt the basic Sweep algorithm to perform nested depth-first searches on layers of states. If the accepting cycle spans several layers (i.e., MLACs) we use a variation of the MAP algorithm in order to look for accepting cycles containing persistent states. The choice of MAP originates from its independence from any search order policy which makes it more easily compatible with the sweep-line method. Since the two searches are independent, we propose two versions of our algorithm. The off-line version first tries to look for SLACs and then for MLACs if the first search did not detect an acceptable cycle. The on-line variation, interleaves both searches and is thus able to report existing MLACs faster. We have implemented th algorithms in the ASAP verification platform and compared it with other LTL model checking algorithms. The conclusions we drew from these experiments are:

- LTL-Sweep uses roughly the same amount of memory as Sweep while being 4 times slower than Sweep in its off-line version and 8 times slower in its on-line version;
- When the run time increases it is usually compensated by a low memory consumption that keeps LTL-Sweep competitive with other algorithms ;
- MAP and LTL-Sweep are in general incomparable: their performance can considerably vary according to the model analysed ;
- LTL-Sweep could in general not compete with NDFS for fast accepting cycles discovery but could terminate earlier for some specific models ;
- Both the on-line and off-line versions have their pros and cons: the former can usually report accepting cycles earlier while the latter usually visits fewer states in the absence of accepting cycle. This suggests that each could be useful at different stages of the verification process.

We identified some possible rooms for improvement. First, data could be exchanged between the two procedures to optimise the search for MLACs. We propose to maintain as the search progresses, a progression graph that summarises the connections between the layers and that could be useful to prune the search for MLACs. Second, we plan to investigate to which extent the progress measure could be used to order states efficiently when looking for MLACs.

Our experiments showed that our algorithm achieves a good memory reduction if we take algorithm Sweep as a reference. This can, however, be penalised by an increase of the execution time. One direction for future research would be to design a parallel version of LTL-Sweep to address this issue. As algorithm MAP, and unlike NDFS, LTL-Sweep does not use an inherently sequential nested depth-first search this motivates this research direction. Moreover, the two components LTL-Sweep is made of are relatively independent: the search for SLACs and the search for MLACs can be performed in parallel. Several issues still have to be tackled. For instance, we have to take care that such a combination does not cancel the sweep-line reduction by letting processes explore different layers of states (while the sequential algorithm always keep a single layer in memory at a time). On the other hand, using a global "clock" to determine how processes explore the system would not necessarily yield a good time reduction.

References

1. Barnat, J., Brim, L., Šimeček, P., Weber, M.: Revisiting Resistance Speeds Up I/O-Efficient LTL Model Checking. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 48–62. Springer, Heidelberg (2008)
2. Behrmann, G., Larsen, K.G., Pelánek, R.: To Store or Not to Store. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 433–445. Springer, Heidelberg (2003)
3. Billington, J., Gallasch, G., Kristensen, L.M., Mailund, T.: Exploiting Equivalence Reduction and the Sweep-Line Method for Detecting Terminal States. IEEE Transactions on SMC - Part A 34(1), 23–38 (2004)
4. Brim, L., Černá, I., Moravec, P., Šimša, J.: Accepting Predecessors Are Better than Back Edges in Distributed LTL Model-Checking. In: Hu, A.J., Martin, A.K. (eds.) FMCAD 2004. LNCS, vol. 3312, pp. 352–366. Springer, Heidelberg (2004)
5. 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)
6. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. The MIT Press (1999)
7. Courcoubetis, C., Vardi, M.Y., Wolper, P., Yannakakis, M.: Memory Efficient Algorithms for the Verification of Temporal Properties. In: Clarke, E., Kurshan, R.P. (eds.) CAV 1990. LNCS, vol. 531, pp. 233–242. Springer, Heidelberg (1991)
8. Couvreur, J.-M.: On-the-Fly Verification of Linear Temporal Logic. In: Wing, J.M., Woodcock, J. (eds.) FM 1999. LNCS, vol. 1708, pp. 253–271. Springer, Heidelberg (1999)
9. Evangelista, S., Kristensen, L.M.: Search-Order Independent State Caching. In: Jensen, K., Donatelli, S., Koutny, M. (eds.) Transactions on Petri Nets and Other Models of Concurrency IV. LNCS, vol. 6550, pp. 21–41. Springer, Heidelberg (2010)

10. Evangelista, S., Kristensen, L.M.: Hybrid On-the-fly LTL Model Checking with the Sweep-Line Method. Technical report, Université Paris 13 (2012), http://www-lipn.univ-paris13.fr/ evangelista/biblio-sami/doc/sweep-ltl.pdf
11. Gallasch, G.E., Billington, J., Vanit-Anunchai, S., Kristensen, L.M.: Checking Safety Properties On-the-fly with the Sweep-Line Method. STTT 9(3-4), 371–392 (2007)
12. 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)
13. 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: CPN, pp. 19–38 (2004)
14. Godefroid, P., Holzmann, G.J., Pirottin, D.: State-Space Caching Revisited. In: Probst, D.K., von Bochmann, G. (eds.) CAV 1992. LNCS, vol. 663, pp. 178–191. Springer, Heidelberg (1993)
15. 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)
16. Holzmann, G., Peled, D., Yannakakis, M.: On Nested Depth First Search. In: SPIN 1996 (1996)
17. Holzmann, G.J.: Tracing Protocols. AT&T Technical J. 64(10), 2413–2434 (1985)
18. 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)
19. 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)
20. Mailund, T., Westergaard, M.: Obtaining Memory-Efficient Reachability Graph Representations Using the Sweep-Line Method. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 177–191. Springer, Heidelberg (2004)
21. Pelánek, R.: BEEM: Benchmarks for Explicit Model Checkers. In: Bošnački, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 263–267. Springer, Heidelberg (2007)
22. Schmidt, K.: LoLA A Low Level Analyser. In: Nielsen, M., Simpson, D. (eds.) ICATPN 2000. LNCS, vol. 1825, pp. 465–474. Springer, Heidelberg (2000)
23. Vanit-Anunchai, S., Billington, J., Gallasch, G.E.: Analysis of the Datagram Congestion Control Protocols Connection Management Procedures using the Sweep-line Method. STTT 10(1), 29–56 (2008)
24. Vardi, M.Y., Wolper, P.: An Automata-Theoretic Approach to Automatic Program Verification. In: LICS 1986, pp. 332–344 (1986)
25. 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)

Safety Slicing Petri Nets[⋆]

Astrid Rakow

Universität Oldenburg

Abstract. We define a safety slice as a subnet of a marked Petri net Σ that approximates Σ's temporal behavior with respect to a set of interesting places Crit. This safety slice can be used to verify and falsify stutter-invariant linear-time safety properties when Crit is the set of places referred to by the safety property. By construction it is guaranteed that the safety slice's state space is at most as big as that of the original net. Results on a benchmark set demonstrate effective reductions on several net instances. Therefore safety slicing as a net preprocessing step may achieve an acceleration for model checking stutter-invariant linear-time safety properties.

Slicing is a technique to syntactically reduce a model in such a way that at best the reduced model contains only those parts that may influence the property the model is analyzed for. It originated as a method for program debugging[16] but has found applications in many other domains. We here introduce a slicing approach tailored to Petri nets as a means to alleviate the state space explosion problem for model checking Petri nets. We present a safety slicing algorithm that determines what parts of a marked Petri net Σ can be sliced away (i.e. discarded) so that the remaining net is equivalent to the original w.r.t. a stutter-invariant linear-time safety property ψ. The remaining net is called *safety slice* Σ' and is built for a so called *slicing criterion* Crit.

We will formally show that safety slices allow for verification and falsification of stutter-invariant linear-time safety properties. Hence when one wants to examine whether a marked Petri net Σ satisfies a stutter-invariant linear-time safety property ψ, the safety slice may be examined instead. The safety slice may have a substantially smaller state space, yielding an acceleration in model checking. As the safety slicing algorithm is linear in the size of the net Σ (not its state space!), even when slicing does not accelerate model checking, the overhead will usually be insignificant.

In [11] we presented a more conservative slicing algorithm. There we introduced $\mathrm{CTL}^*_{\text{-X}}$ slices, which preserve $\mathrm{CTL}^*_{\text{-X}}$ properties assuming a weak fairness assumption on the original net Σ. By definition a safety slice for Crit is a subnet of the $\mathrm{CTL}^*_{\text{-X}}$ slice for Crit, so that safety slicing offers the potential to generate smaller slices than the $\mathrm{CTL}^*_{\text{-X}}$ preserving algorithm but sacrifices the preservation of liveness properties.

[⋆] This work is partly supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems".

Outline. In the next section we introduce the basic notions of this paper. As an introduction to safety slicing, we sketch the algorithm for CTL$^*_{\text{-X}}$ slicing in Sect. 2. In Sect. 3 we refine this algorithm to the more aggressive safety slicing algorithm and formally prove that a safety slice preserves stutter-invariant linear-time safety properties. We then discuss related work in Sect. 4 and present evaluation results in Sect. 5. We outline future work and conclude the paper in Sect. 6.

1 Basic Definitions

The developed slicing techniques aim to alleviate the model checking problem for temporal logic specifications. Temporal logic allows to specify properties referring to the system behavior over time [1]. In [11] we presented a first slicing approach preserving any property expressed in CTL$^*_{\text{-X}}$, i.e. CTL* (see e.g. [1]) without next-time operator. In this paper however we focus on a less conservative algorithm that only preserves stutter-invariant linear-time safety properties, which are formally introduced in this section.

Sets and Sequences. For a set X we denote the union of finite and infinite words over X, $X^* \cup X^\omega$, as X^∞. For a finite sequence $\gamma = x_1 x_2 ... x_n \in X^\infty$, $|\gamma|$ is n, the length of γ. If γ is infinite, $|\gamma| = \infty$. $\gamma(i)$ denotes the i-th element, $1 \leq i < |\gamma| + 1$, and γ^i denotes the suffix of γ that truncates the first i positions of γ, $0 \leq i < |\gamma| + 1$. $\gamma' = \mathit{proj}_{X'}(\gamma)$ denotes the projection of γ to $X' \subseteq X$, i.e. γ' is derived from γ by omitting every $x_i \in X \setminus X'$. Two sequences γ_1 and γ_2 are stutter-equivalent iff $\mathit{unstutter}(\gamma_1) = \mathit{unstutter}(\gamma_2)$, where *unstutter* merges finitely many successive repetitions of the same sequence element into one. So $\gamma_1 = x_1 x_2 x_3$ and $\gamma_2 = x_1 x_2 x_3 x_3 x_3$ are stutter-equivalent whereas $\gamma_3 = x_1 x_2 x_3 x_3 ....$ is not stutter-equivalent to γ_1 or γ_2. We extend the functions *unstutter* and *proj* to sets of sequences in the usual way.

Petri Net Definitions. A *Petri net* N is a triple (P, T, W) where P and T are disjoint sets and $W : ((P \times T) \cup (T \times P)) \to \mathbb{N}$. An element p of P is called a *place* and $t \in T$ is called a *transition*. The function W defines weighted arcs between places and transitions. A Petri net is *finite* iff P and T are finite sets. In this paper we consider finite Petri nets only.

The *preset* of $p \in P$ is ${}^\bullet p = \{t \in T \mid W(t, p) > 0\}$ and the *postset* of p is $p^\bullet = \{t \in T \mid W(p, t) > 0\}$, analogously ${}^\bullet t$ and $t^\bullet$ are defined.

A *marking* of a net N is a function $M : P \to \mathbb{N}$, which assigns a number of *tokens* to each place. With a given order on the places, $p_1, ..., p_n$, M can be represented as a vector in $\mathbb{N}^{|P|}$, where the i-th component is $M(p_i)$.

A transition $t \in T$ is *enabled* at marking M, $M[t\rangle$, iff $\forall p \in {}^\bullet t : M(p) \geq W(p, t)$. If t is enabled it can *fire*. The firing of t generates a new marking M', $M[t\rangle M'$, which is determined by the *firing rule* as $M'(p) = M(p) - W(p, t) + W(t, p), \forall p \in P$. The definition of $[\rangle$ is extended to transition sequences σ as follows. A marking M always enables the empty firing sequence ε and its firing generates M. M enables a transition sequence σt, $M[\sigma t\rangle$, iff $M[\sigma\rangle M'$ and $M'[t\rangle$. If $M[\sigma\rangle$, the transition sequence σ is called a *firing sequence* from M. The *effect* of σ on a place $p \in P$, $\Delta(\sigma, p) \in \mathbb{Z}$, is defined by $\Delta(\varepsilon, p) = 0$ and $\Delta(\sigma t, p) = \Delta(\sigma, p) + W(t, p) - W(p, t)$.

Given a firing sequence $\sigma = t_1 t_2 ...$ with $M[t_1\rangle M_1[t_2\rangle M_2 ...$, the sequence $M M_1 M_2 ...$ is called the *marking sequence* of σ from M, $\mathcal{M}(M, \sigma)$. As $\mathcal{M}(M, \sigma)|_{\tilde{P}} := M|_{\tilde{P}} M_1|_{\tilde{P}} M_2|_{\tilde{P}} ...$ we denote the elementwise restriction of $\mathcal{M}(M, \sigma)$ to $\tilde{P} \subseteq P$. A marking M is called *final* iff there is no nonempty firing sequence from M. A firing sequence σ from M is *maximal* iff either σ is of infinite length or σ generates a final marking. A marking sequence $\mathcal{M}(M, \sigma)$ is *maximal* iff σ is a maximal firing sequence. By convention, we regard a finite maximal marking sequence μ as equivalent to the infinite marking sequence μ' that repeats the final marking of μ infinitely often.

A Petri net $\Sigma = (N, M_{\mathsf{init}})$ with a designated initial marking M_{init} is called a *marked Petri net*. A marking of Σ is *reachable* if there is a firing sequence from M_{init} that generates M, $M_{\mathsf{init}}[\sigma\rangle M$. The set of reachable markings of Σ is denoted as $[M_{\mathsf{init}}\rangle$. In the following we will use N synonymous with (P, T, W) and Σ synonymous with (N, M_{init}) and subscripts carry over to the components, i.e. we use N_1 synonymous with (P_1, T_1, W_1). Moreover, we denote a marking generated by firing $\sigma \in T^*$ from the initial marking M_{init} as M_σ.

Transition System of a Petri Net. A transition system is one standard model to describe the behavior of system. We use transition systems here as an intermediate: We define stutter-invariant linear-time safety properties on transition systems and introduce TS_Σ the transition system of a Petri net Σ.

Definition 1 (Transition System). *A* transition system TS *with initial state is a tuple* $(S, Act, R, AP, L, s_{\mathsf{init}})$ *where*

- S *is the set of states,*
- Act *is a set of actions,*
- $R \subseteq S \times Act \times S$ *is the transition relation with* $\forall s \in S : \exists \alpha \in Act : \exists s' \in S : (s, \alpha, s') \in R,$
- AP *is a set of atomic propositions,*
- $L : S \to 2^{AP}$ *is a state labelling function.*
- s_{init} *is a designated initial state of* TS

Note, that a transition system according to Def. 1 has no terminal states, i.e. every state has a successor via R.

A *path* π from a state s is a sequence of states such that $\pi(1) = s$ and $\forall i, 1 \leq i < |\pi| + 1 : \exists \alpha_i \in Act : (\pi(i), \alpha_i, \pi(i+1)) \in R$. A *trace* ϑ of a path π is $L(\pi) := L(\pi(0))L(\pi(1))...$. $\mathsf{Traces}_{TS}(s)$ denotes the set traces of (TS, s). $\mathsf{Traces}_{TS(s),\mathsf{fin}}$ and $\mathsf{Traces}_{TS(s),\mathsf{inf}}$ denote the sets of finite and infinite traces of paths of TS starting at s. We use TS and $(S, Act, R, AP, L, s_{\mathsf{init}})$ synonymously.

Next we define the transition system of a Petri net. We denote the set of places a temporal logic property φ refers to as *scope*(φ).

The behavior of a marked Petri net Σ can be captured by a transition system TS_Σ. The reachable markings of Σ are the states of TS_Σ. If $M[t\rangle M'$, then there is also a transition from state M to M' via action t in TS_Σ. Hence a path $\mu = M_0 M_1 \dots M_n$ in TS_Σ corresponds to the firing sequence $\sigma = t_1 t_2 \dots t_n$ with marking sequence $\mathcal{M}(M_0, \sigma) = M_0 M_1 \dots M_n$.

A maximal firing sequence may be finite and thus generate a final marking that does not enable any transition, but every state M of TS_Σ has to have at least one successor. By convention we therefore introduce a new action symbol τ and define that a final marking M reaches itself via τ. By this extension any marking sequence corresponds to a path and any maximal firing sequence corresponds to an infinite path.

Definition 2 (TS_Σ). *Given a Petri net Σ and a set of atomic propositions $AP_\Sigma \subseteq P \times \mathbb{N}$, the transition system $TS_\Sigma(\Sigma, AP_\Sigma)$ is the tuple $(S_\Sigma, Act_\Sigma, R_\Sigma, AP_\Sigma, L_\Sigma, M_{\mathsf{init}})$ with*

– $S_\Sigma = [M_{\mathsf{init}}\rangle$,
– $Act_\Sigma = T \uplus \{\tau\}$,
– $R_\Sigma = \{(M, t, M') \mid M, M' \in [M_{\mathsf{init}}\rangle \wedge t \in T \wedge M[t\rangle M'\} \cup \{(M, \tau, M) \mid M \in [M_{\mathsf{init}}\rangle \wedge \forall t \in T : \neg M[t\rangle\}$
– $L_\Sigma = \{(M \mapsto A) \mid M \in S_\Sigma \wedge A = \{(p, x) \in AP_\Sigma \mid M(p) = x\}\}$.

Stutter-invariant Safety Properties. In the following we introduce the notion of stutter-invariance and characterize safety properties following [1]. For the following we fix a set of atomic propositions AP.

Linear-time properties express constraints on infinite paths or more precisely on infinite traces.

Definition 3 (LT Property). *A linear-time property (LT Property) over the set of atomic propositions AP is a subset of $(2^{AP})^\omega$.*

$TS, s \models \mathcal{P}$ means that all infinite traces starting from s satisfy $\mathcal{P}$. Note, that by Def. 1, a finite trace can always be extended to an infinite trace, since our transition systems have no terminal states. We formally define $TS, s \models \mathcal{P}$ by:

Definition 4 (Satisfaction Relation for LT Properties). *Let $\mathcal{P}$ be an LT property over AP and TS a transition system.*

$TS, s \models \mathcal{P} \ \Leftrightarrow\ \mathsf{Traces}_{TS,\mathsf{inf}}(s) \subseteq \mathcal{P}$.

Stutter-invariant linear-time properties do not distinguish between stutter-equivalent traces.

Definition 5 (Stutter-invariant [6,8]). *Let ϑ and ϑ_2 be in $(2^{AP})^\omega$. A property $\mathcal{P}_{stutter} \subseteq (2^{AP})^\omega$ is stutter-invariant if whenever ϑ and ϑ_2 are stutter-equivalent then either both ϑ and ϑ_2 satisfy $\mathcal{P}_{stutter}$ or both violate $\mathcal{P}_{stutter}$.*

We are now ready to define safety properties. A safety property can be thought of as stating that nothing bad will eventually happen [6]. When a safety property $\mathcal{P}_{safe}$ is violated, a finite prefix is sufficient to expose the behavior forbidden by $\mathcal{P}_{safe}$. Formally a safety property is an LT property that, if any possible infinite trace ϑ violates $\mathcal{P}_{safe}$, it has a bad finite prefix ϑ_{pref}, such that any other (possible) trace $\tilde{\vartheta}$ with prefix $\vartheta_{\mathsf{pref}}$ also violates $\mathcal{P}_{safe}$.

Definition 6 (safety property). *An LT property $\mathcal{P}_{safe}$ over AP is a safety property if for all words $\vartheta \in (2^{AP})^\omega \setminus \mathcal{P}_{safe}$ there is a finite prefix ϑ_{pref} of ϑ such that*

$$\mathcal{P}_{safe} \cap \{\tilde{\vartheta} \in (2^{AP})^\omega \mid \vartheta_{pref} \text{ is a prefix of } \tilde{\vartheta}\} = \emptyset.$$

Any such prefix ϑ_{pref} is called a bad prefix for $\mathcal{P}_{safe}$. The set of all bad prefixes for $\mathcal{P}_{safe}$ is denoted by $BadPref(\mathcal{P}_{safe})$.

This definition allows us to derive a satisfaction relation referring to the finite behaviors of TS only. A transition system satisfies a safety property $\mathcal{P}_{safe}$ from state s iff the set of finite traces from s does not have a bad prefix, that means nothing bad happens starting from s.

Proposition 7 (Satisfaction Relation for Safety Properties [1]). *For a transition system TS and safety property $\mathcal{P}_{safe}$ it holds that*
$TS, s \models \mathcal{P}_{safe}$ *if and only if* $\mathsf{Traces}_{TS,\mathsf{fin}}(s) \cap BadPref(\mathcal{P}_{safe}) = \emptyset$.

The fact that satisfiability of safety properties can be characterized by the finite behaviors of TS will allow us to define more effective reductions for safety preserving slicing than for CTL$^*_{\text{-x}}$ slicing, which also preserves liveness properties.

If we interpret in the following a temporal property φ on a Petri net Σ, we always assume that the set of atomic propositions of φ is contained in AP_Σ, the set of atomic propositions of TS_Σ.

2 CTL$^*_{\text{-x}}$ Slicing

If we want to observe the behavior of Σ w.r.t. a set of places Crit –which may for instance be the set of places a CTL$^*_{\text{-x}}$ property φ refers to–, the behavior is dependent on other parts of the net. Following these dependencies backwards starting from Crit will give us the slice. The basic idea for our Petri net slicing algorithms is to define the dependencies based on the locality property of Petri nets: The token count of a place p is determined by the firings of incoming and outgoing transitions of p. Whether such a transition can fire, depends on the token count of its input places.

If we want to observe the marking on a set of places Crit, we can iteratively construct a subnet $\hat{\Sigma} = (\hat{P}, \hat{T}, \hat{W}, \hat{M}_{\mathsf{init}})$ of Σ by taking all incoming and outgoing transitions of a place $p \in \hat{P}$ together with their input places, starting with $\hat{P} = \mathsf{Crit}$. The subnet $\hat{\Sigma}$ certainly captures every token flow of Σ that influences the token count of a place $p \in \mathsf{Crit}$.

We refine the above construction by distinguishing between *reading* and *non-reading* transitions. A reading transition of places R cannot change the token count of any place in R. We formally define t to be a reading transition of $R \subseteq P$ iff $\forall p \in R : W(p,t) = W(t,p)$. If t is not a reading transition of R, we call t a non-reading transition of R. Let us now iteratively build a subnet $\Sigma' = (P', T', W', M'_{\mathsf{init}})$ by taking all non-reading transitions of a place $p \in P'$ together with their input places, starting with $P' = \mathsf{Crit}$.

Definition 8 (CTL$^*_{\text{-X}}$ slice, $slice_{CTL^*_{\text{-X}}}$). *Let Σ be a marked Petri net and $\mathit{Crit} \subseteq P$ a non-empty set, called* slicing criterion*. The following algorithm constructs $slice_{CTL^*_{\text{-X}}}(\Sigma, \mathit{Crit})$ of Σ for the slicing criterion Crit.*

```
generateSlice($\Sigma$, $Crit$){
    $T'$, $P_{done} := \emptyset$;
    $P' := Crit$;
    while ( $\exists p \in (P' \setminus P_{done})$ ) {
        Let $p$ be a place in $P' \setminus P_{done}$;
        while ( $\exists t \in (({}^\bullet p \cup p^\bullet) \setminus T') : W(p,t) \neq W(t,p)$ ) {
            Let $t$ be a transition in $\{t \in (({}^\bullet p \cup p^\bullet) \setminus T') \mid W(p,t) \neq W(t,p)\}$;
            $P' := P' \cup {}^\bullet t$;
            $T' := T' \cup \{t\}$; }
        $P_{done} := P_{done} \cup \{p\}$; }
    return ($P', T', W|_{(P',T')}, M_{\text{init}}|_{P'}$); }
```

The algorithm always terminates and always determines a subnet $slice_{CTL^*_{\text{-X}}}(\Sigma, \mathsf{Crit})$ for any given slicing criterion Crit. Though, the slice may equal the original net Σ. If $\mathsf{Crit} \subseteq \mathsf{Crit}'$, $slice_{CTL^*_{\text{-X}}}(\Sigma, \mathsf{Crit})$ is a subnet of $slice_{CTL^*_{\text{-X}}}$ (Σ, Crit'). Figure 1 illustrates the effect of `generateSlice`.

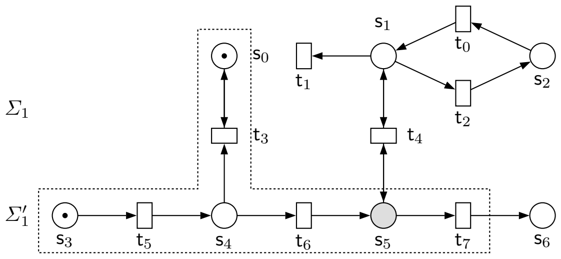


Fig. 1. Slicing a Petri net. The original net Σ_1 and its slice $\Sigma'_1 = slice_{CTL^*_{\text{-X}}}(\Sigma_1, \{s5\})$.

The slice $slice_{CTL^*_{\text{-X}}}(\Sigma, \mathsf{Crit})$ may be smaller than $\hat{\Sigma}$, the subnet constructed without considering reading transitions. Even for certain strongly connected nets the algorithm `generateSlice` might produce a slice Σ' that is smaller than Σ, whereas $\hat{\Sigma}$ for a strongly connected net is always equals to Σ.

In [11] it was shown, that it holds

$$\Sigma \models \varphi \textit{ fairly w.r.t. } T' \Leftrightarrow slice_{CTL^*_{\text{-X}}}(\Sigma, \mathsf{Crit}) \models \varphi$$

for a CTL$^*_{\text{-X}}$ formula φ, $scope(\varphi) \subseteq \mathsf{Crit}$ and where T' is the set of transitions of $slice_{CTL^*_{\text{-X}}}(\Sigma, \mathsf{Crit})$. Basically, a firining sequence σ is fair w.r.t. T', if σ is either maximal or σ is infinite and if σ eventually permanently enables a $t' \in T'$, a transition $t \in T'$ will be fired infinitely often– t may or may not equals t'. We refer the interested reader to [11] for a formal definition. $\Sigma \models \varphi$ *fairly w.r.t.* T' holds if all fair firing sequences of Σ –more precisely, their corresponding traces–

satisfy φ. This kind of weak fairness assumption on Σ guarantees progress within its slices subnet and is necessary when studying liveness properties.

The CTL$^*_{\text{-x}}$ slicing algorithm is fairly conservative. Assuming a very weak fairness assumption on Σ it approximates the temporal behavior quite accurately by preserving all CTL$^*_{\text{-x}}$ properties. For safety slicing we focus on the preservation of stutter-invariant linear-time safety properties only. So we need to preserve the temporal behavior of Σ less accurately. This allows us to define a more aggressive slicing algorithm that may generate smaller slices than the CTL$^*_{\text{-x}}$ slicing algorithm. In the next section we will first characterize in which respect we have more freedom in capturing Σ's behavior, and then formulate the safety slicing algorithm.

3 Safety Slicing

In this section we will develop the safety slicing algorithm which preserves stutter-invariant linear-time safety properties.

The Ease of Slicing for Safety Properties. The reason why the slicing algorithm can be more aggressive for safety properties is due to the fact that satisfiability of safety properties can already be determined inspecting finite prefixes of traces of TS_Σ. A transition system satisfies a safety property $\mathcal{P}_{safe}$ iff its set of finite traces does not have a bad prefix (c.f. Prop. 7). Two transition systems satisfy the same stutter-invariant safety-properties if their sets of finite paths are stutter-equivalent:

Proposition 9. *Let TS_1 and TS_2 be two transition systems with the same set of atomic propositions, $AP_1 = AP_2$. Let $\mathcal{P}_{safe} \subseteq (2^{AP_1})^\omega$ be a stutter-invariant safety property.*

If $unstutter(\mathsf{Traces}_{TS_1,\mathsf{fin}}(s_{\mathsf{init}1}))=unstutter(\mathsf{Traces}_{TS_2,\mathsf{fin}}(s_{\mathsf{init}2}))$,
then $TS_1, s_{\mathsf{init}1} \models \mathcal{P}_{safe}$ if and only if $TS_2, s_{\mathsf{init}2} \models \mathcal{P}_{safe}$.

Proof. We first show that $TS_1, s_{\mathsf{init}1} \models \mathcal{P}_{safe}$ implies $TS_2, s_{\mathsf{init}2} \models \mathcal{P}_{safe}$. Let us assume that $TS_1, s_{\mathsf{init}1} \models \mathcal{P}_{safe}$. Let ϑ_2 be a finite trace of TS_2 from $s_{\mathsf{init}2}$. By assumption, TS_1 has a stutter-equivalent trace ϑ_1, $unstutter(\vartheta_1) = unstutter(\vartheta_2)$. Since $TS_1, s_{\mathsf{init}1} \models \mathcal{P}_{safe}$ and since there are by Def. 1 no terminal states in TS_1, ϑ_1 is the prefix of an infinite trace $\vartheta_1\vartheta_{suf}$ that satisfies $\mathcal{P}_{safe}$. Since ϑ_1 and ϑ_2 are stutter-equivalent, $\vartheta_2\vartheta_{suf} \models \mathcal{P}_{safe}$. This implies that $\vartheta_2 \notin BadPref(\mathcal{P}_{safe})$. By Prop. 7 it follows that $TS_2, s_{\mathsf{init}2} \models \mathcal{P}_{safe}$.

It follows analogously that $TS_2, s_{\mathsf{init}2} \models \mathcal{P}_{safe} \Rightarrow TS_1, s_{\mathsf{init}1} \models \psi_{safe}$. □

Because marking sequences of Σ correspond to paths of TS_Σ, it follows that two Petri nets satisfy the same stutter-invariant safety properties if their (sets of) finite firing sequences generate (sets of) stutter-equivalent marking sequences.

Building the Safety Slice. The basic idea for constructing a *safety slice* is to build a slice for a set of places Crit by taking all non-reading transitions connected to

Crit and all their input places, so that we get the exact token count on Crit. But for all other places we are more relaxed: We iteratively take only transitions that increase the token count on places in P' and their input places (c.f. Def. 10). Intuitively, that way we can run any "sliced" firing sequence of Σ on its safety slice Σ' such that we have in Σ' at least as many tokens on the places and the same token count on Crit. This guarantees that we capture "enough" behavior of Σ within Σ'. Also we will see that every firing sequence of Σ' is also a firing sequence of Σ. Hence the slice does not expose too much behavior.

Definition 10 (safety slice, $slice_S$). *Let Σ be a marked Petri net and let $\mathsf{Crit} \subseteq P$ be the slicing criterion. The* safety slice *of Σ for slicing criterion Crit, $slice_S(\Sigma, \mathsf{Crit})$, is the subnet generated by the following algorithm.*

```
generateSafetySlice($\Sigma$, $\mathsf{Crit}$){
    $T' :=  \{t \in T \mid \exists p \in \mathsf{Crit} : W(p,t) \neq W(t,p)\}$;
    $P' :=  {}^\bullet T' \cup \mathsf{Crit}$;
    $P_{\mathsf{done}} := \mathsf{Crit}$;
    while ( $\exists p \in (P' \setminus P_{\mathsf{done}})$ ) {
        Let $p$ be a place in $P' \setminus P_{\mathsf{done}}$;
        while ( $\exists t \in ({}^\bullet p \setminus T') : W(p,t) < W(t,p)$ ) {
            Let $t$ be a transition in $\{t \in ({}^\bullet p \setminus T') \mid W(p,t) < W(t,p)\}$;
            $P' := P' \cup {}^\bullet t$;
            $T' := T' \cup \{t\}$; }
        $P_{\mathsf{done}} := P_{\mathsf{done}} \cup \{p\}$; }
    return ($P', T', W|_{(P',T')}, M_{\mathsf{init}}|_{P'}$); }
```

This safety slice allows to verify and falsify linear-time stutter-invariant safety properties.

Figure 2 illustrates the effect of `generateSafetySlice`. Whereas $slice_S(\Sigma_2, \{s_6\})$ does not contain the transition t_4, the CTL* slice $slice_{CTL^*_{-X}}(\Sigma_2, \{s6\})$ includes it, as t_4 can decrease the token count on s_7.

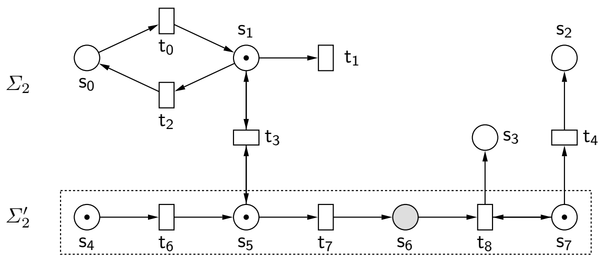


Fig. 2. Slicing a Petri net for safety. The original net Σ_2 and its slice $\Sigma_2' = slice_S(\Sigma_2, \{s6\})$.

Figure 1 shows the $slice_{CTL^*_{-X}}(\Sigma, \{s_5\})$. The safety slice $slice_S(\Sigma, \{s_5\})$ is smaller than $slice_{CTL^*_{-X}}(\Sigma, \{s_5\})$. It omits t_3 and s_0. Let us consider the liveness property $\mathbf{AF}(s_5, 1)$, meaning that eventually s_5 is marked with one token. Whereas the safety slice eventually places a token on s_5, the original net and also $slice_{CTL^*_{\text{-X}}}(\Sigma, \{s_5\})$ may not, because they may fire t_3. That is, $slice_S(\Sigma, \{s_5\}) \models \varphi$, whereas Σ fairly $\not\models \varphi$. Hence the safety slice does not preserve liveness properties.

Impact of Slicing. Let us consider the size of Σ's state space in terms as reachable states and state transitions as an indicator of the impact of slicing, as it usually has a strong influence on time and space needed for model checking.

If slicing removes dead subnets, in which never any transition is enabled, the effect on the state space is void. In contrast, if slicing removes parts of the net that expose highly concurrent behavior, the savings may be huge.

For an example consider the net in Fig. 2, but let us change its initial marking on s_1 to (a) $M_{\mathsf{init}}(s_1) = 0$, (b) $M_{\mathsf{init}}(s_1) = 1$ as in Fig. 2 and (c) $M_{\mathsf{init}}(s_1) = 3$. The safety slice is the same in all three cases with 9 reachable states and 11 state transitions. The state space sizes of the unsliced nets are (a) $(18, 45)$, (b) $(54, 197)$ and (c) $(180, 822)$ given as pairs of (states, state transitions).

Hence, whether a net is reducible depends on the model structure, whereas the impact of slicing depends on the system dynamics. As the dynamics is difficult to predict by just studying the model structure, the impact of slicing is difficult to predict as well.

3.1 Proving Safety Slice's Properties

To show that the safety slice preserves indeed stutter-invariant safety properties, it suffices to show that the sets of finite firing sequences of Σ and Σ' generate stutter-equivalent traces, as we have seen.

Correspondence of Firing Sequences. We first show the correspondence of firing sequences. We will show that for a given firing sequence σ of Σ we can fire the projected firing sequence $proj_{T'}(\sigma)$ on the safety slice Σ'. We can omit transitions in $T \setminus T'$, since they do not increase the token count of any place in P', so the token count on all places will be at least as high as it is on Σ firing σ. Further, every firing sequence of a safety slice Σ' is a firing sequence of Σ.

Firing Sequences, Marking Sequences, Traces. We then show that corresponding firing sequences σ and σ' generate corresponding markings, $M_\sigma|_{\mathsf{Crit}} = M'_{\sigma'}|_{\mathsf{Crit}}$. We consider markings M of Σ and M' of Σ' as correspondent iff they coincide on Crit, because we assume that $scope(\varphi) \subseteq \mathsf{Crit}$. It thus follows that two marking sequences that are stutter-equivalent w.r.t. their submarkings on Crit represent stutter-equivalent traces, which concludes our proof.

For the following let $\mathsf{Crit} \subseteq P$ be a set of places and $\Sigma' = slice_S(\Sigma, \mathsf{Crit})$ be the safety slice of Σ. If we interpret a safety property φ on a slice $slice_S(\Sigma, \mathsf{Crit})$, we assume that $scope(\varphi) \subseteq \mathsf{Crit}$.

Preservation of Safety Properties. We start by unveiling basic properties of the safety slice. The algorithm constructs the safety slice, so that any transition, that may increase the token count on a place in P', is element of T'.

Lemma 11. *Let p be a place and t be a transition of Σ.*

$$p \in P' \wedge W(p,t) < W(t,p) \Rightarrow t \in T'.$$

Proof. Since at `line 12` of `generateSafetySlice` P_{done} equals P', we show that from `line 4` the property $p \in P_{\mathsf{done}} \wedge W(p,t) < W(t,p) \Rightarrow t \in T'$ holds.

First note, that the algorithm never removes transitions from T'. P_{done} is initialized to Crit in `line 4`. In `line 2` all transitions that may change the token count on Crit become elements of T'. Only in `line 11` the set P_{done} is extended – by a place p. The second while-block (`line 7 to 10`) guarantees that at `line 11` T' already includes any $t \in {}^{\bullet}p$ with $W(p,t) < W(t,p)$. □

A transition sequence σ of Σ generates at most as many tokens on P' as its projection to T', $proj_{T'}(\sigma)$, because in Σ' a place p' is connected to all transitions $t \in T$ that can potentially increase its token count (Eq. 1a).

As W' is the restriction of W to P' and T', a transition sequence in T' has the same effect on P' in Σ and Σ' (Eq. 1b).

The effect on Crit of a transition sequence σ of Σ is the same as of $proj_{T'}(\sigma)$, because all transitions that may change the token count on Crit are in T' (Eq. 1c). For the following equations let $\sigma \in T^{\infty}$ be a transition sequence of Σ and $\sigma' \in T'^{\infty}$ be a transition sequence of Σ'.

$$\forall p \in P' : \ \Delta_{\Sigma}(\sigma,p) \le \Delta_{\Sigma'}(proj_{T'}(\sigma),p). \tag{1a}$$

$$\forall p \in P' : \ \Delta_{\Sigma}(\sigma',p) = \Delta_{\Sigma'}(\sigma',p). \tag{1b}$$

$$\forall p \in \mathsf{Crit} : \ \Delta_{\Sigma}(\sigma,p) = \Delta_{\Sigma'}(proj_{T'}(\sigma),p). \tag{1c}$$

Proof. Equation 1a to 1c can easily be shown by induction on the length of σ and σ' respectively. We first formally prove Eq. 1a and then sum up the proof arguments for Eq. 1b and 1c.

The induction base $|\sigma| = 0$ holds trivially, as the effect ε is always void.

$l \to l+1$: Let $\sigma t \in T^*$ be a transition sequence of length $l+1$. We assume that $\Delta_{\Sigma}(\sigma,p) \le \Delta_{\Sigma'}(proj_{T'}(\sigma),p), \forall p \in P'$ holds. For the following let p' be an arbitrary place in P'.

Let us first assume $t \in T'$. As W' equals $W|_{(P',T')}$, it follows that $\Delta_{\Sigma}(t,p') = \Delta_{\Sigma'}(t,p')$ and hence $\Delta_{\Sigma}(\sigma t,p') \le \Delta_{\Sigma'}(proj_{T'}(\sigma t),p')$. In case $t \in T \setminus T'$, $proj_{T'}(\sigma t)$ equals $proj_{T'}(\sigma)$. So $\Delta_{\Sigma'}(proj_{T'}(\sigma t),p') = \Delta_{\Sigma'}(proj_{T'}(\sigma),p')$ holds. According to Lemma 11, t does not increase the token count on p'. Consequently, $\Delta_{\Sigma}(\sigma t,p') \le \Delta_{\Sigma}(\sigma,p')$ and thus $\Delta_{\Sigma}(\sigma t,p) \le \Delta_{\Sigma'}(proj_{T'}(\sigma t),p)$ holds.

Equation 1b follows, because W' equals $W|_{(P',T')}$. Hence $\Delta_{\Sigma}(t,p) = \Delta_{\Sigma'}(t,p)$, $\forall t \in T', \forall p \in P'$ holds.

The induction proof for Eq. 1c is analogously done to the proof of Eq. 1a. We use that, by `line 2` of `generateSafetySlice`, T' includes all transitions that may change the token count on Crit. □

We now turn to behavioral correspondences between the original net and its safety slice. By the next proposition the sets of firing sequences of Σ and Σ' correspond.

Proposition 12. *Let σ be a firing sequence and M be a marking of Σ.*

(i) $M_{\mathsf{init}}\,[\sigma\rangle\,M \;\Rightarrow\; \exists M' \in [M'_{\mathsf{init}}\rangle : M'_{\mathsf{init}}[\mathit{proj}_{T'}(\sigma)\rangle M'$ *with* $M(p) \le M'(p),\ \forall p \in P'$*.*

Let σ' be a firing sequence and M' a marking of Σ'.

(ii) $M'_{\mathsf{init}}[\sigma'\rangle M' \;\Rightarrow\; \exists M \in [M_{\mathsf{init}}\rangle :\ M' = M|_{P'} \,\wedge\, M_{\mathsf{init}}\,[\sigma'\rangle\,M$*.*

Proof. We show Prop. 12 by induction on the length l of σ and σ', respectively. For the induction base $l = 0$ its enough to note that by Def. 10, $M'_{\mathsf{init}} = M_{\mathsf{init}}|_{P'}$.

$l \to l+1$: First we show (i). Let σt be a firing sequence of Σ of length $l+1$. By the induction hypothesis, $\sigma' := \mathit{proj}_{T'}(\sigma)$ is a firing sequence of Σ' and generates a marking $M'_{\sigma'}$ with at least as many tokens on P' as M_σ, $M_\sigma(p) \le M'_{\sigma'}(p),\ \forall p \in P'$. If t is an element of T', it follows from $M_\sigma[t\rangle$ that M'_σ enables t. By Eq. 1a, it follows that $M_{\sigma t}(p) \le M'_{\sigma' t}(p),\ \forall p \in P'$. If $t \in T \setminus T'$, $\mathit{proj}_{T'}(\sigma) = \mathit{proj}_{T'}(\sigma t)$ which is a firing sequence of Σ' by the induction hypothesis. A transition in $T\setminus T'$ can only decrease the token count on P', thus $M_{\sigma t}(p) \le M_\sigma(p) \le M'_{\sigma'}(p),\ \forall p \in P'$.

For (ii) let $\sigma' t$ be a firing sequence of Σ' with length $l+1$. Since $M'_{\sigma'}$ enables t and by Eq. 1b, also $M_{\sigma'}$ enables t and the generated markings coincide on P', $M_{\sigma' t}|_{P'} = M'_{\sigma' t}$. □

The following proposition implies in combination with Prop. 12 that the sets of finite traces of TS_Σ and $TS_{\Sigma'}$ are stutter-equivalent. It states, that given two marking sequences μ, μ' generated by corresponding firing sequences, we can find for any finite prefix of μ' a stutter-equivalent corresponding finite prefix of μ and vice versa. As we are now assuming that $\mathit{scope}(\varphi) \subseteq \mathsf{Crit}$, we restrict markings to Crit.

At the first glance, Prop. 13 may seem overly complicated by talking about prefixes. But note, $\mathit{unstutter}(\mathcal{M}(M_{\mathsf{init}}, \sigma)|_{\mathsf{Crit}}) = \mathit{unstutter}(\mathcal{M}(M'_{\mathsf{init}}, \sigma')|_{\mathsf{Crit}})$ does not necessarily hold, since either just σ or σ' may be maximal and hence one marking sequence would be finite whereas the other would be infinite.

Proposition 13. *Let $\sigma \in T^*$ be a firing sequence of Σ with $\sigma' := \mathit{proj}_{T'}(\sigma)$.*

(i) If μ is a finite prefix of $\mathcal{M}(M_{\mathsf{init}}, \sigma)$, then there is a finite prefix μ' of $\mathcal{M}(M'_{\mathsf{init}}, \sigma')$ with $\mathit{unstutter}(\mu|_{\mathit{Crit}}) = \mathit{unstutter}(\mu'|_{\mathit{Crit}})$.

Let $\sigma' \in T'^*$ be a firing sequence of Σ'.

(ii) If μ' is a finite prefix of $\mathcal{M}(M'_{\mathsf{init}}, \sigma')$, then there is a finite prefix μ of $\mathcal{M}(M_{\mathsf{init}}, \sigma')$ with $\mathit{unstutter}(\mu|_{\mathit{Crit}}) = \mathit{unstutter}(\mu'|_{\mathit{Crit}})$.

Proof. We only prove (i). (ii) follows analogously. We show that $\mathcal{M}(M'_{\mathsf{init}}, \sigma')|_{\mathsf{Crit}}$ starts with a stutter-equivalent version of $\mu|_{\mathsf{Crit}}$. The proof is by induction on the length l of μ.

First note that the initial markings M_{init} and M'_{init} coincide on Crit and hence for a prefix of length 1 the above holds.

$l \to l+1$: Let μM be a prefix of $\mathcal{M}(M_{\mathsf{init}}, \sigma)$ of length $l+1$. Let $\sigma_\mu t$ be the firing sequence generating μM. Let σ'_μ be the projection of σ_μ to T', $proj_{T'}(\sigma_\mu)$. By the induction hypothesis $\mathcal{M}(M'_{\mathsf{init}}, \sigma'_\mu)$ has a prefix μ' such that $\mu|_{\mathsf{Crit}}$ and $\mu'|_{\mathsf{Crit}}$ are stutter-equivalent. The case in which $\mu|_{\mathsf{Crit}}$ and $\mu M|_{\mathsf{Crit}}$ are stutter-equivalent follows trivially. Otherwise, t changes the submarking on Crit and hence t is an element of T'. Let M' be the marking generated by $\sigma'_\mu t$. So $\mathcal{M}(M'_{\mathsf{init}}, \sigma'_\mu t)$ has a prefix that starts with μ' and ends with M', $\mu'\mu'_2 M'$. By Eq. 1c, M coincides with M' on Crit. Since $unstutter(\mu|_{\mathsf{Crit}}) = unstutter(\mu'|_{\mathsf{Crit}})$ holds, it follows that μ' reflects all changes on Crit caused by σ_μ. Hence there cannot be a change on the submarking of Crit within μ'_2. So $\mu'\mu'_2 M'$ is stutter-equivalent to $\mu' M'$ and hence stutter-equivalent to μM. □

Theorem 14 (Preservation of Safety Properties). *Let Σ be a Petri net and $\mathsf{Crit} \subseteq P$ be a set of places. Let Σ' be $slice_S(\Sigma, \mathsf{Crit})$ and φ a stutter-invariant linear-time safety property with $scope(\varphi) \subseteq \mathsf{Crit}$.*

$\Sigma \models \varphi$ if and only if $\Sigma' \models \varphi$.

Proof. By Prop. 9 it is sufficient to show that $unstutter(\mathsf{Traces}_{TS_\Sigma,\mathsf{fin}}(M_{\mathsf{init}})) = unstutter(\mathsf{Traces}_{TS_{\Sigma'},\mathsf{fin}}(M'_{\mathsf{init}}))$. Let ϑ be a finite trace of TS_Σ. Let σ be a corresponding firing sequence of Σ, i.e. σ corresponds to a path μ with $L(\mu) = \vartheta$. By Prop. 12, $\sigma' = proj_{T'}(\sigma)$ is also a firing sequence of Σ'. Hence it follows by Prop. 13, that there is a finite path μ' in TS'_Σ such that $\mu'|_{\mathsf{Crit}}$ and $\mu|_{\mathsf{Crit}}$ are stutter-equivalent. Since $scope(\varphi) \subseteq \mathsf{Crit}$, it follows that μ' generates a trace ϑ' that is stutter-equivalent to ϑ.

Analogously follows that for a finite trace ϑ' of $TS_{\Sigma'}$ there is stutter equivalent trace ϑ of TS_Σ. □

4 Related Work

The slicing and other reduction approaches are relatively old research areas and have received much attention. In this section we highlight differences and similarities to the most relevant works.

4.1 Petri Net Slicing

In [4] C. K. Chang and H. Wang presented a first slicing algorithm on Petri nets for testing. For a given set of communication transitions CS, their algorithm determines the sets of paths in the Petri net graph, called concurrency sets, such that all paths within the same set should be executed concurrently to allow for the execution of all transitions in CS.

Whereas the approach of Chang and Wang does not yield a reduced net, Llorens et. al. developed an algorithm to generate a reduced Petri net [7]. They showed how to use Petri net slicing for reachability analysis and debugging presenting a forward and backward algorithm for Petri nets with maximal arc weight 1, as shown in Fig. 3. A forward slice is computed for all initially marked places. They presented a second algorithm to compute a backward slice for a slicing criterion Crit based on our CTL$^*_{\text{-X}}$ slicing algorithm `generateSlice` as presented in [12,11]. Their (combined) slice is defined by $\Sigma' = (P', T', W|_{(P',T')}, M_{\mathsf{init}}|_{P'})$ with $(P', T') = \text{forwardSlice}(\Sigma) \cap \text{backwardSlice}(\Sigma, \mathsf{Crit})$.

```
forwardSlice($\Sigma$){
    $T'$:=$\{t \in T \mid M_{\mathsf{init}}[t\rangle\}$;
    $P'$:= $\{p \in P \mid M_{\mathsf{init}}(p) > 0\} \cup T'^{\bullet}$;
    $T_{\text{do}}$:= $\{t \in T \setminus T' \mid {}^{\bullet}t \subseteq P'\}$;
    while ( $T_{\text{do}} \neq \emptyset$ ) {
        $P'$:= $P' \cup T_{\text{do}}^{\bullet}$;
        $T'$:= $T' \cup T_{\text{do}}$;
        $T_{\text{do}}$:= $\{t \in T \setminus (T') \mid {}^{\bullet}t \subseteq P'\}$ }
    return $(P', T')$ }
```

```
backwardSlice($\Sigma, C$){
    $T'$:= $\emptyset$;
    $P'$:= $C$;
    while ( ${}^{\bullet}P' \neq T'$ ) {
        $T'$:= $T' \cup {}^{\bullet}P'$;
        $P'$:= $P' \cup {}^{\bullet}T'$;}
    return $(P', T')$}
```

Fig. 3. Llorens' forward and backward slice according to [7]

Obviously the forward slice can also be used as a preprocessing step to model checking and removes dead transitions only. Their slice was considered correct iff for every firing sequence σ of the original net Σ it holds that the restriction $\sigma' = proj_{T'}(\sigma)$ can be performed on Σ' and for every place p' of the slice it holds that firing σ' generates at least as many tokens as σ. We infer that their slice allows falsification but no verification of lower bounds, and their slice allows verification and falsification of upper bounds, but no decision whether a certain submarking is reachable.

The principal difference between the `backwardSlice` of Llorens et. al. and our CTL$^*_{\text{-X}}$ slicing algorithm is that `backwardSlice` includes only those transitions that increase the token count on slice places whereas CTL$^*_{\text{-X}}$ slicing also includes transitions that decrease the token count. Now our *safety* slicing algorithm combines the two approaches. It uses CTL$^*_{\text{-X}}$ slicing on Crit and a refined version of `backwardSlice` on $P' \setminus \mathsf{Crit}$. By exploiting read arcs and considering arc weights, `line 5` in the `backwardSlice` algorithm (c.f. Fig. 3) can be replaced by

$$T' := T' \cup \{t \mid t \in {}^{\bullet}P' \land \exists p \in P' : W(t,p) > W(p,t)\};$$

Now the backward algorithm adds new transitions only if they might produce *additional* tokens on interesting places. This principle is used in the safety slicing algorithm of Def. 10.

Let us compare the three algorithms—the algorithm of Llorens for examining bounds, our algorithm preserving CTL$^*_{\text{-X}}$ properties and our algorithm preserving safety properties. The idea of forward slicing can be used for our algorithms as well. It can be seen as a preprocessing step applied before the backward slicing. The idea to use read arcs and to extend the algorithm to weighted Petri nets is also applicable to the algorithm of Llorens et al. So let us compare the algorithms for backward slicing considering the version of Llorens et al. extended for weighted Petri nets as discussed above. Our algorithm for slicing of $\mathrm{CTL}^*_{\text{-X}}$ properties is the least aggressive but most conservative algorithm, that is its slices are bigger or as big as slices generated by the other algorithms but preserves the most properties. The algorithm for slicing of safety properties is more aggressive than that preserving $\mathrm{CTL}^*_{\text{-X}}$ but less aggressive than the algorithm preserving bounds. The algorithm of Llorens is the most aggressive algorithm and is also the least conservative. Note, that all three variants produce the same results on strongly-connected nets.

4.2 Petri Net Reductions

Petri net slicing is a structural reduction technique, as slicing constructs a smaller net based on the model structure, i.e. the Petri net graph. There are only a few Petri net reductions that preserve temporal properties. Pre- and postagglomeration [2] are two very powerful structural reduction rules and probably also the most established. In [9] it was shown that they preserve $\mathrm{LTL}_{\text{-X}}$ properties.

Pre- and postagglomerations merge two transition sets $H := {}^\bullet p$ and $F := p^\bullet$ around a place p into a new one, HF. Applying these rules changes the net structure. So when model checking the reduced net a counterexample needs a translation first to be executable on the original net. Whereas slicing preserves the net structure by taking every place and transition the places in $scope(\varphi)$ causally depends on, agglomerations can also be applied in between to shorten causal dependencies. But agglomerations are not applicable in the following scenarios: (1) Transition sets $H := {}^\bullet p$ and $F := p^\bullet$ are not agglomerateable, if place p is marked. (2) Given a place p with more than one input and output transition, if any transition in $F := p^\bullet$ has an input place other than p, $H := {}^\bullet p$ is not postagglomerateable. (3) Given a place p with ${}^\bullet p = \{h\}$, $h \in T$, if h has an output place other than p, $F := p^\bullet$ is not preagglomerateable and (4) if other transitions consume tokens from the input places of h, h is not agglomerateable at all. It is easy to build a net that exposes a lot of these constructs but is nicely sliceable for a given property.

5 Evaluation

In this section we present evaluation results on the benchmark set of J. C. Corbett [5]. The set of 75 examples consists of real Ada tasking programs as well as standard benchmark examples from the concurrency analysis literature. In the benchmark set there are five non-scalable systems and seventeen systems were scaled and are present in four different sizes. We say that nets belong to the same *family* if they are scaled up versions of the same system.

We applied a fully automatic evaluation procedure that does not require the specification of temporal properties. For each place of a net, a slice was generated.

We belive that not every single place corresponds to a meaningful or relevant temporal property. So, to avoid an overly optimistic evaluation result, we filter out the smallest slices. This clearly is a heuristic, as the smallest slices may or may not be due to a meaningless slicing criterion. We measure the effect of slicing in terms of savings of the reachable state space, as the size of the state space usually has a strong influence on time and space needed for model checking.

We call a slice with a state space smaller than its original's *properly effective*. When considering uncondensed state spaces (cf. Sect. 5.1), slicing guarantees that the reachable state space of a slice is at most as big as the original's. When we consider state spaces that are condensed using partial order reductions (cf. Sect. 5.2) this is not neccessarily the case. Nevertheless, we demonstrate that slicing may even be an useful preprocessing step when it is daisy chained with partial order reduction techniques. Slices that have a state space greater than the original's are called *limited effective*.

5.1 Effect on the Full State Space

Let us consider the savings in terms of the state space, i.e. in terms of the number of reachable states and state transitions. According to Table 1 safety slicing gains much more savings than $\text{CTL}^*_{\text{-X}}$ slicing.

Table 1. Results on the full state space

	$\frac{\#\text{properly red. families}}{\#\text{families}}$	mean state space savings *(states, state trans.)*	mean net graph savings *(places, trans.)*
safety slicing	10/23	(0.16, 0.13)	(0.10, 0.03)
$\text{CTL}^*_{\text{-X}}$ slicing	9/23	(0.07, 0.09)	(0.09, 0.02)

Table 2 illustrates the results when only slices are considered that save at least 10% of the state space. A coverage of x% means that for x% of the places in the original nets there is a slice that saves at least 10% of the state space. So for more than a third of all places a safety slice exists with a state space that is at least 10% smaller than the state space of the original net.

Table 2. Reducts with a State Space Saving of 10%

	$\frac{\#\text{properly red. families}}{\#\text{families}}$	coverage on reducible *places* [%]	coverage on all nets *places* [%]
safety slicing	9/23	85.36	35.39
$\text{CTL}^*_{\text{-X}}$ slicing	6/23	48.38	18.58

We already know from the Table 1 that safety slicing gains the greater savings than $\text{CTL}^*_{\text{-X}}$ slicing. Comparing the results of $\text{CTL}^*_{\text{-X}}$ slicing (a) and safety slicing (b) in Fig. 4 shows that safety slicing is able to reduce the same nets more aggressively than $\text{CTL}^*_{\text{-X}}$ slicing and also is able to reduce more nets.

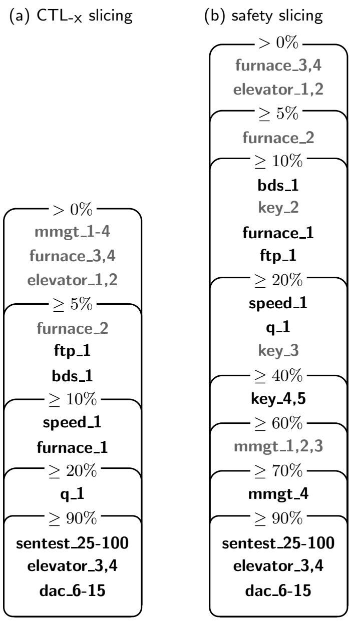


Fig. 4. Properly effective reduced nets clustered by their savings w.r.t. the full state space. Each column displays the savings of the respective slicing technique. A net name appears within a cluster when a net has a properly effective reduct with a saving within the cluster's range. The earliest occurrence of a family is marked in black.

5.2 Partial Order Reductions

Partial order reductions (=POR) refer to a family of powerful reduction techniques developed to avoid the blow-up caused by concurrent behaviours. One reason of the state space explosion problem is that the interleaving semantics represents concurrency of actions by interleaving them in all possible ways, whereas the actions' total effect is independent of their ordering. POR condense state spaces by decreasing the number of equivalent interleavings.

In this section we examine the combination of safety slicing and POR by comparing the condensed state space of the original net with the condensed state spaces of its slices. As in Sect. 5.1, we filter out the smallest reducts *with respect to the ful state space*. The combination of slicing and POR is particularly interesting, because the slicing effect as measured in Sect. 5.1 profits from eleminating concurrent behaviours. We use the stubborn set technique [13] of PROD tool[10] as POR technique. We chose to condense the state space by deadlock preserving stubborn sets as we have built slices for single places not temporal-logical formulas. Usually a state space condensed to preserve deadlocks is expected to be smaller than (or equal to) a state space condensed to preserve safety properties or e.g. LTL-X properties [13]. We hence believe that the results presented in the following allow to study the general effects of combining Petri net graph reductions with stubborn sets.

Results with Respect to Condensed State Spaces. Since we now measure the results with respect to the condensed state space, we say that we have a *saving of x* of states (state transitions), if the reduct has factor x less states (state transitions) than the original net has in its condensed state space. Analogously, we use *overhead*, *benefit* and *cost* with respect to the condensed state space.

Of course, the *condensed* state space of a *reduced* net generated by the stubborn set technique is smaller than (or equals) the *full* state space of the reduced net and is hence also smaller than the full state space of the original, but the *condensed* state space of a reduced net may not be smaller than the *condensed* state space of the original net if the stubborn set performs worse on the reduced net (cf. Fig. 5). It may be counterintuitive that the *condensed* state space of the *reduced* net can be bigger than the *condensed* state space of the *unreduced* net even when the full state space of the *reduced* net is substantially smaller than the full state space of the *unreduced* net. But as POR usually implement a heuristic to determine which transitions can be considered as independent, such a heuristic can work better for one net than for the other so that the stubborn set condensation on the original may be more effective than the condensation on the reduced net.

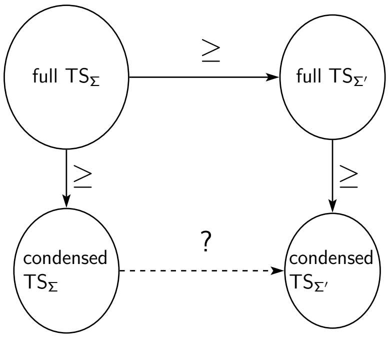


Fig. 5. Condensed and reduced state spaces. TS_Σ refers to the state space of the original system and $TS_{\Sigma'}$ to the state space of a reduct.

Table 3. Mean savings w.r.t. condensed state spaces. The second column gives the savings gained by slicing w.r.t. the condensed state space.

	mean savings relative to the condensed state space (states,state trans.)	# limited effective # nets	# properly effective # nets
safety slicing	$0.373, 0.265$	14	714
$\mathrm{CTL}^*_{\text{-X}}$ slicing	$0.7, 1.2$	41	631

Let us study the results summarised in Table 3. According to Table 3 applying $\mathrm{CTL}^*_{\text{-X}}$ slicing does not yield a great benefit w.r.t. the mean state space, whereas safety slicing significantly decreases the mean state space yielding a benefit of 36.3% of the states and 26.5% of the state transitions with respect to the

	(a) partial order red. (=po)	(b) po + CTL-X slicing	(c) po + safety slicing
$> 0\%$	abp_1	*abp_1*	*abp_1*
$\geq 10\%$	dph_4; gas_q_1; elevator_3,4	**dph_4**; *gas_q_1*	**dph_4**; *gas_q_1*
$\geq 20\%$	gas_nq_2; key_2,3,4,5	*gas_nq_2*; **key_2,3,4,5**	*gas_nq_2*
$\geq 30\%$	dph_5; elevator_2	**dph_5**; **elevator_2**	**dph_5**; **elevator_2**
$\geq 40\%$	elevator_1; gas_q_2; gas_nq_3; dph_6	**elevator_1**; *gas_q_2*; *gas_nq_3*; **dph_6**	*gas_q_2*; *gas_nq_3*; **dph_6**
$\geq 50\%$	gas_q_3; ring_3; dpd_4; mmgt_1; gas_nq_4; over_2; dph_7	*gas_q_3*; *ring_3*; *dpd_4*; **mmgt_1**; *gas_nq_4*; *over_2*; **dph_7**	*gas_q_3*; *ring_3*; *dpd_4*; *gas_nq_4*; **elevator_1**; *over_2*; **dph_7**
$\geq 60\%$	gas_q_4; mmgt_2,3; gas_nq_5	*gas_q_4*; mmgt_2,3; *gas_nq_5*	*gas_q_4*; **mmgt_1**; *gas_nq_5*
$\geq 70\%$	speed_1; furnace_2,3,4; ring_5; dpd_5; mmgt_4; over_3; sentest_25-100	**speed_1**; furnace_2,3,4; *ring_5*; *dpd_5*; **mmgt_4**; *over_3*	**speed_1**; **furnace_2,3,4**; *ring_5*; *dpd_5*; **mmgt_2**; *over_3*
$\geq 80\%$	dac_6; ring_7; dpd_6; q_1; over_4,5; furnace_1	**sentest_100**; *ring_7*; *dpd_6*; **q_1**; *over_4,5*; **furnace_1**	**key_2**; **mmgt_3,4**; *ring_7*; *dpd_6*; **q_1**; *over_4,5*; **furnace_1**
$\geq 90\%$	bds_1; dac_9,12,15; ftp_1; ring_9; dpd_7	**elevator_3,4**; **sentest_25,50,75**; **bds_1**; **dac_6-15**; **ftp_1**; *ring_9*; *dpd_7*	**key_3,4,5**; **elevator_3,4**; **sentest_25-100**; **bds_1**; **dac_6-15**; **ftp_1**; *ring_9*; *dpd_7*

Fig. 6. Properly effective reduced nets clustered by their savings w.r.t. the condensed state spaces. (a) lists the savings gained by PROD's stubborn set reduction. (b) and (c) show the reductions on the condensed state space when CTL$^*_{\text{-X}}$ and saftey slicing reduction is applied. The earliest occurrence of a family is marked in black. Nets reduced by the respective reduction are set in bold face. Nets left unchanged by the respective reductions are set in italics.

condensed state space. This benefit is about twice as much as safety slicing could save on the full state space (cf. Table 1). This is mainly due to three families that are more effectively condensed by stubborn set reductions when safety sliced.

Table 3 shows that there are (also for $\mathrm{CTL}^*_{\text{-X}}$ slicing) many instances where the application of the reductions increases the state space savings and that the majority of reducts improves the state space savings. According to Fig. 6 some nets were reduced so much that they now appear in a higher savings cluster while all other nets remain in the same savings cluster.

Summary and Conclusions. In this section we examined the effect of safety slicing—in comparison with $\mathrm{CTL}^*_{\text{-X}}$ slicing and in combination with POR. To examine the general effects of such a combination we used deadlock preserving stubborn sets of PROD. Safety slicing increased the state space savings on the full state space as well as on the condensed state space considerably.

6 Conclusion and Future Work

In this paper we introduced Petri net safety slicing in order to alleviate the state space explosion problem for model checking Petri nets. A safety slice $slice_S(\Sigma, \mathsf{Crit})$ satisfies the same stutter-invariant linear-time safety properties φ as the original net, given that φ refers to the places of the slicing criterion Crit only. Safety slicing can yield more aggressive reductions than $\mathrm{CTL}^*_{\text{-X}}$ slicing but sacrifices the preservation of liveness properties.

It seems worthwhile to develop refined slicing algorithms for certain (classes of) properties that allow to formulate even more aggressive reductions. A good starting point seems antecedent slicing [14,15], a form of conditional slicing where information about system input is encoded as antecedent of an LTL formula. If we study a formula of the form $\psi := \mathbf{G}\,(\varphi_1 \Rightarrow \mathbf{F}\,\varphi_2)$, we only need to include transitions that make the antecedent φ_1 true, we do not need to include transitions that are fired when φ_1 cannot become true [15]. We conjecture that in this setting a safety slicing like algorithm can be used for the antecedent places, $scope(\varphi_1)$, whereas $\mathrm{CTL}^*_{\text{-X}}$ slicing has to be applied to places in $scope(\varphi_2)$. Since both, safety slicing and $\mathrm{CTL}^*_{\text{-X}}$ slicing, are not able to reduce strongly connected nets, an important aspect is to explore whether the antecedent can be used to eliminate transitions when their firing implies that the antecedent cannot become true.

References

1. Baier, C., Katoen, J.-P.: Principles of Model Checking. The MIT Press (2008); 12, 112–120, 422–425
2. Berthelot, G.: Checking Properties of Nets Using Transformation. In: Rozenberg, G. (ed.) APN 1985. LNCS, vol. 222, pp. 19–40. Springer, Heidelberg (1986)
3. Brückner, I.: Slicing Integrated Formal Specifications for Verification. PhD thesis. University of Paderborn (March 2008)

4. Chang, C.K., Wang, H.: A slicing algorithm of concurrency modeling based on Petri nets. In: Hwang, K., Jacobs, S.M., Swartzlander, E.E. (eds.) Proc. of the 1986 Int. Conf. on Parallel Processing, pp. 789–792. IEEE Computer Society Press, Washington (1987)
5. Corbett, J.C.: Evaluating Deadlock Detection Methods for Concurrent Software. IEEE Transactions on Software Engineering 22(3), 161–180 (1996)
6. Lamport, L.: What Good is Temporal Logic? In: Information Processing 1983: Proceedings of the IFIO 9th World Computer Congress, pp. 657–668 (1983)
7. Llorens, M., Oliver, J., Silva, J., Tamarit, S., Vidal, G.: Dynamic Slicing Techniques for Petri Nets. In: Proceedings of the Second Workshop on Reachability Problems in Computational Models (RP 2008), Liverpool, UK. Electronic Notes in Theoretical Computer Science, vol. 223, pp. 153–165 (December 2008)
8. Peled, D., Wilke, T.: Stutter-Invariant Temporal Properties are Expressible Without the Next-time Operator. Information Processing Letters 63(5), 243–246 (1997)
9. Poitrenaud, D., Pradat-Peyre, J.-F.: Pre- and Post-agglomerations for LTL Model Checking. In: Nielsen, M., Simpson, D. (eds.) ICATPN 2000. LNCS, vol. 1825, pp. 387–408. Springer, Heidelberg (2000)
10. PROD. Pr/T-net reachability analysis tool. Helsinki Univerity of Technology, `http://www.tcs.hut.fi/Software/prod/`
11. Rakow, A.: Slicing Petri nets with an Application to Workflow Verification. In: Geffert, V., Karhumäki, J., Bertoni, A., Preneel, B., Návrat, P., Bieliková, M. (eds.) SOFSEM 2008. LNCS, vol. 4910, pp. 436–447. Springer, Heidelberg (2008)
12. Rakow, A.: Slicing Petri Nets. Technical Report, Carl von Ossietzky Universität Oldenburg, 20 pages (2007), `http://parsys.informatik.uni-oldenburg.de/pubs/SlPN_tr.pdf`
13. Valmari, A.: The State Explosion Problem. In: Reisig, W., Rozenberg, G. (eds.) APN 1998. LNCS, vol. 1491, pp. 429–528. Springer, Heidelberg (1998)
14. Vasudevan, S., Emerson, E.A., Abraham, J.A.: Efficient model checking of hardware using conditioned slicing. In: Proc. of the 4th International Workshop on Automated Verification of Critical Systems, AVOCS 2004. Electronic Notes in Theoretical Computer Science, vol. 128(6), pp. 279–294. Elsevier Science Publishers (2004)
15. Vasudevan, S., Emerson, E.A., Abraham, J.A.: Improved Verification of Hardware Designs Through Antecedant Conditioned Slicing. International Journal of Software Tools and Technology Transfer (STTT) 9(1), 89–101 (2007)
16. Weiser, M.: Program slicing. In: Proceedings of the 5th International Conference on Software Engineering, pp. 439–449. IEEE Press, Piscataway (1981)

Capacity Planning for Vertical Search Engines: An Approach Based on Coloured Petri Nets

Veronica Gil-Costa[1,3], Jair Lobos[2],
Alonso Inostrosa-Psijas[1,2], and Mauricio Marin[1,2]

[1] Yahoo! Research Latin America
[2] DIINF, University of Santiago of Chile
[3] CONICET, University of San Luis Argentina
gvcosta@yahoo-inc.com

Abstract. This paper proposes a Colored Petri Net model capturing the behaviour of vertical search engines. In such systems a query submitted by a user goes through different stages and can be handled by three different kinds of nodes. The proposed model has a modular design that enables accommodation of alternative/additional search engine components. A performance evaluation study is presented to illustrate the use of the model and it shows that the proposed model is suitable for rapid exploration of different scenarios and determination of feasible search engine configurations.

Keywords: Web search engines, Petri Net applications.

1 Introduction

Vertical search engines are single-purpose dedicated systems devised to cope with highly dynamic and demanding workloads. Examples include advertising engines in which complex queries are executed on a vertical search engine each time a user displays an e-mail in a large system such as Yahoo! mail. As potentially millions of concurrent users are connected to e-mail at any time, the workload on the search engine is expected to be of the order of many hundred thousand queries per second. When vertical search engines are used as part of large scale general purpose search engines, queries per second intensity is featured by the unpredictable behavior of users who are usually very reactive to worldwide events.

In such cases, models and tools for performance evaluation are useful to plan capacity of data center clusters supporting computation of search engines. The typical questions one would like to quickly answer through a reasonable model of the actual system are like "given that next year we expect a X% increment in query traffic, what are the feasible sizes of the different search engine services so that we make an efficient use of hardware resources deployed in the data center?". An appropriate answer to this question could reduce operational costs at the data center. A model like the one proposed in this paper is also useful in research when one wants to evaluate alternative ideas for service design and deployment on cluster of processors.

In this paper we focus on vertical search engines since they are amenable for modelling as they are built from a small number of fairly simple components called services. The question of modelling general purpose search engines remains as an open area of research.

The contribution of this paper is the proposal of a method to model actual parallel computations of vertical search engines. The method makes use of Petri nets which is a well-known tool for modelling complex systems. Petri net realization is designed in a modular manner which enables evaluation of different alternatives for service design and configuration.

Typically each service of a vertical search engine is deployed on a large set of processors forming a cluster of processors. Both processors and communication network are constructed from commodity hardware. Each processor is a multi-core system enabling efficient multi-threading on shared data structures, and message passing is performed among processors to compute on the distributed memory supported by the processors. Modelling each of those components is a complex problem in its own merits.

Fortunately, ours is a coarse grain application where the running time cost of each service is dominated by a few primitive operations. We exploit this feature to formulate a model based on coloured Petri nets where tokens represent user queries which are circulated through different cost causing units in accordance with the computations performed to solve the query in the actual search engine. Queries are modeled as directed acyclic graphs whose arcs represent the causality relationships among the different steps related to processing the query in the actual system, and vertices represent points in which cost must take place by considering the effects of other queries under processing. The relative cost of primitive operations is determined by means of small benchmark programs. The coarse grain feature allows us to model the effects of actual hardware and system software by identifying only a few features that have a relevant influence in the overall cost of computations.

We validated the proposed Petri net model against a complex discrete event simulator of the same system and an actual implementation of a small search engine. The advantage of the Petri net model over the simulator is that it is much simpler and efficient than the simulator, and it is fairly easy to extend to include new features or change behavior of services by using a graphical language and model verification tests.

This paper is organized as follows: In Section 2 we review Petri nets concepts, query processing and related works. Section 3 presents our proposed method to model vertical search engines. Section 4 shows the model validation and experimental results. Conclusions follow in Section 5.

2 Background

We briefly introduce coloured Petri nets (CPNs), then we describe vertical Web Search Engines (WSEs) and related work.

CPN is a high-level Petri net formalism, extending standard Petri nets to provide basic primitives to model concurrency, communication and synchronization.

The notation of CPNs introduces the notion of token types, namely tokens are differentiated by colors, which may be arbitrary data values. They are aimed at practical use because they enable the construction of compact and parameterized models. CPNs are bipartite directed graphs comprising places, transitions and arcs, with inscriptions and types allowing tokens to be distinguishable [16,25]. In Figure 4 we see part of a CPN. Places are ovals (e.g., Queue) and transitions are rectangles (e.g., start). Places have a type (e.g., Server × Query) and can have an initial marking which is a multi-set of values (tokens) of the corresponding type. Arcs contain expressions with zero or more free variables. An arc expression can be evaluated in a binding using the assignments, resulting in a value. A binding is a transition and assignment of values to all its free variables. Then, a binding is enabled if all input places contain at least the tokens prescribed by evaluating the arc expressions.

Hierarchical Colored Petri Nets (HCPNs) introduce a facility for building a CPN out of subnets or modules. The interface of a module is described using port places, places with an annotation In, Out, or I/O. A module can be represented using a substitution transition, which is a rectangle with a double outline (e.g., FS in Figure 2). The module concept of CPN is based on a hierarchical structuring mechanism allowing a module to have sub-modules and reuse of sub-modules in different parts of the model. Places connected to a substitution transition are called socket places and are connected to port places using port/socket assignments.

There are useful software to support CPN modelling like CPN-AMI[1], GreatSPN[2] and Helena [3]. In particular, CPN-tools [4] supports HCPNs modeling and provides a way to "walk through" a CPN model by allowing one to investigate different scenarios in detail and check whether or not the model works as expected. It is possible to observe the effects of the individual steps directly on the graphical representation of the CPN model. Also, this tool auto-generates Java code, a CPN simulator, which can be modified to introduce others metrics and can be used to fastly run very large numbers of queries without using the graphical interface. We emphasize that performance metric results are always highly dependent on the stream of user queries that are passed through the search engine. Thus any reasonable performance evaluation study must consider the execution of thousands of millions of actual user queries. In this context, the graphical definition of the model is useful for model construction and verification, and then the production model is executed through the Java CPN simulator.

2.1 Query Processing

Large scale general Web search engines are commonly composed of a collection of services. Services are devised to quickly process user queries in an on-line manner. In general, each service is devoted to a single operation within the

[1] http://www.lip6.fr/cpn-ami
[2] http://www.di.unito.it/~greatspn/
[3] http://www.lipn.univ-paris13.fr/~evangelista/helena
[4] http://cpntools.org/

whole process of solving a query. One of such services is the caching service (CS), which is in charge of administering a distributed cache devoted to keep the answers for frequent queries (they are query results composed of document IDs). This service is usually partitioned using a hashing based strategy such as Memcached [14]. A second service is the Index Service (IS), which is responsible for calculating the top-k results (document IDs) that best match a query. A third service, called Front-Service (FS) is in charge of receiving new queries, routing them to the appropriate services (CS, IS) and performing the blending of partial results returned from the services. Other related services include: a) construction of the result Web page for queries, b) advertising related to query terms, c) query suggestions, d) construction of snippets, which is a small summary text surrounding the document ID (URL) of each query result, between others. Given the huge volume of data, each service is deployed on a large set of processors wherein each processor is dedicated to efficiently perform a single task. Multi-threading is used to exploit multi-core processing on data stored in the processor.

In this work we focus on vertical WSEs consisting of three main services: Front-End Service (FS), Caching Service (CS) and Index Service (IS). Each service is deployed on a different cluster of processors or processing nodes. Figure 1 shows the query processing operations performed by a WSE as explained below.

The Front-End Service (FS) is composed of several processing nodes and each node supports multi-threading. This service is composed of a set of FS nodes where each one is mapped onto a different processor. Each FS node receives user queries and sends back the top-k results to the requester (a user or another machine requiring service). After a query arrives to a FS node f_i, we select a caching service (CS) machine to determine whether the query has been previously processed. For the CS cluster architecture, we use an array of $P \times D$ processors or caching service nodes. A simple LRU (Least Recently Used) approach is used. The memory cache partition is performed by means of a distributed memory object caching system named Memcached [14], where one given query is always assigned to the same CS partition. Memcached uses a hash function with uniform distribution over the query terms to determine the partition cs_j which should hold the entry for the query. To increase throughput and to support fault tolerance, each partition is replicated D times. Therefore, at any given time, different queries can be solved by different replicas of the same partition. Replicas are selected in a round-robin way.

If the query is cached, the CS node sends the top-k result document IDs to the FS machine. Afterwards the FS f_i sends the query results to users. Otherwise, if the query is not found in cache, the CS node sends a hit-miss message to the FS f_i. Then, the FS machine sends an index search request to the index service (IS) cluster. The IS contains an index built from a large set of documents. The index is used to speed up the determination of what documents contain the query terms. A document is a generic concept, it can be an actual text present in a Web page or it can be a synthetic text constructed for the specific application like a short text for advertisement. The index allows the fast mapping among query terms and documents.

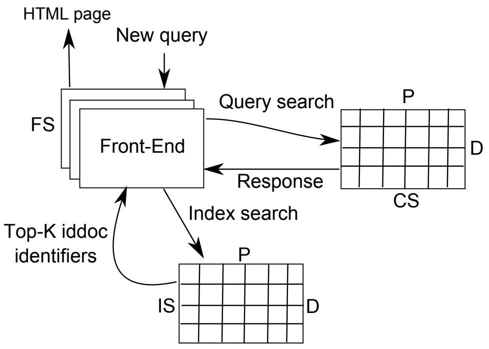


Fig. 1. Query processing

The amount of documents and indexes are usually huge and thereby they must be evenly distributed onto a large set of processors in a sharing nothing fashion. Usually these systems are expected to hold the whole index in the distributed main memory held by the processors. Thus, for the IS setting, the standard cluster architecture is an array of $P \times D$ processors or index search nodes, where P indicates the level of document collection partitioning and D the level of document collection replication. The rationale for this 2D array is as follows: each query is sent to all of the P partitions and, in parallel, the local top-k document IDs in each partition are determined. These local top-k results are then collected together by the FS node f_i to determine the global top-k document IDs.

The index stored in each index search node is the so-called inverted index [5]. The inverted index [5,30,20,23,29,22] (or inverted file) is a data structure used by all well-known WSEs. It is composed of a *vocabulary table* (which contains the V distinct relevant terms found in the document collection) and a set of *posting lists*. The posting list for term $c \in V$ stores the identifiers of the documents that contain the term c, along with additional data used for ranking purposes. To solve a query, one must fetch the posting lists for the query terms, compute the intersection among them, and then compute the ranking of the resulting intersection set using algorithms like BM25 or WAND [6]. Hence, an inverted index allows for the very fast computation of the top-k relevant documents for a query, because of the pre-computed data. The aim is to speed up query processing by first using the index to quickly find a reduced subset of documents that must be compared against the query, to then determine which of them have the potential of becoming part of the global top-k results.

2.2 Related Work

There are several performance models for capacity planning of different systems [2,21,26], but these models are not designed in the context of web search engines. The work in [13] presents a performance model for searching large text databases by considering several parallel hardware architectures and search algorithms. It examines three different document representations by means of simulation and explores response times for different workloads. More recently the work in [11] presents an Inquery simulation model for multi-threading distributed IR system.

The work in [12] presents a framework based upon queuing network theory for analyzing search systems in terms of operational requirements: response time, throughput, and workload. But the proposed model does not use the workload of a real system but a synthetic one. A key feature in our context is to properly consider user behavior through the use of actual query logs. Notice that in practical studies, one is interested in investigating the system performance by using an existing query log. This log has been previously collected from the queries issued by the actual users of the search engine. To emphasize this fact, below we call it "*the query log under study*". Moreover, [12] assumes perfect balance between the service times of nodes. Another disadvantage is that it does not verify the accuracy of their model with actual experimental results.

In [9,10] the authors simulate different architectures of a distributed information retrieval system. Through this study it is possible to approximate an optimal architecture. However, they make the unrealistic assumption that service times are balanced when the Information Retrieval nodes handle a similar amount of data when processing a query. This work is extended in [7] to study the interconnection network of a distributed Information Retrieval system. This work is also extended in [8] to estimate the communication overhead.

In [17] the authors present algorithms for capacity analysis for general services in on-line distributed systems. The work in [19] presents the design of simulation models to evaluate configurations of processors in an academic environment. The work presented in [18] proposes a mathematical algorithm to minimize the resource cost for a server-cluster. In our application domain, the problem of using mathematical models is that they are not capable of capturing the dynamics of user behavior nor temporarily biased queries on specific query topics. The work in [18] is extended in [28] to include mechanisms which are resilient to failures.

The work presented in [4] and continued in [3] characterizes the workload of the search engines and use the approximate MVA algorithm [24,21]. However, this proposal is evaluated on a very small set of IS nodes, with only eight processors and just one service, namely IS. The effects of asynchronous multi-threading is not considered as they assume that each processor serves load using a single thread. This work also does not consider the effects caused in the distribution of inter-arrival times when queries arrive at more than one FS node. They also use the harmonic number to compute average query residence time at the index service (IS) cluster. This can be used in WSE systems such that every P_i index partition delivers its partial results to a manager processor (front service node in our architecture) and stays blocked until all P partitions finish the current query. This is an unrealistic assumption since current systems are implemented using asynchronous multi-threading in each processor/node. In the system used in this work, the flow of queries is not interrupted. Each time an IS partition finishes processing a query, it immediately starts the next one.

The limitations of previous attempt to model vertical search engines show that this problem is not simple to solve. Our proposal is resorting to a more empirical tool but more powerful tool in terms of its ability to model complex systems. Namely we propose using Coloured Petri Nets (CPN) to model vertical search engines computations. To our knowledge this is the first CPN based capacity planning model proposed in the literature for vertical search engines.

3 Modelling a Vertical Search Engine

Our approach is to model query routing through the search engine services. The model main objective is to check whether a given configuration of services is able to satisfy constraints like the following. The services are capable of (1) keeping query response times below an upper-bound, (2) keeping all resources workload below 40% and (3) keeping query throughput at the same query arrival speed. Query throughput is defined as the number of queries processed per unit of time.

In our model we focus on high level operation costs. This is possible because query processing can be decomposed into a few dominant cost operations such as inverted list intersection, document ranking, cache search and update, and blending of partial results. We also represent a few key features of hardware cost and very importantly these features are directly related to the cost dominant primitive operations. In the following we explain each component of our model.

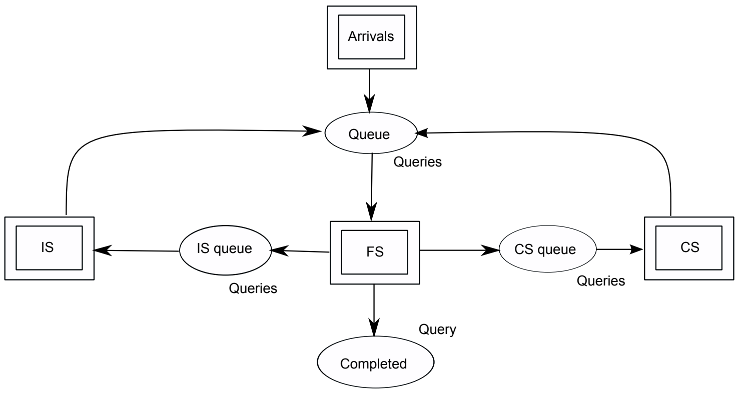


Fig. 2. Vertical Web Search Engine model using CPN

The high level view of our model is shown in Figure 2. Queries arrive with an exponential distribution to the system through the *Arrival* module (the suitability of the exponential distribution for this case has been shown in [4]). We have three more modules which model the front service (*FS*), the index service (*IS*) and the caching service (*CS*). Each module is associated with a socket place. The socket place called *Completed* receives queries that have already been processed.

The query and its top-k document ID results are sent to the user/requester. The *IS Queue* and *CS Queue* are used to communicate queries that travel from the FS service to the IS and CS, respectively. Finally the *Queue* socket place receives new queries and query results from the IS and CS. This query routing based approach can be easily extended to more services as needed.

All service clusters support multi-threading. Namely, each service node has multiple threads. Each thread is idle until a new query arrives to the node. Then the thread takes and processes the query. When finished, the query goes to the next state and the thread checks whether there is any waiting query in the queue of the node. If so, it takes the query to process. Otherwise it becomes idle again.

Each query has four possible states of execution: (1) *q.new* represents arriving queries and have to be processed in the CS cluster, (2) *q.hit* represents queries found in the cache, (3) *q.no_hit* represents queries that have to be processed in the IS cluster, and (4) *q.done* indicates that the query has been finished.

3.1 Modelling the Communication Infrastructure

We apply the communication cost of sending a query through the network. Cluster processors are grouped in racks. We use a network topology that interconnects communication switches and processors that is commonly used in data centers. Namely we use a Fat-Tree communication network [1] with three levels as shown in Figure 3. At the bottom processors are connected to what is called Edge switches. At the top we have the so-called Core switches and in the middle we have the Aggregation switches. Node x sends a message to node y by visiting five switches before reaching its destination. To send a message from node x to node z it has to go through three switches (two Edge switches and one Aggregation switch). Only one Edge switch is visited when sending a message from a node x to other node in the same rack, e.g. node w. An interesting property of this network is that it allows to achieve a high level of parallelism and to avoid congestion. Namely, node x sends a message to node y through path A, meanwhile node x sends a message to node v through a different path (dotted line path in Figure 3).

Due to its complexity, we do not actually simulate the above Fat-Tree protocol in the CPN but we empirically estimate the cost of sending messages

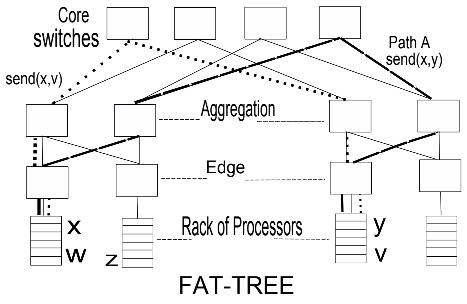


Fig. 3. Fat-tree network

throughout this network. We use this cost in our model to cause transition delays between services. To this end, we run a set of benchmarks programs on actual hardware to determine the communication costs. We then used a discrete-event simulation model of the Fat-Tree, a model that actually simulates the message passing among switches, to obtain an empirical probability distribution related to the number of switch-hops required to go from one service node to another. Therefore, each time we have to send a message from service s_i to service s_j, we estimate the simulation time by considering the number of hops (using the empirical distribution) and the communication cost obtained through the benchmark programs.

We also developed a CPN model for a communication switch for LAN networks which represents the case of a commodity cluster of processors hosting a small vertical search engine (details below).

3.2 Front-Service (FS)

The Front-Service (FS) cluster manages the query process and determines its route. We model the Front-Server cluster as shown in Figure 4 which consists of a number of servers initially idle. The number of replicas is set by the *num_of_FS* parameter. When a query arrives, we take an idle server and increase simulation time using the *timeFS(query)* function. If we are processing a binding of results we apply an average cost of merging the documents retrieved by the IS nodes. Otherwise, we apply an average cost of managing the query. Both costs were obtained through benchmark programs.

If the query execution state is *q.new* we send the query to the CS cluster. If the query execution state is *q.hit* or *q.done* the query routing is finished and we deliver the results to the user. Additionally, *q.done* adds the time required to merge the partial results obtained by the IS cluster. Finally, the query is sent to the IS cluster if the state is *q.no_hit*. The FS does not change the execution state of a query, just decides the route.

3.3 Caching-Service (CS)

The caching-service (CS) keeps track of the most frequent queries and their results. The CPN CS model includes sub-models as shown in Figure 5. In this example we develop a CS with three partitions. Each partition models the time of service and the competition for resources. Namely, the Memcached algorithm can send queries to one of three sets of processors. Each partition is replicated to increase throughput. The *#CS query* variable is used in the model to simulate the query flow through different CS partitions. As we explained, the Memcached algorithm is used to partition the CS cluster. This algorithm evenly distributes the queries among partitions by means of a hashing function on the query terms. Therefore, each CS partition has the same probability of receiving a query. The branch condition in the model is evaluated by comparing a random real number, generated with a uniform distribution in a [0, 1] interval, against the cumulative

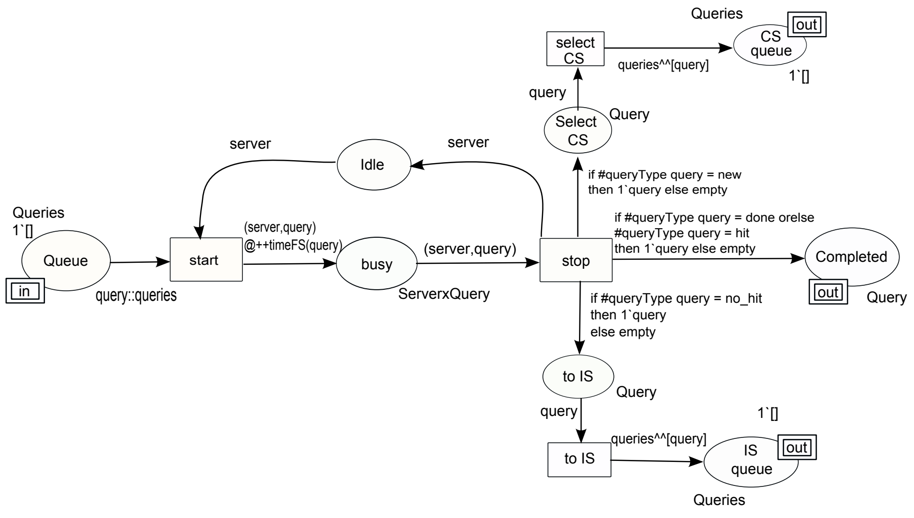


Fig. 4. Front-Service model using CPN

probability of each partition. Namely, if the random number satisfies $r < \frac{1}{3}$, then the query is sent to the first CS partition.

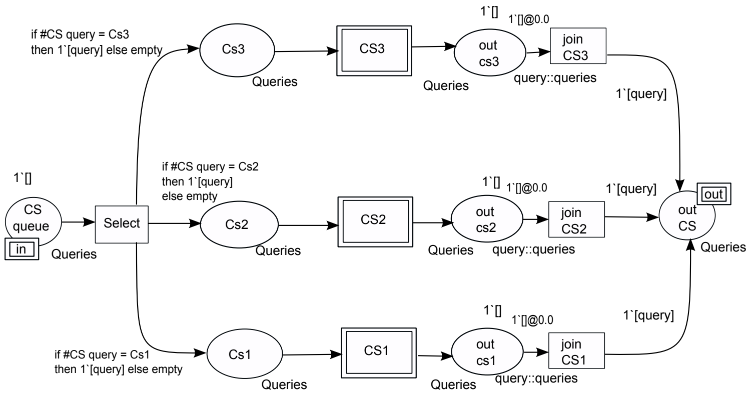


Fig. 5. Caching Service model using CPN

Inside a CS partition we have a CPN multi-server system as shown in Figure 6. When the query arrives to a partition, we select a free server and we increase simulation time using the *timeCS()* function. The number of replicas is set by the *num_of_CS* parameter. In this example, the query reports a cache hit with a probability of 46%. This value can be adjusted according to a desired percentage of hits observed in an actual execution of the query log under study on a LRU cache. The CS cluster also changes the execution state of the query.

We can perform three operations on a CS node: insert a query with their top-k results, update a query priority and erase a query entry. These operations are independent of the lengths of the posting lists. The erase and update operations have constant cost and the insert operation depends on the k size. Therefore, the number of partitions does not affect the average service time in this cluster.

Figures 7 (a) and (b) show the hit ratio obtained by benchmark programs executed using the query log under study as we increase the cache size and the number of partitions. With a memory size of 8GB it is not possible to increase the number of hits significantly beyond 200 CS partitions. On the other hand, with a cache size of 50K cache entries we reach the maximum number of hits with a minimum of 25600 CS partitions.

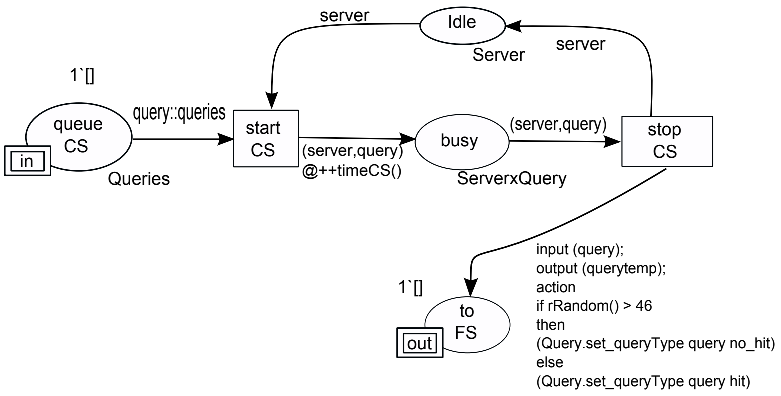


Fig. 6. Multi-server system model for a CS partition

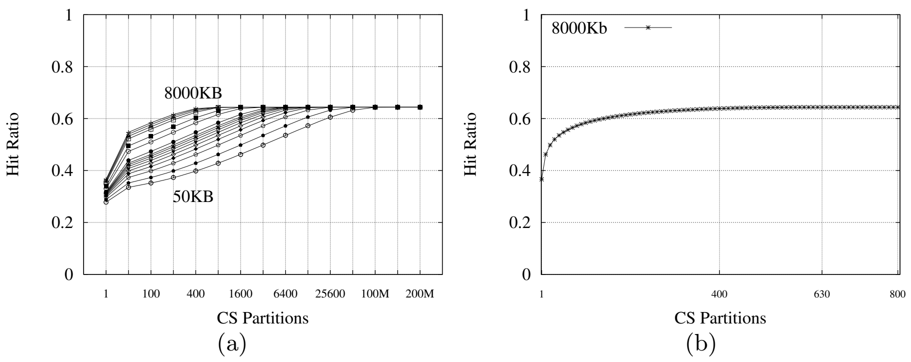


Fig. 7. Cache hits obtained by different number of partitions

3.4 Index-Service (IS)

The Index-Server (IS) cluster accesses the inverted index to search for relevant documents for the query and performs a ranking operation upon those documents. The CPN Index-Server (Figure 9) consists of a number of servers initially idle. Each server processes a query at a time. For a given query every IS partition performs the same operations (intersection of posting lists and ranking) although with different parts of the inverted index. As the inverted index is uniformly distributed among processors all index size partitions tend to be of the same size with an $O(\log n)$ behavior for the maximum size n [3]. We have observed that this $O(\log n)$ feature fades away from the average query cost when we consider systems constructed by using non-blocking multi-threaded query processing.

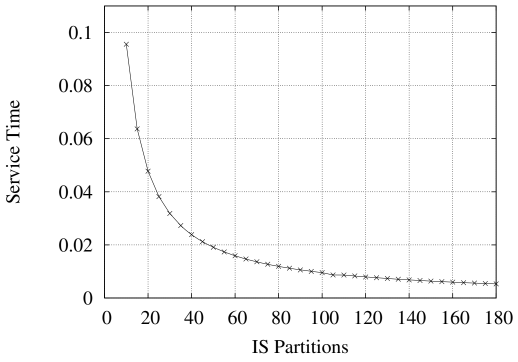


Fig. 8. Average query service times obtained with different IS partitions

The service time of this cluster depends on the posting list size of the query terms. The intersection and ranking operations over larger posting lists require larger service time. Figure 8 shows how the IS service time decreases with more partitions for the query log under study and using a benchmark program that actually implements the inverted file and document ranking process (notice that our CPN model is expected to be used as a tool exploited around an actual vertical search engine, from which it is possible to get small benchmark programs that are executed to measure the different costs on the actual hardware). Almost with 140 partitions the IS cluster cannot improve performance any longer, because the inverted index becomes very small. Therefore, in the CPN model the service time is adjusted according to the number of index partitions IS_p. Each query is associated with a posting list size randomly selected from a set of all possible posting lists sizes of our query log. This value is used together with the number of partitions IS_p to estimate the document ranking cost in the *valuer* variable of Figure 9.

Therefore, from the point of view of the CPN simulation and as it is not of interest to obtain metrics about the quality of results, we can model just one

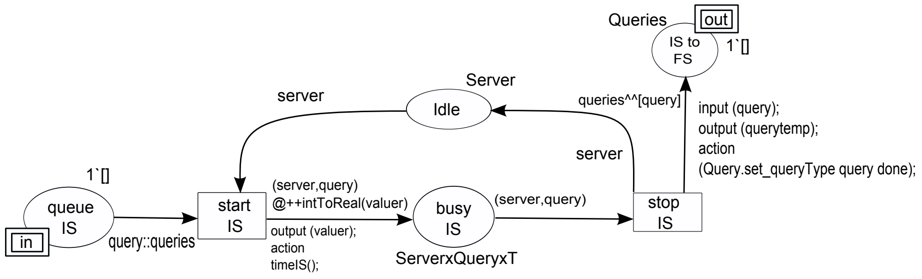


Fig. 9. Index Service model using CPN

IS partition. The number of replicas is set in the global parameter *num_of_IS*. After a query is served in this cluster, it changes its status to *q.done*.

In this work we estimate the cost of the WAND algorithm for document ranking [6]. We use results obtained from the WAND benchmark to study the average running time required to solve a query in the IS cluster. Query running times reported by the WAND algorithm can be divided into two groups: (1) the time required to update a top-k heap, and (2) the time required to compute the similarity between the query and a document. The heap is used to maintain the local top-k document scores in each index node. The similarity between the query and a document is given by the score of a document to a given query.

Figure 10.(a) shows that the total running time reported by the WAND algorithm required to compute the similarity among documents and a query is dominant over the time required to update the heap. Computing the similarity is less expensive, like 15% of the time required to update the heap. But, the number of heap updates is much lower than the number of similarity computations. Figure 10.(b) at left shows the average number of heap updates and similarity computations per query. Each query performs 0.1% heaps updates of the total number of operations with $P = 32$ and top-10 document ID results. This percentage increases to 1% with $P = 256$ because the size of the posting lists are smaller and the WAND algorithm can skip more similarity computations. For a larger top-k the number of heap update operations increases.

Results of Figure 10.(b) at right show the variance of the number of similarity computations. The variance also decreases with more processors, because posting lists are smaller and the upper bound of each term of the query tends to be smaller. Thus all processors perform less similarity computations and tend to perform almost the same amount of work.

4 A Performance Evaluation Study Using the CPN Model

In this section we first validate our model against the execution of a small vertical web search engine. Then we use this model to evaluate the impact in performance

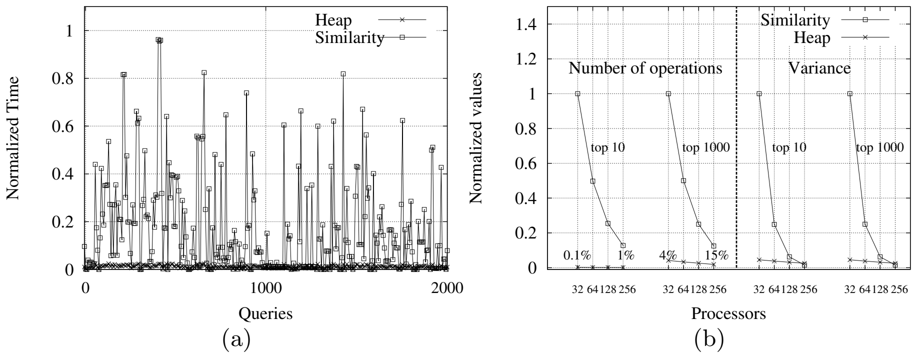


Fig. 10. Benchmarking the WAND document ranking method

of alternative service configurations (number of partitions and replicas). The aim is to illustrate the potential use of the CPN model in a production environment. Below we refer to "workload" to mean average utilization of processors hosting the FS, CS and IS services. Utilization is defined over a given period as the fraction of time that a given processor is busy performing computations.

4.1 Model Validation

We consider a vertical search engine deployed on a small cluster of 30 processing nodes with 4 cores each. We run experiments over a query log of 36,389,567 queries submitted to the AOL Search service between March 1 and May 31, 2006. We pre-processed the query log following the rules applied in [15] by removing stopwords and completely removing any query consisting only of stopwords. We also erased duplicated terms and assumed that two queries are identical if they contain the same words no matter the order. The resulting query trace has 16,900,873 queries, where 6,614,176 are unique queries and the vocabulary consists of 1,069,700 distinct query terms. These queries were also applied to a sample (1.5TB) of the UK Web obtained in 2005 by the Yahoo! search engine, over which a 26,000,000 terms and 56,153,990 documents inverted index was constructed. We executed the queries against this index in order to get the top-k documents for each query by using the WAND method. We set the CPN simulator with the statistics obtained from this log. The simulator supports different query arrival rates λ_0 (new queries per second).

We are interested in two main measures: workload and query response time. In Figure 11 the y-axis shows the relative error (difference in percentage between values) obtained for each measure with the CPN model and the x-axis stands for different services configurations. A configuration is a tuple $\langle FS, CS_p, CS_r, IS_p, IS_r\rangle$ where FS is the number of front service replicas, IS_r represents the number of IS replicas and IS_p the number of IS partitions, the same nomenclature is used for the CS. A low value in the x-axis represents configurations with a small number of processors where workload is about 85%-95%. On the other hand, a high value in the x-axis represents configurations with more processors with processor workload of about 20%-30%.

Figure 11(a) shows results obtained with $\lambda_0 = 1000$. We run different services configurations with a range from 40 to 80 for values of the configuration tuple. The IS workload presents the higher error rate, about 5.3% for configurations with few service nodes. The FS workload measure also presents a maximum error close to 5%. Finally, the CS workload is the closest one to the real result. The maximum error presented by this service is at most 3.5%. Finally, the average query response time presents a maximum error of at most 4%.

Figure 11(b) shows results obtained with a larger $\lambda_0 = 3000$ and a different set of services configurations. We changed the set of services configurations to avoid saturating the system early in the simulation. We increase the range of configuration values from 40 to 120. In this experiment, the maximum error rate of about 5% is reported by the IS cluster. Query response time presents an error rate of at most 3.5%. Therefore, these results verify that our CPN model can predict with an error close to 5% the relevant costs of the vertical search engine under study.

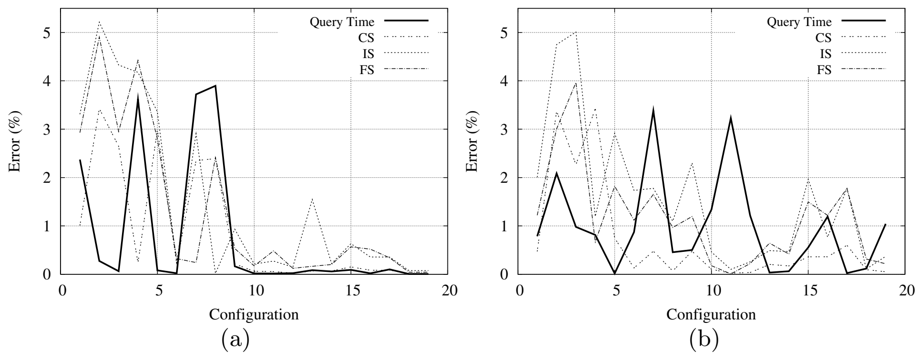


Fig. 11. CPN model validation: (a) query arrival rate $\lambda_0 = 1000$ and (b) query arrival rate $\lambda_0 = 3000$

4.2 Model Assessment

In this section we aim to answer questions like: "Given a number of processors, What is the query traffic that they can support?". To this end we stress our search engine model by increasing the arrival rate λ_0. In this way we can determine the workload supported by the search engine and how average query response time is affected by the increased workload. In the following experiments we set to 8 the number of threads per processor which is consistent with the current hardware deployed in production for the vertical search engine.

Figure 12 shows the results obtained for query traffic ranging from very low (A) to very high (I). We start with $x = A = \lambda_0 = 800$ and we increase the arrival query rate by 400 in each experiment until we reach $x = I = \lambda_0 = 4000$. In Figure 12.(a) we use a total of 355 processors and in Figure 12.(b) we use 463

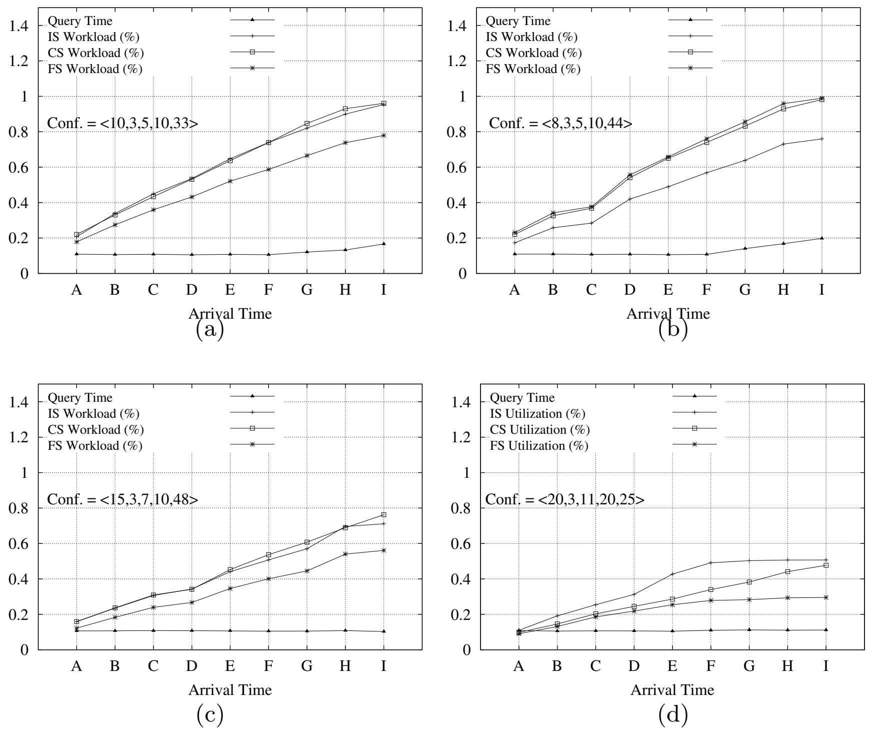


Fig. 12. Workload and average query response time obtained with different configurations. Query traffic varies from low (A) to very high (I).

processors. In both figures, workload is above 80% from $x = G = \lambda_0 = 3200$ but for different cluster services. In the former we reach this percentage of workload with the CS and the IS clusters. In the last, we maintain the number of CS replicas but we increase the number of IS replicas as we decrease the FS replicas. Thus, the FS cluster reaches this percentage of workload instead the IS cluster. As expected service time tends to increase as services clusters are saturated. With more processors, a total of 516 in Figure 12.(c) and 553 in Figure 12.(d), the workload is kept below 80%. There is no saturation and therefore query response time is not affected.

With these experiments we can answer our question about the query traffic that can be supported by a given services configuration. For the first two configurations in Figure 12.(a) and Figure 12.(b) it is clear that with $x = C$ we get a workload close to 40%. For the two last figures, we get a workload close to 40% in $x = E$. Beyond these points we cannot meet the second constrain (keep all resources workload below 40%). This constrain has to be satisfied to support suddenly peaks in query traffic. The other two constraints (keep query response time below an upper-bound, and keep query throughput at query arrival speed) cannot be guaranteed with a workload above 80%-90%. From this point on, the system is saturated and queries have to wait for too long in services queues.

4.3 Evaluation Using a Switched LAN

In this section we illustrate with an example that the proposed CPN model can easily incorporate additional components. We show this by evaluating our search engine model with a different kind of network. To this end, we replace the Fat-tree network by a switched LAN network as explained in [27]. In Figure 13 we add a new module *WS* connecting all services queues. We also have to add three more places (*IS SW, CS SW, FS SW*) to make this connection feasible. Inside the *SW* module we have connection components. Each component has input/output ports and internal buffers represented as places. There are two transitions: one to model the processing of input packages and one to model the processing of output packages. We have one component for each service cluster.

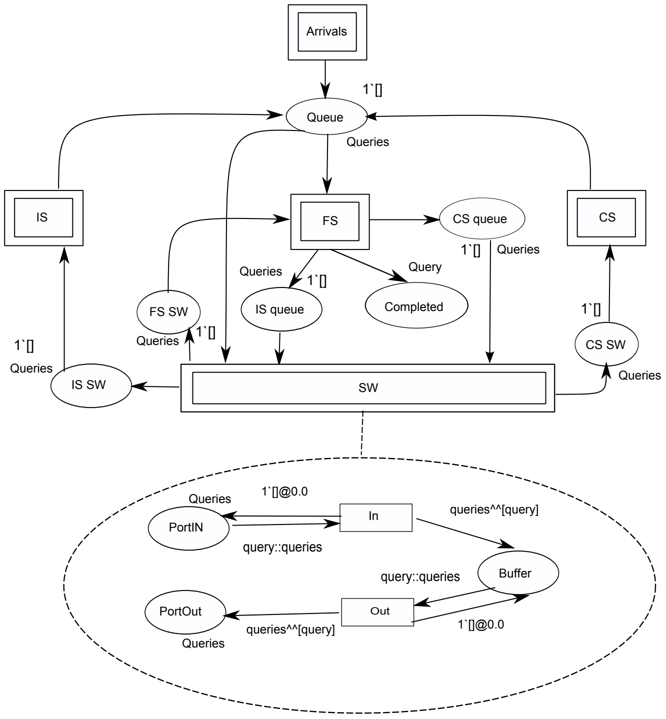


Fig. 13. Web search engine modeled with a switched LAN network

To evaluate the switched LAN network we run some benchmark programs to determine the time of sending a message from one service node to another over a Switch Linksys SLM2048 48-ports. We compare it against a Fat-Tree constructed using the same kind of 48-ports switches. Figure 14.(a) compares CPN simulations using the two networks. The results show that using a Fat-tree network helps to reduce the average response query time as we increase the query arrival rate. With a maximum query arrival rate of $\lambda_0 = 4000$, the search engine using the Fat-tree network obtains a gain of 10%. But, the LAN network has a second effect over the

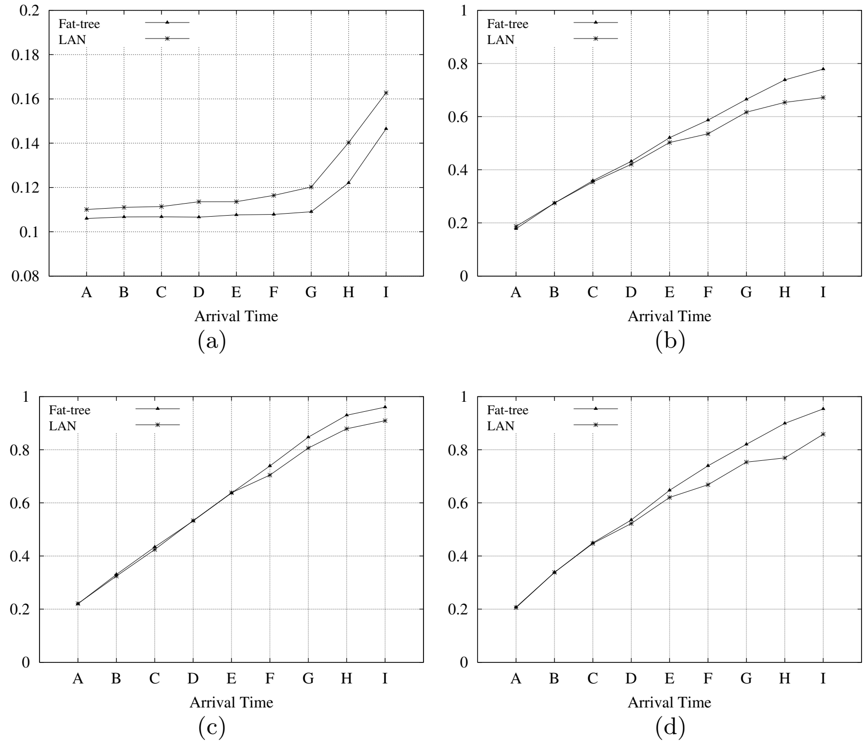


Fig. 14. Results obtained with two different kind of networks. (a) Average query response time. (b) Front-Service workload. (c) Caching-Service workload. (d) Index-Service workload.

search engine. Queries expend more time inside the network and inside the queues connecting services with the network than inside the services clusters. Therefore, using a switched LAN network the services workload placed on processors tends to be lower than in the case of the Fat-tree. In Figure 14.(b) the FS cluster reports 13% higher workload using the Fat-tree. In Figure 14.(c) this difference is only 5% for the CS cluster. In Figure 14.(d) we show that the IS cluster gets 12% lower workload using the switched LAN network. These results were obtained with a services configuration $\langle 10, 3, 5, 1, 33 \rangle$.

5 Conclusions

We have presented a model for predicting performance of vertical search engines which is based on a Coloured Petri Net (CPN). We illustrated the use of the CPN model through the execution of a corresponding CPN simulator in the context of a specific vertical search engine currently deployed in production. From the actual implementation of the search engine, several small pieces of code are extracted to construct benchmark programs. These programs are executed on the same search engine hardware and the results enable the setting of the simulator parameters. Typically vertical search engines are built from a software architecture that makes it simple to generate appropriate benchmark programs for the CPN simulator.

After validating the model results against the search engine results, we used the CPN simulator to evaluate relevant performance metrics under different scenarios. This study showed that the proposed CPN model is suitable for solving the problem of determining the number of cluster processors that are required to host the different services that compose the respective search engine. This is a relevant question that data center engineers must answer in advance when a new vertical search engine instance is required to serve a new product deployed in production, or in accordance with an estimation of query traffic growth for the next term, to determine how big the different services must be in order to decide the amount of hardware required to host the new search engine instance.

An advantage of the proposed model is that it has been designed in a modular manner so that new components (services) can be easily accommodated. We illustrated this feature by evaluating the impact of introducing an alternative communication network technology. The enhanced model was simple and fast to formulate and experimentation was quickly conducted to evaluate the performance metrics for the new case at hand.

Acknowledgment. This work has been partially supported by FONDEF D09I1185 R&D project.

References

1. Al-Fares, M., Loukissas, A., Vahdat, A.: A scalable, commodity data center network architecture. SIGCOMM 38, 63–74 (2008)
2. Arlitt, M., Krishnamurthy, D., Rolia, J.: Characterizing the scalability of a large web-based shopping system. J. of ACM Trans. Internet Technol. 1, 44–69 (2001)
3. Badue, C.S., Almeida, J.M., Almeida, V., Baeza-Yates, R.A., Ribeiro-Neto, B.A., Ziviani, A., Ziviani, N.: Capacity planning for vertical search engines. CoRR, abs/1006.5059 (2010)
4. Badue, C.S., Baeza-Yates, R.A., Ribeiro-Neto, B.A., Ziviani, A., Ziviani, N.: Modeling performance-driven workload characterization of web search systems. In: CIKM, pp. 842–843 (2006)
5. Baeza-Yates, R., Ribeiro-Neto, B.: Modern Information Retrieval. Addison-Wesley (1999)
6. Broder, A.Z., Carmel, D., Herscovici, M., Soffer, A., Zien, J.Y.: Efficient query evaluation using a two-level retrieval process. In: CIKM, pp. 426–434 (2003)
7. Cacheda, F., Carneiro, V., Plachouras, V., Ounis, I.: Network Analysis for Distributed Information Retrieval Architectures. In: Losada, D.E., Fernández-Luna, J.M. (eds.) ECIR 2005. LNCS, vol. 3408, pp. 527–529. Springer, Heidelberg (2005)
8. Cacheda, F., Carneiro, V., Plachouras, V., Ounis, I.: Performance analysis of distributed information retrieval architectures using an improved network simulation model. Inf. Process. Manage. 43(1), 204–224 (2007)

9. Cacheda, F., Plachouras, V., Ounis, I.: Performance Analysis of Distributed Architectures to Index One Terabyte of Text. In: McDonald, S., Tait, J.I. (eds.) ECIR 2004. LNCS, vol. 2997, pp. 394–408. Springer, Heidelberg (2004)
10. Cacheda, F., Plachouras, V., Ounis, I.: A case study of distributed information retrieval architectures to index one terabyte of text. Inf. Process. Manage. 41(5) (2005)
11. Cahoon, B., McKinley, K.S., Lu, Z.: Evaluating the performance of distributed architectures for information retrieval using a variety of workloads. ACM Trans. Inf. Syst. 18, 1–43 (2000)
12. Chowdhury, A., Pass, G.: Operational requirements for scalable search systems. In: CIKM, pp. 435–442 (2003)
13. Couvreur, T.R., Benzel, R.N., Miller, S.F., Zeitler, D.N., Lee, D.L., Singhal, M., Shivaratri, N.G., Wong, W.Y.P.: An analysis of performance and cost factors in searching large text databases using parallel search systems. Journal of The American Society for Information Science and Technology 45, 443–464 (1994)
14. Fitzpatrick, B.: Distributed caching with memcached. J. of Linux, 72–76 (2004)
15. Gan, Q., Suel, T.: Improved techniques for result caching in web search engines. In: WWW, pp. 431–440 (2009)
16. Jensen, K., Kristensen, L.: Coloured Petri Nets. Springer, Heidelberg (2009)
17. Jiang, G., Chen, H., Yoshihira, K.: Profiling services for resource optimization and capacity planning in distributed systems. J. of Cluster Computing, 313–329 (2008)
18. Lin, W., Liu, Z., Xia, C.H., Zhang, L.: Optimal capacity allocation for web systems with end-to-end delay guarantees. Perform. Eval., 400–416 (2005)
19. Lu, B., Apon, A.: Capacity Planning of a Commodity Cluster in an Academic Environment: A Case Study (2008)
20. Marin, M., Gil-Costa, V.: High-performance distributed inverted files. In: Proceedings of CIKM, pp. 935–938 (2007)
21. Menasce, D.A., Almeida, V.A., Dowdy, L.W.: Performance by Design: Computer Capacity Planning. Prentice Hall (2004)
22. Moffat, A., Webber, W., Zobel, J.: Load balancing for term-distributed parallel retrieval. In: SIGIR, pp. 348–355 (2006)
23. Moffat, A., Webber, W., Zobel, J., Baeza-Yates, R.: A pipelined architecture for distributed text query evaluation. Information Retrieval 10(3), 205–231 (2007)
24. Reiser, M., Lavenberg, S.S.: Mean-value analysis of closed multichain queuing networks. J. ACM 27(2), 313–322 (1980)
25. van der Aalst, W., Stahl, C.: Modeling Business Processes – A Petri Net-Oriented Approach. MIT Press (2011)
26. Wang, H., Sevcik, K.C.: Experiments with improved approximate mean value analysis algorithms. Perform. Eval. 39, 189–206 (2000)
27. Zaitsev, D.A.: An evaluation of network response time using a coloured petri net model of switched lan. In: Proceedings of Fifth Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools, pp. 157–167 (2004)
28. Zhang, C., Chang, R.N., Perng, C.-S., So, E., Tang, C., Tao, T.: An optimal capacity planning algorithm for provisioning cluster-based failure-resilient composite services. In: Proceedings of the 2009 IEEE International Conference on Services Computing, SCC 2009, pp. 112–119 (2009)
29. Zhang, J., Suel, T.: Optimized inverted list assignment in distributed search engine architectures. In: IPDPS (2007)
30. Zobel, J., Moffat, A.: Inverted files for text search engines. J. of CSUR 38(2) (2006)

An Infrastructure for Cost-Effective Testing of Operational Support Algorithms Based on Colored Petri Nets

Joyce Nakatumba, Michael Westergaard*, and Wil M.P. van der Aalst

Eindhoven University of Technology, The Netherlands
{jnakatum,m.westergaard,w.m.p.v.d.aalst}@tue.nl

Abstract. *Operational support* is a specific type of process mining that assists users while process instances are being executed. Examples are *predicting* the remaining processing time of a running insurance claim and *recommending* the action that minimizes the treatment costs of a particular patient. Whereas it is easy to evaluate prediction techniques using cross validation, the evaluation of recommendation techniques is challenging as the recommender influences the execution of the process. It is therefore impossible to simply use historic event data. Therefore, we present an approach where we use a *colored Petri net* model of user behavior to drive a real workflow system and real implementations of operational support, thereby providing a way of evaluating algorithms for operational support before implementation and a costly test using real users. In this paper, we evaluate algorithms for operational support using different user models. We have implemented our approach using Access/CPN 2.0.

1 Introduction

Some business processes are unstructured or only very loosely structured. This is the case where a process has never been formalized or where the process requires a lot of freedom. Examples of such processes are processes in small very agile companies, or processes in disaster handling or healthcare, where flexibility and experience plays a more prominent role than a strictly structured process. While such freedom may be good for an experienced user, it may not provide enough support for less experienced users. Using process mining, it is possible to provide *operational support* [3] for running processes of this kind. Under operational support, users are provided with on-line information about the running process, and can even be given recommendations about the next actions to be taken in order to arrive at a goal [15].

In [11], we defined a meta-model for operational support. Here, a client sends a *partial execution trace* along with a *query* to an operational support service.

* This research is supported by the Technology Foundation STW, applied science division of NWO and the technology program of the Dutch Ministry of Economic Affairs.

A query is simply a question to which a *response* is received. Operational support allows four types of queries. A *simple* query checks the performance of the current partial execution trace, for example, what is the total time since the start of current execution? A *compare* query compares the performance of the current partial trace to other similar traces. For example, is the execution time of the current trace to this point higher or lower than the average? A *predict* query looks into the future of traces similar to the current and uses that to provide predictions about the current trace. For example, what is the expected total execution time for this trace? Finally, a *recommend* query gives the best possible next action to be done based on the current partial trace. For example, what is the best action to execute in order to complete the execution as fast as possible?

It is easy to evaluate algorithms for the first two types of queries (simple and compare). These are basically lookup functions combined with standard operators computing average, variance, etc. As shown in [1,4] it is also possible to evaluate predict queries using cross-validation. For example, when using k-fold cross-validation, the set of process instances is partitioned in k parts. $k-1$ parts are used to learn a predictive model. The instances in the remaining part are used to evaluate the quality of the predictive model. Because historic data is used, it is possible to compare the predicted value with the real value. This experiment can be repeated k times thus providing insight into the quality of the predictive algorithm.

Algorithms providing recommendations are much more difficult to evaluate. Since recommendations influence the execution, it is impossible to directly use historical data. Users will change their behavior based on these recommendations, so it is impossible to simply use the observed behavior where users got no recommendations.

Here, we introduce a general setting for testing recommendations as shown in Fig. 1. At the bottom right we have a User. The user is executing a process. The process may be implemented using a Workflow System and as shown here or it may be ad-hoc. The user consults Operational Support to get advice about which step to execute. The idea we present here is to model the user using a colored Petri net (CPN) [8] model and have that model interact directly with a real workflow system, i.e., Declare [2,7] and real implementations of operational support in the ProM framework [13,19]. That way we do not need real users, making the approach much more affordable, yet still interact with real systems, so we get realistic results. Using real systems instead of modeled counter-parts also makes it much easier to do the modeling, as we only have to focus on the user behavior, and not on replicating already existing systems and algorithms. Our approach also allows rapid prototyping of algorithms by implementing them using a CPN model and directly integrating them in a real tool for operational support. We can then use the algorithms directly in tools acting as clients for operational support, including workflow systems and our testing platform.

The contribution of this paper is two-fold: First and most importantly, we present the test suite modeled in colored Petri nets that provides a means to

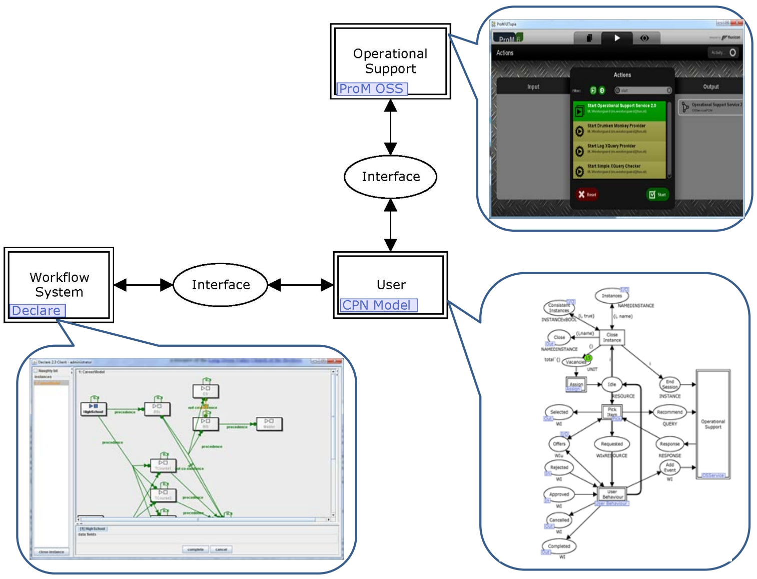


Fig. 1. Abstract Testing Platform

test simple algorithms for operational support in a cost-effective way. The algorithms we present here are not intended as real examples of algorithms for providing operational support, but only to illustrate the test-suite. Second, we provide a framework based on colored Petri nets making it very easy to prototype algorithms for operational support. Through log generation of the CPN model, we are able to also provide an evaluation of simple recommendation algorithms for operational support in a simple but non-trivial setting.

The term *operational support* refers to a collection of process mining techniques [1] executed while people are still working on cases. Several papers describe techniques to predict the remaining total execution time of running cases. See [3,4] for pointers to such techniques. Another type of operational support, more relevant for this paper, is providing *recommendations*. In [16] simple ad-hoc models are created to support recommendation. In [17], case-based reasoning is applied to find similar cases. See [14] for a more general overview of recommenders. To the best of the authors' knowledge, there is no preexisting work on unified testing recommenders except for ad-hoc testing of individual recommenders compared to no support.

In [11] a generic framework for operational support based on queries is proposed. This is used in this paper. In [19] the operational support service of ProM is modeled and analyzed using CPN Tools [6]. The integration of CPN Tools with other components was described in [18]. In [9] it was shown how CPN components and workflow components can be exchanged for testing and simulation. This approach will also be followed in this paper.

We use Declare [2,7] as an example of a workflow system. It is selected because it allows more flexibility than procedural approaches to process modeling and therefore benefits from recommendation techniques.

The remainder of the paper is organized as follows: In Sect. 2, we provide the background needed to understand the remainder of the paper including a running example. In Sect. 3, we discuss the user model, focusing on the modeling of the time needed to execute tasks. In Sect. 4, we discuss the recommendation algorithms tested in this paper including presenting a generic CPN model that can be used as a starting point for rapid prototyping of operational support algorithms. Section 5 provides a description of the experiments carried out to evaluate the algorithms described using the various user models. Finally, Sect. 6 summarizes our conclusions and provides directions for future work.

2 Background

In this section we provide the background needed to understand the rest of this paper. We briefly introduce the Declare language which is an example of a workflow language that we use for modeling our running example. We also introduce an architecture for operational support. We summarize the Access/CPN 2.0 [18] library for running a CPN model together with software components.

Running Example and Declare. Consider the example in Fig. 2, modeling a study process. This example is created in Declare [7], which is a workflow system based on a declarative language. Compared to conventional procedural workflow systems, Declare allows for much more flexibility [2]. In Declare, *tasks* are shown as rectangles and can initially be executed in any order. Tasks are constrained by *constraints*, shown as arcs. We shall not go into details about the constraints of Declare but refer the interested reader to [2,7]. Here, we just supply an abstract overview of the behavior of the model. Basically, a student can choose either an academic or a practical path to a degree. A student can initially either choose to go to HighSchool or to get a job (Work). Going to high school allows students to be admitted for a BSc (the academic path). Alternatively, a student may decide to get a job. Having had a job allows the student to enter two practical supplementary courses (PCourse1 and PCourse2). In order to be admitted to the four theoretical courses (TCourse1–TCourse4), a student must both have had a job and also been to high school. Having completed all six supplemental courses is a prerequisite to Qualify for starting a master's study. A student is also allowed to start a master's study if he has completed a BSc. Out of the two master's degrees offered, only one can be completed (for financial reasons). Only after completing a master's degree in business information systems (MSc, BIS) can a student become a true Master of BPM.

In our example, we can optimize towards at least two goals: getting a master's degree as fast as possible or becoming a master of BPM. The difficulty for a student is that he has at any point a lot of freedom. For example, he can at any point in time decide to get a job which may open new possibilities. During

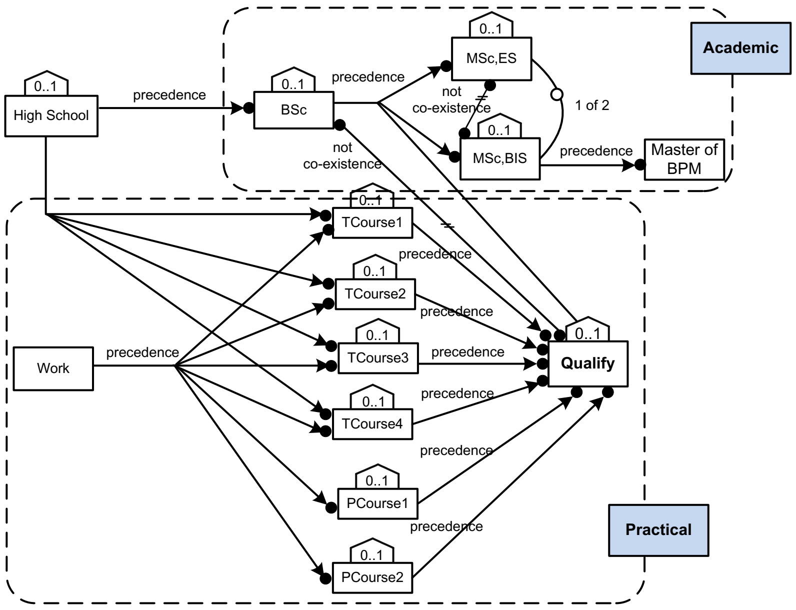


Fig. 2. A study process model in Declare

the execution many activities may be allowed (e.g., after going to high school and having a job, BSc, the 6 courses and Work are allowed actions), making it difficult to select the best action to take. Operational support aims to assist users in making such decisions. For example, based on historic information we can recommend particular actions in a given context.

Operational Support Architecture. Operational support can be implemented in many ways. For example, one way is just to suggest executing a random task and another way is to look at what students did before. In order to support any present and future algorithms in a coherent way, we use the architecture for operational support shown in Fig. 3. Here, a Client communicates with a Workflow System and with the operational support service (OS Service; OSS in the following). The OSS forwards requests to a number of operational support providers (OS providers; providers in the following), which may implement different algorithms. The OSS receives responses from the providers which it sends back to the Client. In [19] we

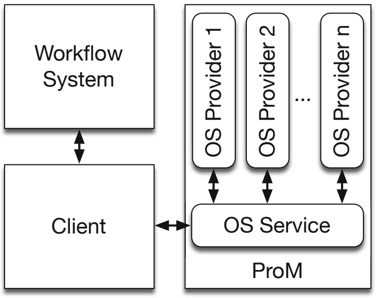


Fig. 3. Architecture of the operational support in ProM

```
public interface Provider extends Serializable {
    boolean accept(Session s);
    void destroy(Session s);

    <R, L> Recommendation<R> recommend(Session s,
                                XLog availableItems, L query);

    void updateTrace(Session session, XTrace trace);
}
```

Listing 1. Provider interface

presented a protocol and architecture making it possible to access different algorithms using a common protocol and this architecture. In [11] we defined a common meta-model for operational support, allowing a common interface to all algorithms.

In order to implement an algorithm for the operational support service, we need to implement the interface in Listing 1. The interface also has methods for the other kinds of queries, but we have hidden them as we are not interested in them here. accept and destroy control the life-cycle of the provider, and recommend handles actual queries. Queries get a set of all availableItems to pick among and a query to optimize towards. The result is a Recommendation, which basically is an event recommended to execute. updateTrace is called whenever the client has executed more events. All methods have an additional session parameter which can be used to store case-local data.

Cosimulation Using Access/CPN 2.0. Our goal is to take a model similar to the one in Fig. 1 and refine the User using a sub-module described as a colored Petri net. This is already supported by CPN Tools [6], a tool for editing and analysis of colored Petri nets. Furthermore, we want to use the actual Declare workflow system as a replacement for the substitution transition Workflow System and the actual implementation of operational support in ProM as a replacement for the Operational Support substitution transitions. Access/CPN 2.0 [18] is a library for interaction between CPN models and Java programs, and supports exactly this kind of interaction.

Using Access/CPN 2.0, it is possible to implement a simple interface and have the library run a *cosimulation*, where the model is executed and synchronized with the Java code.

3 User Behavior Modeling

In this section, we show how we model user behavior. Our model is a concrete implementation of the abstract testing platform in Fig. 1. The model is parameterized and allows different user behaviors. While we allow configurability of probabilities of completing or cancelling a current task, more interesting configuration options include which timing model to use and whether a user uses operational support. Our entire model comprises 14 pages, 127 places, and 41 transitions.

The top level of our model can be seen in Fig. 4. It consists of the Workflow System (left) and the User (right). The workflow system has exposes a number of Instances of the process to be executed (cf. instance 1 for the Study process in Fig. 4). For each instance, it also indicated whether the instance is Consistent, i.e., if it can safely be terminated. Furthermore, the workflow system also exposes a number of Offers which are concrete tasks that can be executed by users. For example, from the study model shown in Fig. 2 initially the possible offers for instance 1 are HighSchool and Work as seen as tokens on the Offers place. A user can pick an offer and inform the workflow system that is has Selected that work item. The workflow system then either approves or rejects the request. If the request is Rejected, it is dropped. Otherwise if the request is Approved, the user starts working, for example, from Fig. 4 HighSchool has been approved by the workflow system and is seen as a token on the Approved place. At some point, the user either cancels work or completes it, i.e., the workflow system is informed using Cancelled and Completed places. When an instance is consistent, i.e., all the required work items offered to the user have been completed then the user can Close the instance.

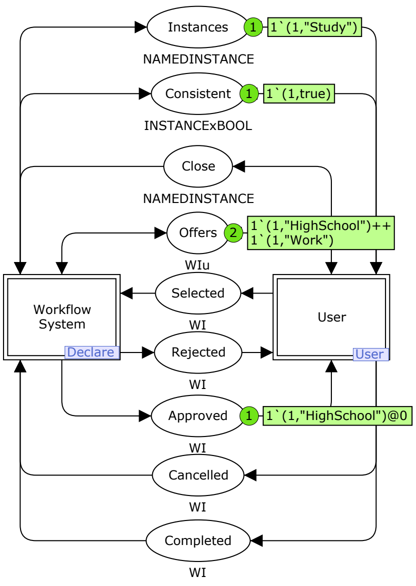


Fig. 4. Workflow system and user

3.1 User Model

A user is modeled as shown in Fig. 5. Users are generated on-demand (depending on the instances that have sent from the workflow system) by the Assign page, which generates users as needed up to a certain threshold (Vacancies). A user starts off being Idle. An idle user can select to Pick Item and start working. This can be done using the aid of Operational Support. After a task has been Requested, it can be executed (or cancelled), making sure to inform the workflow system accordingly. Alternatively, an idle user can Close a consistent instance (i.e., an instance where all the required work items have been completed).

The operational support service can handle recommendation requests (Recommend) and provide Responses. Furthermore, the OSS can be notified when the user has decided to End Session, which is useful for clean-up. After a user has executed an event it is sent to Add Event to allow the operational support service to construct an execution trace. The operational support service is implemented

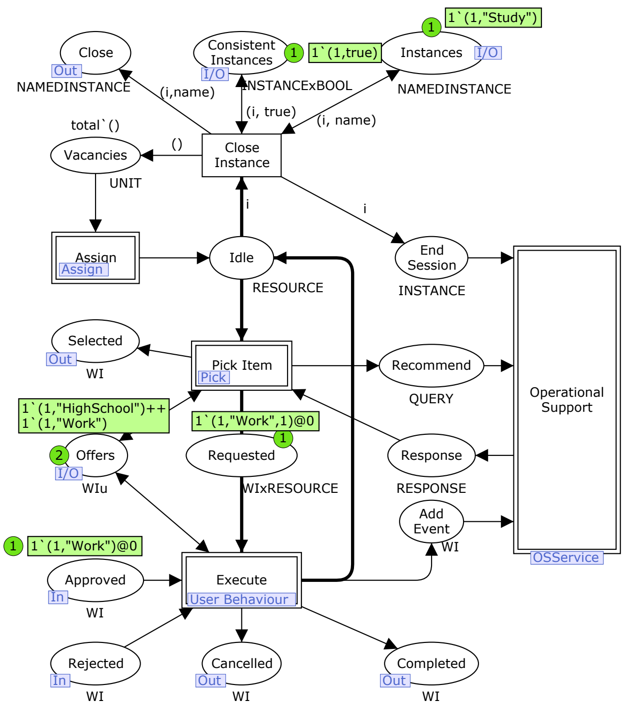


Fig. 5. User model

as a Java module using Access/CPN 2.0 [18]. It comprises 683 lines of code, 227 of which handles a very flexible interface to CPN models allowing formatting of queries in many ways and 143 lines of GUI integration code, making the actual interface just over 300 lines.

Pick Item Model. In Fig. 6, we see that an idle user first needs to decide whether to use support or not. We do this before actually asking for support due to efficiency of simulation. While a more realistic scenario would be to ask for support and use that as a guide, we can see this as just another algorithm for support, and we use the No Support to model a completely clueless user picking at random. The probability for whether support is used or not is configurable. No matter which branch we pick, we end up with a new work item on Selected and transferring control to Requested. It is only when we use support that we send Recommend requests to operational support and get Responses.

When we use operational support, we need a list of all enabled events for a given instance. We thus do as in Fig. 7: we first Select Instance, i.e., which instance of the process to work on. Then we build a list of all events enabled in that instance (Populate Offers for the Selected Instance). From Fig. 7, Selected Instance will be populated by High School and Work. When we have added all events to the list, we can Perform Query, sending the list of enabled events to

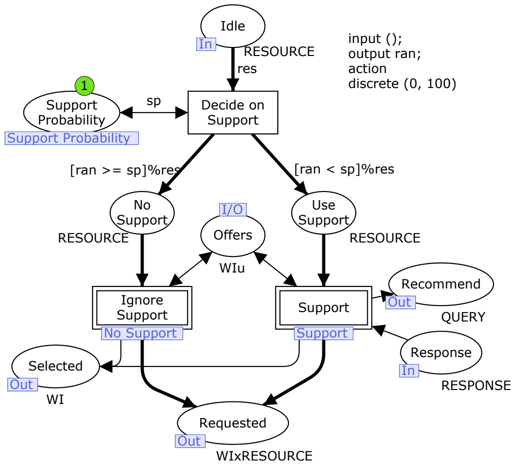


Fig. 6. Pick item model

Recommend of the operational support service (cf. token on the Recommend in Fig. 7). We are then Waiting for a Response, and when it arrives, we blindly Pick Recommended, inform the workflow system of which task we have Selected and transition to the Requested state. For the modelling notations we use here are discussed in [5].

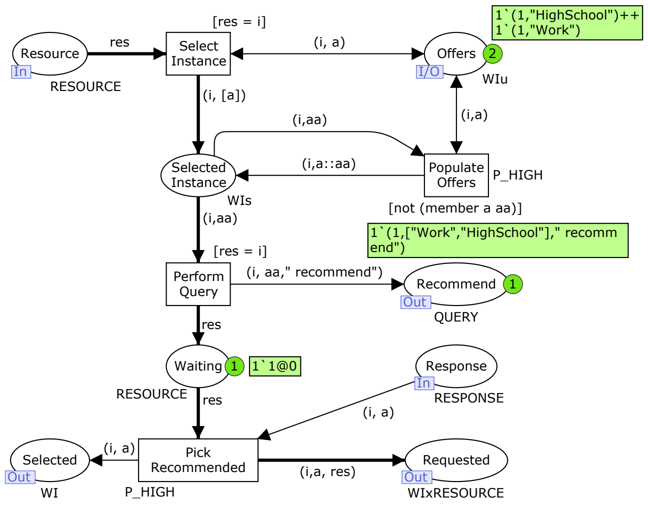


Fig. 7. Implementation of picking using operational support

Task Execution. Task execution (Fig. 8) at the abstract level starts when a task is Requested. If the workflow system Rejected the request, it is Aborted and the user returns to the Idle state. If the request is Approved, it is Executed, the workflow system is informed of success (Completed) or failure (Cancelled), and the operational support service is notified if a new task was executed (Add Event). We have four different implementations of Execute: one with constant execution times, one where execution times are sampled from a probability distribution, one where execution time is dependent on the previously executed task, and one where execution time is dependent on the stress level of the user.

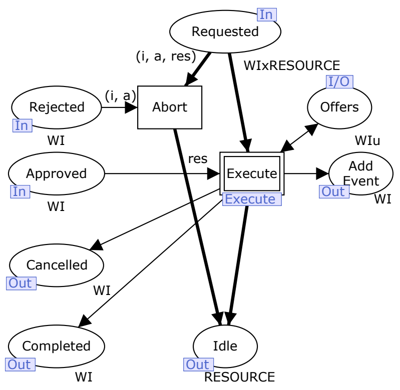


Fig. 8. Abstract task execution

Constant Time or Sample from Probability Distribution. The first two implementations can be treated together as constant time can be viewed as a specific probability distribution. Here, we assume that the execution time for tasks is independent of what was done before and the stress level of the user. Thus, we start in the Requested state in Fig. 9. Due to the timing model of CPN models, we need to make the decision of whether to Cancel Work or to Complete Work as soon as we Start Work. Thus, we pick a random number and if it is below a configurable threshold (Cancel Rate) then Cancel is selected, otherwise In Progress (cf. user 1 is executing Work in Fig. 9). This module is only enabled if the Timing Model is PROB. We get the timing information from shared Time Database and Cancel Database, which contain the timing information for successfully executing and the penalty for cancelling an activity. If we Cancel Work, we inform the workflow system and go to the Idle state, and when we Complete Work, we inform the operational support service as well as the workflow system before transitioning to the Idle state. The transitions all have a guard binding transtype, which indicated when a work item is started, and either cancelled or completed. This is used to subsequently import a simulation log into ProM for further analysis.

Batch Processing. The idea of this timing model is that if one executes similar tasks one after another, then one becomes faster due to step-up time reduction and learning effects. To model this "conveyor belt effect", we need to keep track of the last executed task and how many times we have executed the same task. In our example, we consider all the practical courses to be similar enough to use batch processing and all theoretical courses as well. In Fig. 10 we see the start of the module for batch processing (the remainder is the same as in Fig. 9). Instead of just one start transition, we now have two: one to Start New Work and

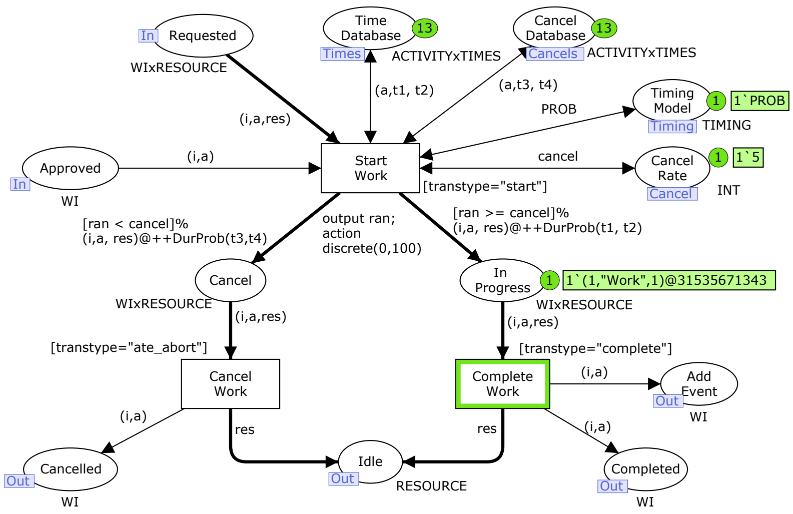


Fig. 9. Task execution model based on a probability distribution

one to Start Batch Work. The apparent complexity is due to the two transitions doing almost the same. Both have access to the two configuration options and two databases as the Start Work transition in Fig. 9. The place Last keeps track of which task we executed last and how many times we have executed a similar task. We read the last task from Last and compare it to the current task to execute. If they are the same, we Start Batch Work and if they are not the same, we Start New Work. In both cases, we update Last accordingly and if we execute batch work, we let the timing be dependent on how many times we have executed the same task.

Execution Time influenced by Workload. According to "Yerkes-Dodson Law of Arousal", the execution speed increases as the stress increases up to a certain optimal level beyond which the performance decreases [10,12,20]. In our model we let the execution time be dependent on the number of tasks offered in the queue for a user. This can be modeled as in Fig. 11. Like for batch processing, we only see the initial fragment as the remainder is the same as in Fig. 9. The basic idea is that we need to count how many tasks are in Offers. This is the same construction as we used to build a list of all tasks for operational support in Fig. 7, and has therefore been hidden in a substitution transition Count Tasks for legibility. Otherwise, Start Work is the same as in the simple case, except we compute the execution time with an extra parameter, namely the number of available tasks (ct) on the Count place for each instance (i).

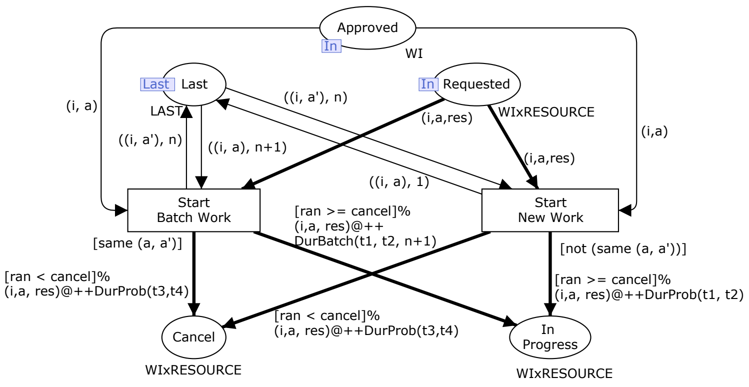


Fig. 10. Task execution model implementing batch processing

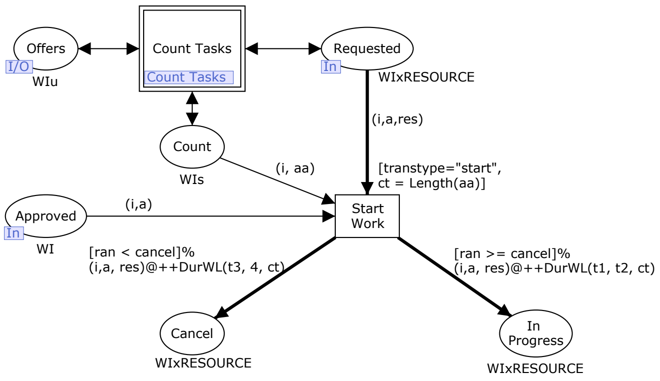


Fig. 11. Task execution taking workload into account

4 Recommendation Algorithms

In this section, we describe the four recommendation algorithms evaluated using simulation. Two of the algorithms are completely general and require no configuration, one is general but requires configuration, and one only works for our running example.

We can implement an algorithm by directly implementing the interface in Listing 1. For simple algorithms, this includes a lot of overhead, which may not be needed for quick testing. For this reason, we have created a generic implementation of the Provider interface using Access/CPN 2.0. This just requires that implementers make a simple CPN model. The generic connector comprises 505 lines of Java code including 206 lines of GUI integration code, making for just under 300 lines of logic.

To make it easy for implementers to get started, we have developed a template model which can be used to very quickly prototype operational support algorithms. The model comprises 7 pages, 30 places and 12 transitions, but a user typically only has to worry about a single page. In the remainder of this section, we introduce our generic template model, how to implement the two simplest providers using this framework, we give a brief overview of a more advanced provider and show how we developed a provider specific for our running example. The reason for going through the operational support service instead of just incorporating the providers directly in the model is to improve reusability. First of all, we can use any provider from any model and do not have to copy one algorithm from one test model to another. Second, we can use the algorithms immediately and directly from any tool allowing operational support. This decoupling makes it easy to test algorithms on humans interacting with a workflow system if we do not want to just rely on simulation results.

4.1 Provider Model

The provider model quite closely reflects the interface in Listing 1. At the top level (Fig. 12) we handle Accept calls (corresponding to the accept method) and the kind of query (corresponding to recommend). The updateTrace method is not represented explicitly; instead the Traces place contains all currently active partial traces. The destroy method is not necessary as resource management is handled by the plug-in.

Provider setup just decides whether to accept or ignore a session. The default implementation unconditionally accepts sessions. The Query module separates each of the four kinds of queries to their own pages.

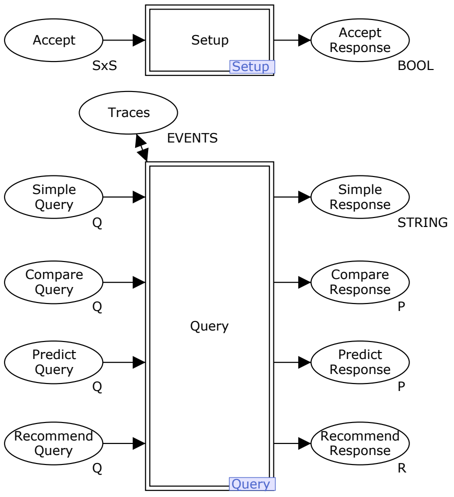


Fig. 12. Top model of a provider

Random Provider. The random provider just recommends a random enabled task from the list of available tasks. Thus, we expect this provider to have the same behavior as using no operational support at all and is put here as a baseline to compare with the more advanced recommendation techniques. The full implementation is shown in Fig. 13. We receive a request on Recommend Query. The request contains evts, a list of available tasks. Our response is just a random event picked using pickRandomEvent if it exists, or a dummy response otherwise (so we always provide an answer).

Batch Provider. The batch recommender always recommends, if possible, the same event as the one executed last. If this is not possible, it just recommends a random event. This is intended to work together with the batch timing, where

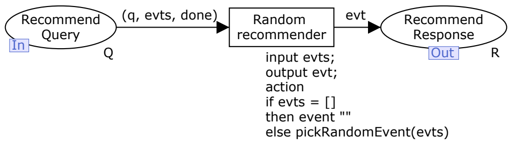


Fig. 13. Random recommender

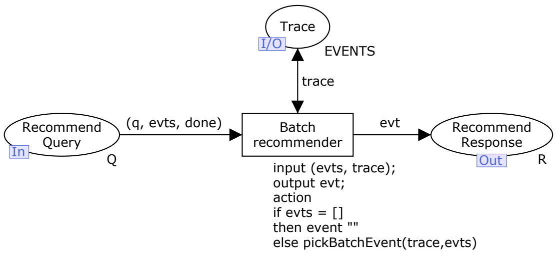


Fig. 14. Batch recommender

executing similar tasks together is faster than executing them interleaved with others. The implementation of the batch provider shown in Fig. 14 is very similar to the random provider, except we now also make use of the execution Trace and use the pickBatchEvent function to pick the event that is similar to the last executed one if one exists, and otherwise a random event.

Model-Specific Provider. The authors are very familiar with the running example, and therefore have an idea of the best way to execute it. We can encode this knowledge in a model-specific recommender. While the previous two recommenders work with any model, they are also not very intelligent or good at finding optimal executions (see also the next section on experiments). By making a model-specific recommender, we can make one that is better, but less generally applicable.

Creating a recommender using our framework is very simple, so we can implement a strategy recommending the academic route directly as in Fig. 15. The implementation has a set of events that should always be recommended with high priority if offered (Preferred). The transition Preferred Activity checks if a preferred event is among the offered ones and if so recommends it. The transition has high priority and will thus be selected before others. If no preferred activity is available, we just return the first (Other Activity). This exploits a known best (at least in some settings) implementation and the fact that all the events are only offered once in this model.

4.2 Log-Based Recommender

We want a recommender that provides better advice than guessing randomly, but at the same time, we would like to avoid models or situation specific

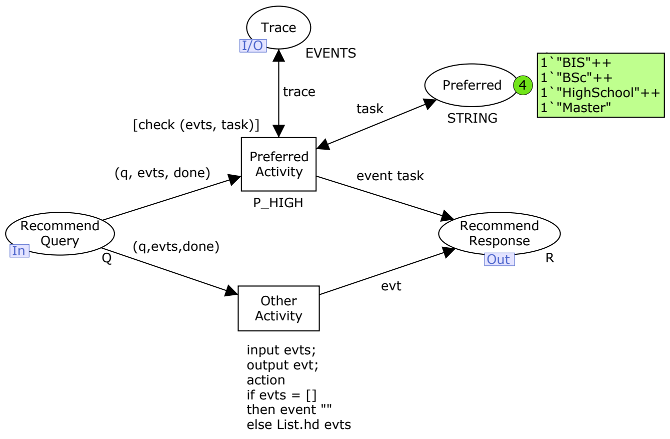


Fig. 15. Model-specific recommender

recommenders. For this purpose the log-based recommender is designed. This provider uses a historical log as guidance for providing recommendations. The idea is that we have a predicate selecting traces from the log we consider to be the same as the current one. We then compute a value on all such traces and return the next event of the trace which yields the best result. In our example, we would consider all traces with the same sequence of completed events similar to the current trace. Our computation would be the complete execution time, and our order would prefer shorter execution times.

The log-based provider is implemented in Java and comprises 513 lines of code, all of which is logic. Furthermore, this provider depends on a complicated library for querying XML documents using XQuery as discussed in [11]. All in all, we do not want to duplicate this code in a CPN model.

5 Experiments

In this section we outline how we have used our testing platform (described in Sect. 3) to test the various providers described in Sect. 4. First of all, we perform tests with the random recommender for all timing models to demonstrate that this yields the same as not using operational support. Also, we use this to eliminate the simplest timing model (constant time for execution) from future tests. Then we evaluate the batch and model-specific recommenders for the remaining timing models. Finally, we evaluate the log-provider using "historical logs" generated by the random provider. We evaluate the providers according to our two goals: shortest execution time and highest success rate, where a trace is successful if the task Master of BPM is executed.

Table 1. Random Provider

	Support	Time Model			
		CONST	PROB	BATCH	WL
Time	0	3998	4029	3941	2206
	50	4024	3971	3905	2207
	90	4014	3994	4015	2220
	100	4034	4071	4043	2205
Success	0	9.0	10.3	10.8	9.2
	50	11.5	11.2	11.1	10.5
	90	11.0	10.8	10.9	11.2
	100	10.7	11.4	11.5	10.2

5.1 Random Provider

Our first test is more a sanity test to test that everything works; if tasks are picked at random with the same probability, it should not matter whether we use support or not. Furthermore, we expect the execution time to be the same for the timing models using constant time as a probability distribution. Finally, we expect the success rate to be nearly the same for all executions, independently of how much support is used and which timing model is used.

In Table 1 we see the results of our first experiments. The table is split in two: the top part shows the average execution time for a trace and the bottom part the success rate as defined above. We show results for each timing model (CONST = execution time for each task is a constant, PROB = execution is independent and identically distributed, BATCH = execution time is dependent on whether you execute the same task more than once, and WL = execution time decreases as stress increases) and for four different values of support. Here, 0% support means that we always chose at random and 100% support means we always ask operational support for advice. All numbers represent 1000 traces.

We see that the support percentage has no effect on either the execution time or the success rate for the tasks with the same time model. We also see that the CONST and PROB time models have the same behavior: the execution time and the success rate are very similar for those. The BATCH time model also has similar though slightly lower execution time whereas the WL time model has a significantly shorter average execution time. The reason for this is that they are allowed to execute tasks faster (if randomly batching or having more tasks in the work list). The qualitative measurement, the success rate does not change (as expected as we only change how long tasks take, not how they are selected).

5.2 Batch Provider

For our second test we want to evaluate the simple batch heuristics. We do not need to perform executions for 0% support (it is the same as the numbers in Table 1). We have also removed the simple time model, CONST as it yields the same results as PROB and is very far from reality. The results are summarized in Table 2.

Table 2. Batch Provider

	Support	Time Model		
		PROB	BATCH	WL
Time	50	4085	3996	2372
	90	4178	4095	2595
	100	4198	4134	2641
Success	50	11.5	11.8	11.5
	90	8.5	13.4	12.5
	100	12.8	15.4	14.4

Table 3. Model-specific Provider

	Support	Time Model		
		PROB	BATCH	WL
Time	50	4047	4001	2376
	90	3866	3766	2797
	100	3793	3711	2946
Success	50	31.0	30.8	32.3
	90	45.8	46.5	46.1
	100	47.3	51.9	51.2

We see generally and sometimes even significantly larger execution times using the batch provider compared to using the random provider (from Table 1). Surprisingly, we also see the execution time increase when the support rate goes up and even for the BATCH time model, which would be expected to benefit from batching. The reason is that while the batch provider recommends batching together similar tasks, it also suggests repeating working as we see in Fig. 2 Work and Master of BPM are the only tasks that can be executed more than once. Thus, this provider does not force progress, and may even prevent it, leading to longer execution times even though individual tasks are executed faster.

5.3 Model-Specific Provider

This is the same test as the one performed for the batch provider, except we now evaluate the provider specially tailored to our model. The results are summarized in Table 3.

We see that for the simple timing models, this provider significantly outperforms the previous, as it successfully picks the shortest path to an education. We also see that when timing is workload-dependent, this provider performs worse. The reason is that while the expected time to get a BSc is 3 years and the sum of the time to take the 6 courses (half a year each) and get a qualifying job (1 year) is 4 years, it is faster to take the 6 courses as the concurrent workload is faster. Thus our smart solution is suboptimal in this case. Regarding the qualitative results, we see that following recommendations leads to higher success rates. The reason the success rate approaches 50 and not 100 is that we do not control when an execution ends, so after executing MSc, BIS (or MSc, ES if we do not listen to operational support) we have a 50% chance of terminating the execution and a 50% chance of continuing.

5.4 Log Provider

Here we need some historical data. As this is a model created for demonstration purposes, we do not have any such data. We can, however, generate an execution log using our model. We simply allow CPN Tools to generate a simulation log and import it into ProM. In Tables 4 and 5 we see the results of using the log

Table 4. Log Provider (Running Time)

	Support	Time Model		
		PROB	BATCH	WL
Time	50	3073	3026	1674
	90	2191	2032	1110
	100	1957	1642	957
Success	50	9.6	5.7	7.0
	90	4.1	10.2	1.3
	100	0.4	13.6	0

Table 5. Log Provider (Success)

	Support	Time Model		
		PROB	BATCH	WL
Time	50	4096	4081	2327
	90	3920	4639	2278
	100	3832	4909	2257
Success	50	24.5	14.8	15.5
	90	45.8	28.7	17.5
	100	48.2	44.5	24.0

provider. We have in all cases used a log generated using random selection, but we make sure to use a log generated using the same time model as the one used to generate the results. We use the log provider to optimize for shortest execution time (Table 4) and for highest chance of obtaining a Master of BPM (Table. 5). This is a good indication of what happens when seeding a recommender with actual historical data from the group of people about to execute the process in a similar situation.

We see that when optimizing for shortest running time we in all cases obtain significantly shorter running time when using support than when we do not. Also, the running time is in all cases much shorter than for all previous providers. The success rate is very low, however. This is because it is possible to make a shorter run by picking MSc, ES as this does not enable Master of BPM. When optimizing for success, we see that the success rate increases and is as good as for the hand-crafted provider. The running time is higher than when we optimize for running time, but the success rate increases and is comparable to the success rate of the hand-crafted provider. The success rate for the workload timing model is surprisingly low, which is because the workload path favors the practical path, which needs to execute more tasks, and hence the historical data may not contain a trace with the same interleaving of the courses.

Table 6. Using foreign providers

Source	Time Model		
	PROB	BATCH	WL
PROB	1957	1772	1011
BATCH	1643	1642	1007
WL	1632	1609	957

In Table 6 we have shown the results of runs using logs generated using a different timing model. This is an indication of how well a recommender seeded with randomly generated data using the correct model but with wrong assumptions about the user behavior. In other words, this is an indication of the stability of the simulation results. We have not shown the value of the success rate as it is the same as the one measured in Table 4 as we can become a Master of BPM using both the academic and practical track.

We see that it does not matter much if the log is generated from data with a BATCH or WL time model. Surprisingly, we get the shortest execution times in all cases when using data generated with a timing model taking the workload into account. In fact, the log generated using a simple probability measure performs worse even when used for the matching time model. This is because there are other logs favoring a trace that is also beneficial for this timing model.

6 Conclusion and Future Work

In this paper, we have presented a CPN model for testing operational support providers. The model is connected to the Declare workflow system and the operational support service in ProM using Access/CPN 2.0, making it possible to test real systems with a model of a user. Our user model is parameterizable, and can exhibit four different kinds of timing behavior. We have also presented a CPN model which can be used to quickly prototype an operational support provider for integration in ProM and subsequent use in both our testing platform and existing clients of the operational support service. We believe that using this approach is also useful in other settings where an algorithm modifies the domain it is doing computations on.

We have used our test suite to test four different recommendation providers. We see that contrary to what is often observed, simple algorithms fail compared to more sophisticated adaptive algorithms. We also see that a hand-crafted algorithm using domain knowledge does not necessarily outperform a smart general algorithm when the domain knowledge builds on wrong assumptions (here about the user behavior). Even simple algorithms designed to exploit certain traits of models (like speedup in batch processing) may fail if the assumption is correct, if some other aspect is ignored (like the need to work towards termination and not just focus on batching tasks together). We also see that our log provider, which uses historical data, is surprisingly stable and handles situations when input data does not completely correctly reflect reality, making seeding such algorithms with generated data possible. We also see that if the algorithm providing recommendations is not optimal, user deviations can be a good idea. Even for the log provider, which proved very efficient, deviations may benefit execution in the long run as users may reach completely new and more efficient ways of executing the process by chance, making it possible to provide better recommendations in the future.

Our experiments show that experimental results may deviate quite a bit from expectations, making testing invaluable. We of course need to validate that simulated results show the same tendencies as real life and future work includes testing recommendations in real life. We also see that the winner by far was the most sophisticated algorithm, making future research into even better algorithms very interesting. One caveat of the current implementation is that it completely fails to provide recommendations if the historical data does not contain a similar trace (which is quite likely after executing 5-8 tasks). This can be alleviated by using a model annotated with timing information for providing recommendations instead of just a flat log.

References

1. van der Aalst, W.M.P.: Process Mining-Discovery, Conformance and Enhancement of Business Processes. Springer (2011)
2. van der Aalst, W.M.P., Pesic, M., Schonenberg, H.: Declarative Workflows: Balancing Between Flexibility and Support. Computer Science - Research and Development 23(2), 99–113 (2009)
3. van der Aalst, W.M.P., Pesic, M., Song, M.S.: Beyond Process Mining: From the Past to Present and Future. In: Pernici, B. (ed.) CAiSE 2010. LNCS, vol. 6051, pp. 38–52. Springer, Heidelberg (2010)
4. van der Aalst, W.M.P., Schonenberg, H., Song, M.S.: Time Prediction based on Process Mining. Information Systems 36(2), 450–475 (2011)
5. van der Aalst, W.M.P., Stahl, C., Westergaard, M.: Strategies for Modeling Complex Processes Using Colored Petri Nets. In: Jensen, K., Donatelli, S., Kleijn, J. (eds.) ToPNoC V. LNCS, vol. 6900, pp. 265–291. Springer, Heidelberg (2012)
6. CPN Tools (2012), `cpntools.org`
7. Declare (2012), `http://www.win.tue.nl/declare`
8. Jensen, K., Kristensen, L.M.: Coloured Petri Nets: Modeling and Validation of Concurrent Systems. Springer (2009)
9. Mans, R.S., van der Aalst, W.M.P., Russell, N.C., Bakker, P.J.M., Moleman, A.J.: Process-Aware Information System Development for the Healthcare Domain - Consistency, Reliability, and Effectiveness. In: Rinderle-Ma, S., Sadiq, S., Leymann, F. (eds.) BPM 2009. LNBIP, vol. 43, pp. 635–646. Springer, Heidelberg (2010)
10. Nakatumba, J., van der Aalst, W.M.P.: Analyzing Resource Behavior Using Process Mining. In: Rinderle-Ma, S., Sadiq, S., Leymann, F. (eds.) BPM 2009. LNBIP, vol. 43, pp. 69–80. Springer, Heidelberg (2010)
11. Nakatumba, J., Westergaard, M., van der Aalst, W.M.P.: A Meta-model for Operational Support. BPM Center Report BPM-12-05, BPMcenter.org (2012)
12. Nakatumba, J., Westergaard, M., van der Aalst, W.M.P.: Generating Event Logs with Workload-Dependent Speeds from Simulation Models. In: Enterprise and Organizational Modeling and Simulation. LNBIP. Springer (2012)
13. ProM (2012), `http://www.processmining.org`
14. Resnick, P., Varian, H.R.: Recommender Systems. Comm. of the ACM 40(3), 56–58 (1997)
15. Rozinat, A., Wynn, M.T., 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)
16. Schonenberg, H., Weber, B., van Dongen, B.F., van der Aalst, W.M.P.: Supporting Flexible Processes through Recommendations Based on History. In: Dumas, M., Reichert, M., Shan, M.-C. (eds.) BPM 2008. LNCS, vol. 5240, pp. 51–66. Springer, Heidelberg (2008)
17. Weber, B., Wild, W., Breu, R.: CBRFlow: Enabling Adaptive Workflow Management Through Conversational Case-Based Reasoning. In: Funk, P., González Calero, P.A. (eds.) ECCBR 2004. LNCS (LNAI), vol. 3155, pp. 434–448. Springer, Heidelberg (2004)
18. Westergaard, M.: Access/CPN 2.0: A High-level Interface to Coloured Petri Net Models. In: Kristensen, L.M., Petrucci, L. (eds.) PETRI NETS 2011. LNCS, vol. 6709, pp. 328–337. Springer, Heidelberg (2011)
19. Westergaard, M., Maggi, F.M.: Modeling and Verification of a Protocol for Operational Support Using Coloured Petri Nets. In: Kristensen, L.M., Petrucci, L. (eds.) PETRI NETS 2011. LNCS, vol. 6709, pp. 169–188. Springer, Heidelberg (2011)
20. Wickens, C.D.: Engineering Psychology and Human Performance. Harper (1992)

Designing Weakly Terminating ROS Systems

Debjyoti Bera, Kees M. van Hee, and Jan Martijn van der Werf*

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

Abstract. The Robot Operating System (ROS) is a popular software framework to develop and execute software for robot systems. ROS supports component-based development and provides a communication layer for easy integration. It supports three interaction patterns that are essential for control systems: the publish-subscribe pattern, the remote procedure call pattern and the goal-feedback-result pattern. In this paper we apply Petri nets to develop a structured design method for ROS systems, such that the weak termination property is guaranteed. The method is based on stepwise refinement using three interaction patterns and components modeled as state machines. The method is illustrated with a case study of robot ROSE.

Keywords: Petri nets, Correctness by Construction, Components, Patterns, Weak Termination, Robot Operating System, Architectural Framework.

1 Introduction

Robots assist human beings by taking over some of their tasks. In contrast with industry robots, service robots operate in environments where people live, like a house, an hospital or an office. So, they have to react very fast in order to be safe and effective. The software subsystem of a service robot is a complex distributed system. The complexity arises from the need to interface with heterogeneous hardware while executing concurrently a large variety of computations such as image processing, sensor data processing, planning and control tasks. Component-based frameworks are attractive for programming robot systems because they augment the advantages of the traditional programming frameworks by facilitating the loose coupling between components (asynchronous communication), promoting reuse and dynamic reconfiguration. Over the years a number of component based programming frameworks like OROCOS [3], openRTM [2] and Player/Stage [5] have been developed. A few of them have enjoyed great

* 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).

success as demonstrated by the popular Robot Operating System (ROS) [12]. ROS is a widely used component based development framework for robot systems that runs on top of the Linux operating system. To a large extent, ROS has been able to address the shortcomings of its predecessors. For this reason, it saw an unprecedented growth in popularity over the past few years as a common platform to share knowledge.

ROS provides a communication model (middleware) as a library of the three most recurring interaction patterns in robot systems, namely the *remote procedure call* (RPC) pattern, the *publish-subscribe* (PS) pattern and the *goal-feedback-result* (GFR) pattern. The first two also occur in other types of distributed systems, whereas the third pattern is typical for control systems in robots. ROS lacks an integrated formal technique to verify or validate correctness properties of systems, like deadlock freedom, in a structured way.

In this paper we apply Petri nets to develop a structured design method for ROS systems, such that the weak termination property is guaranteed. Weak termination is a combination of freedom of deadlock and freedom of livelock. It means that a system always must be able to reach a final state from any reachable state. We present a construction method, based on the stepwise refinement principle, that guarantees weak termination at the control flow level. It is not necessary to fully accord the rules of the design approach, as long as the final system can be derived according to the construction rules, the system is ensured to be correct. In fact, our approach is a structured programming technique for distributed programs, comparable to structural programming methods for sequential programs of the late sixties [4].

This paper is structured as follows. In Sect. 2 we sketch the context of the problem: the development of the service robot ROSE. In Sect. 3, we present our modeling problem: the features of ROS to be modeled, which are modeled using Petri nets in Sect. 4. Petri nets offer a convenient way of expressing and analyzing the behavior of asynchronous distributed systems. A *component* is modeled as a strongly connected state machine with one special place called 'idle'. The three interaction patterns are modeled as a special kind of *multi-workflows* ([9]). These multi-workflows have so-called client and server sides, where the clients are open workflows and the servers are open components. In this section, we also show that these patterns are weakly terminating. Sect. 5 presents the construction method. We prove that the method guarantees weak termination of the whole system. Here we use an *architectural diagram* as starting point for the design, which is a graph where the nodes are components and the arcs represent interaction patterns. The refinement method has two phases. In the first phase we refine simultaneously a set of places by a RPC or a GFR pattern. The servers of these patterns form the base for new components. If all components of the system are created, we enter the second phase in which we add the PS patterns. In Sect. 6 we apply the method for a the navigation subsystem of robot ROSE. Sect. 7 concludes the paper, in which we discuss the lessons learnt.

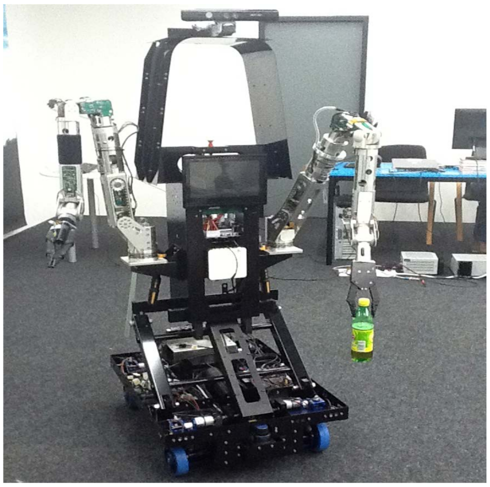


Fig. 1. Robot ROSE

2 Robot ROSE

Service robotics is a rather new branch of robotics. They differ fundamentally from the well-developed industry robots and the experimental humanoids. Industry robots are programmed for a limited set of specified manufacturing tasks. They are position-controlled, i.e. they are able to move to prescribed coordinates only. They operate in dedicated environments. Humanoids are robots that are look-a-likes of human beings that should be able to imitate human behavior. Service robots on the contrary are not supposed to be look-a-likes, they should rather be able to assist humans by performing human tasks. These robots have to operate in unstructured environments where people live. As a consequence, they need to deal with unpredictable situations. Therefore, instead of being position-controlled, service robots use visual servo-ing (vision based control) techniques. These techniques give less accuracy. For these reasons, full autonomy in such environments is very hard to achieve. We focus on service robots that share their autonomy with a human at a distance. One of the tasks they should be able to perform is to autonomously navigate to a user-defined place in a building, while avoiding collisions with other obstacles. We call such robots *tele-operated* service robots.

An example of a tele-operated service robot is ROSE, which is depicted in Fig. 1. ROSE is a care robot for home care of elderly and disabled people. The care robot is envisioned to replace the caretaker at the local site by allowing remote operations. Caretakers operate from a care center and they are able to help elderly people with simple household activities, like warming a meal or taking out the garbage, by remotely commanding robot ROSE at the local site. ROSE is semi-autonomous and realizes shared control with the human in the loop. So a caretaker is able to service several robots for the elderly or disabled people. The robot is equipped with four independently steerable wheels, two 7-DOF (degrees of freedom) arms, a vision system and a wide array of sensors. The use-cases for the robot ROSE require the most common collision free functionalities, such as pick, place, manipulate and map based navigation. The system is tested in a field lab (see [13]).

3 The Robot Operating System

The internal software system of ROSE integrates a wide variety of self-made and off-the-shelf software components and comprises of a deep stack consisting of hardware drivers at the lowest level, continuing up through perception, abstract reasoning and beyond. To manage the complexity, we need a development framework that supports large-scale software integration.

One such development framework is the Robot Operating System (ROS). ROS is a component-based development framework focusing in particular on planning and control systems. It hides the heterogeneity of systems by providing a thin-structured communications layer comprising of the most recurring interaction patterns. The general purpose nature of ROS led to the rapid rise in its popularity as a common platform to share knowledge in the robotics community. Therefore, it is a logical choice for the development of our service robot.

The availability of many open source components for robots emphasizes the need for a structured approach for integration that guarantees certain behavioral properties like weak termination, where we focus on in this paper.

ROS provides a communication model and infrastructure capabilities for component instantiation, dynamic configuration and deployment. It comes with a number of integrated frameworks (like OROCOS [3], Player/Stage [5]), libraries (like OpenCV [14] and KDL [14]), general purpose configurable software modules (like platform and arm navigation, generic hardware drivers etc.) and tools for visualization and simulation (like RViz and Gazebo [5]). ROS is available to the developer as a set of libraries.

In ROS, blocks of functional code can be instantiated as components, in ROS terminology referred to as *nodes*. A component performs iterative computations and is allowed to communicate with other components via an interaction pattern provided by the *communication model* of the framework. An *interaction pattern* is a composition of clients and servers that share a set of buffers over which messages are exchanged. A *buffer* has an associated data type specifying the data type of the messages that can be stored. Messages have strictly typed data structures that can be either simple or compound, the latter can be derived by arbitrary nesting of simple data structures.

At run-time, a ROS system can be configured dynamically, i.e. one or more components can be added or removed from the system at run-time. Dynamic reconfiguration is made possible by the naming and service registration server called *ROS master*. A ROS system has one ROS master. The ROS master acts as a central registry of information about deployed components and their interaction patterns. This allows components to discover each other. In this way location transparency is achieved. Both the arrival as well as the departure of components are registered by the ROS master. Once a component is aware of the available components and the services they provide, connection negotiation and an exchange of messages are carried out on a peer to peer basis.

The *communication model* of ROS is a collection of three very frequently recurring asynchronous interaction patterns in planning and control systems: the publish-subscribe (PS) pattern, the remote procedure call (RPC) pattern and the goal-feedback-result (GFR) pattern. Each interaction pattern is made of a set of clients and a set of servers. Each client and server have a *middleware* module that is responsible for (a) interactions with the ROS master, (b) client-server connection negotiation and management, (c) message transportation (over managed connection), and (d) message buffering capabilities. In the remainder of this section we elaborate on each of these interaction patterns.

Publish-Subscribe Pattern. The PS pattern is an unidirectional notification pattern supporting many to many message broadcasting, i.e., multiple clients are allowed to broadcast messages to multiple servers. Messages are published by the client on some topic. A *topic* describes a message stream by specifying its message type and name. The pattern is well suited for the broadcast of sensor data streams. The ROS Master ensures that during run-time the clients know which servers are subscribed to their topic. When a server is coupled or decoupled, the ROS Master informs all the clients about this change.

Remote Procedure Call Pattern. The RPC pattern comprises of a single server and multiple clients. The server models a set of procedures that can be executed one at a time. Each procedure can have one or more associated clients. A procedure waits for a request message from the client, and terminates after sending back a result message. The client is blocked until the server returns a result message. No other messages are exchanged between the server and the client. When a RPC client is instantiated, it requests the ROS master to find out which server offers the requested procedure. Once located, the client and server negotiate the connection, and messages are exchanged over this connection on a peer-to-peer basis.

Goal-Feedback-Result Pattern. The GFR pattern is a more elaborate interaction pattern. The GFR pattern comprises a single server handling many clients. The GFR pattern is well suited for long running procedures and gives the client the option to preempt the server. The arrival of a goal message at the GFR server triggers a process (generally a control algorithm) whose progress on that goal may be reported via *feedback messages* to the waiting client. At any moment in time, an active client waiting for a message from the server is allowed to cancel the process at the server by sending a *cancel message*. The process at the server always terminates after sending a *result message* to the client indicating the result (i.e., finished, aborted, or canceled).

When a GFR client is instantiated, the ROS master enables the client to locate the GFR server. As in case of the RPC pattern, the GFR client and server negotiate the connection over which messages are exchanged. Furthermore, the server enforces the *single pending goal* policy. This means that at most one pending goal is allowed. So new goals preempt the current pending goal and the client is notified.

4 The Logical Model

For each of the three interaction patterns, we will construct a Petri net and show that these patterns are weakly terminating. The logical model of each interaction pattern abstracts away implementation details of the *middleware* module, like interactions with the ROS master, connection negotiation, connection management, message transportation and buffer management. First, we introduce basic notions needed for the framework.

4.1 Basic Notions

The architectural framework we introduce in this paper is built upon Petri nets with inhibitor arcs and reset arcs, called inhibitor/reset nets. An inhibitor/reset net is a 5-tuple (P, T, F, ι, ρ) where P and T are two disjoint sets of *places* and *transitions* respectively, $F \subseteq (P \times T) \cup (T \times P)$ is the set of *arcs*, and $\iota, \rho : T \to \mathcal{P}(P)$ specify the inhibitor and reset arcs, respectively, with $\mathcal{P}(P)$ denoting the powerset of P. Elements of the set $P \cup T$ are called *nodes*. In the remainder we refer to an inhibitor/reset net as a net. We refer to the individual elements in a tuple by adding its name in subscript, e.g. we write P_N for the set of places of a net N.

Nets can be depicted graphically. Places and transitions are represented as circles and squares, respectively, an arc (n, m) is depicted as a directed arc from node n to node m. A dot-headed arc is drawn from place p to transition t if $p \in \iota(t)$, if $p \in \rho(t)$, a dashed arc is drawn between p and t.

We define the preset of a node n as ${}^{\bullet}_{N}n = \{m | (m, n) \in F\}$ and the postset as $m^{\bullet}_{N} = \{n | (m, n) \in F\}$. As a shorthand, we use the preset (postset) as a function such that ${}^{\bullet}_{N}n(m) = 1$ ($n^{\bullet}_{N}(m) = 1$) if $m \in {}^{\bullet}_{N}n$ ($m \in n^{\bullet}_{N}$) and ${}^{\bullet}_{N}n(m) = 0$ ($n^{\bullet}_{N}(m) = 0$) otherwise. If the context is clear, we omit the subscript. A Petri net is *strongly connected* if for any two nodes there exists a directed path from one to the other.

A net models behavior. The *state* or *marking* of a net $N = (P, T, F, \iota, \rho)$ is a function $m : P \to \mathbb{N}$ where $\mathbb{N} = \{0, 1, 2, \ldots\}$. The pair (N, m) is called a *marked net.* A place $p \in P$ is called *marked* if $m(p) > 0$. A transition $t \in T$ is *enabled* in some marking, denoted by $(N, m)[t\rangle$, if ${}^{\bullet}t(p) \leq m(p)$ for all places $p \in P$ and $m(p) = 0$ for all places $p \in \iota(t)$. An enabled transition may *fire*, resulting in a new marking m' with $m'(p) = m(p) - {}^{\bullet}t(p) + t^{\bullet}(p)$ if $p \in P \setminus \rho(t)$ and $m'(p) = 0$ if $p \in \rho(t)$, and is denoted by $(N, m)[t\rangle(N, m')$. A marking m_n is reachable from a marked net (N, m_0) if markings $m_1, \ldots, m_{n-1}$ and transitions $t_1, \ldots, t_n$

exist such that $(N, m_{i-1})[t_i\rangle(N, m_i)$ for all $1 \leq i \leq n$. A system is a 3-tuple (N, M_0, M_f) where N is a net, M_0 is a set of initial markings, and M_f is a set of desired final markings. A system (N, M_i, M_f) is called *weakly terminating* if for every initial marking $m_i \in M_i$ and every reachable marking m of (N, m_i) a final marking $m_f \in M_f$ is reachable.

We identify two subclasses of nets. A net $N = (P, T, F, \iota, \rho)$ is an *S-net* if $|{}^\bullet t| \leq 1$ and $|t^\bullet| \leq 1$ for all transitions $t \in T$. It is a *workflow net* if a unique *initial place* $i \in P$ and a unique *final place* $f \in P$ exist such that ${}^\bullet i = f^\bullet = \emptyset$ and every node is on a path from i to f. A workflow net that is also an S-net is called an *S-WFN*.

Given two nets $N_1 = (P_1, T_1, F_1, \iota_1, \rho_1)$ and $N_2 = (P_2, T_2, F_2, \iota_2, \rho_2)$, their *union* is the net $(P_1 \cup P_2, T_1 \cup T_2, F_1 \cup F_2, \iota, \rho)$ where $\iota(t) = \iota_1(t) \cup \iota_2(t)$ and $\rho(t) = \rho_1(t) \cup \rho_2(t)$ for all $t \in T_1 \cup T_2$.

4.2 Components and Their Interaction

Components in ROS communicate asynchronously by message exchange. For this, we introduce the notion of *open nets*. An *open net* (OPN) is a tuple $(P, I, O, T, F, \iota, \rho)$ such that $(P \cup I \cup O, T, F, \iota, \rho)$ is a net, P,I and O are pair-wise disjoint, I is a set of *input places* and O is a set of *output places*, i.e., ${}^\bullet i = o^\bullet = \emptyset$ for all $i \in I$ and $o \in O$. We call the net $\mathcal{S}(N) = (P, T, F', \iota, \rho)$ the *skeleton* of N with $F' = F \cap ((P \times T) \cup (T \times P))$. An open net is called an open S-net if its skeleton is an S-net. Likewise, an open net is an open workflow net (OWN) if its skeleton is a workflow net. If the skeleton is also an S-Net, it is called an S-OWN. Two OPNs N and M are *composable* if $(P_N \cup I_N \cup O_N \cup T_N) \cap (P_M \cup I_M \cup O_M \cup T_M) \subseteq ((I_N \cap O_M) \cup (O_N \cap I_M))$. The composition of two OPNs is defined as their union.

Definition 1 (Component). *A* component *is an OPN C whose skeleton is a strongly connected S-net with a special place called* idle*, denoted by v_C. The initial marking of a component has one token in the idle place.*

In component-based systems, components cooperate in order to achieve some goal. Such a cooperation is modeled as a multi workflow net [9]. A multi workflow net (MWF net) is a generalization of classical workflow nets, having multiple initial/final pairs, whereas a single workflow net has only a single initial/final pair.

Definition 2 (Multi-workflow net). *A* multi-workflow net *(MWF net) N is a tuple $(P, T, F, \iota, \rho, E, \triangle)$ where (P, T, F, ι, ρ) is a net, $E \subseteq P \times P$ is the set of initial/final pairs such that $|E| = |\pi_1(E)| = |\pi_2(E)|$, ${}^\bullet\pi_1(E) = \pi_2(E)^\bullet = \emptyset$, $\iota(T), \rho(T) \subseteq S$ and $\triangle \subseteq S$ is the set of* idle places *with $S = P \setminus (\pi_1(E) \cup \pi_2(E))$, where π_1 and π_2 are two projection functions defined on the cartesian product of two sets.*

Simultaneous refinement [9] is a refinement operation that simultaneously refines a set of places of a system with an MWF net. This operation is a natural extension of the refinement of a single place [10].

Definition 3 (Simultaneous refinement). *Let N be a net and $R \subseteq P_N$ be a set of places to be refined such that no $r \in R$ exists with $r \in \iota(t)$ or $r \in \rho(t)$ for all $t \in T_N$. Let M be a MWF net such that N and M are disjoint. Let $\alpha : R \to E_M$ be a total bijection. The refinement of N with M, denoted by $N \odot_\alpha M$, is a net $\bar{N} = N \odot_\alpha M$, where $\bar{N}$ is the net defined as $P_{\bar{N}} = (P_N \setminus R) \cup P_M$, $T_{\bar{N}} = T_N \cup T_M$, $F_{\bar{N}} = (F_N \setminus \bigcup_{q \in R}(({}_N^{\bullet}q \times \{q\}) \cup (\{q\} \times q_N^{\bullet}))) \cup F_M \cup \bigcup_{q \in R}(({}_N^{\bullet}q \times \{\pi_1(\alpha(q))\}) \cup (\{\pi_2(\alpha(q))\} \times q_N^{\bullet}))$, and for all $t \in T_{\bar{N}}$ we have $\iota_{\bar{N}}(t) = \iota_N(t) \cup \iota_M(t)$ and $\rho_{\bar{N}}(t) = \rho_N(t) \cup \rho_M(t)$.*

Components communicate asynchronously. An *interaction pattern* describes the communication between different components. It is a parameterized MWF net that can be inserted into a system. An interaction pattern has two sets, a set of *clients* and a set of *servers*, such that the union of the nets is an MWF net. A client is an OWN, whereas a server is a component.

Definition 4 (Interaction pattern). *An* interaction pattern *is a pair $(\mathcal{C}, \mathcal{S})$ where $\mathcal{C}$ is a set of OWNs, called the* clients, *and $\mathcal{S}$ is a set of components called the* servers, *such that $I_C = O_S$ and $O_C = I_S$, and the union $N = C \cup S$ is a MWF-net, with $E_N = \{(i_X, f_X) \mid X \in \mathcal{C}, i_X, f_X \in P_X, {}^{\bullet}i_X = f_X{}^{\bullet} = \emptyset\}$ and $\Delta_N = \{v_X \mid X \in \mathcal{S}\}$, where $C = \bigcup_{X \in \mathcal{C}} X$ and $S = \bigcup_{X \in \mathcal{S}} X$. We refer to the MWF-net N as $\mathcal{N}(\mathcal{C}, \mathcal{S})$.*

Given a set of components, we can insert an interaction pattern. For a client we select a place within a component that will be refined by the client OWN, a server is inserted into a component by fusing the idle places of the original component and of the server component. Further, we disallow that a server and a client are inserted in the same component.

Definition 5 (Insertion of an interaction pattern). *Let $(\mathcal{C}, \mathcal{S})$ be an interaction pattern, and let $\mathcal{O}$ be a set of pairwise composable components such that $\mathcal{S}$ and $\mathcal{O}$, and $\mathcal{C}$ and $\mathcal{O}$ are pairwise disjoint. Let $\alpha : \mathcal{C} \to \bigcup_{X \in \mathcal{O}} P_X$ be an injective function defining which place is refined by which client, and $\beta : \mathcal{S} \to \mathcal{O}$ be an injective function defining which server is added to which component, such that no client and server are inserted in the same component, i.e., $\alpha(C) \notin P_{\beta(S)}$ for all $C \in \mathcal{C}$ and $S \in \mathcal{S}$.*

The insertion *of $(\mathcal{C}, \mathcal{S})$ in $\mathcal{O}$, denoted by $\mathcal{O} \uplus_{\alpha,\beta} (\mathcal{C}, \mathcal{S})$, is defined by:*

$$\begin{aligned}\mathcal{O} \uplus_{\alpha,\beta}\ (\mathcal{C}, \mathcal{S}) = {} & (\mathcal{O} \setminus \{X \in \mathcal{O} \mid (\exists S \in \mathcal{S} : \beta(S) = X) \vee (\exists C \in \mathcal{C} : \alpha(C) \in P_X)\}) \\ & \cup\ \{X \odot_{\{\alpha(C) \mapsto (i_C, f_C)\}} C \mid X \in \mathcal{O}, C \in \mathcal{C}, \alpha(C) \in P_X\} \\ & \cup\ \{X_{[v_X \mapsto v]} \cup S_{[v_S \mapsto v]} \mid X \in \mathcal{O}, S \in \mathcal{S}, \beta(S) = X\}\end{aligned}$$

where $N_{[p \mapsto r]}$ denotes the renaming of place p into a new place $r \notin P_N$.

Next, we will formalize each of the three interaction patterns as provided by the ROS communication model, and show that each pattern weakly terminates.

4.3 The GFR Pattern

The GFR pattern allows multiple clients to communicate with a single server. A client triggers the server by sending a goal message. It accepts this goal, and it

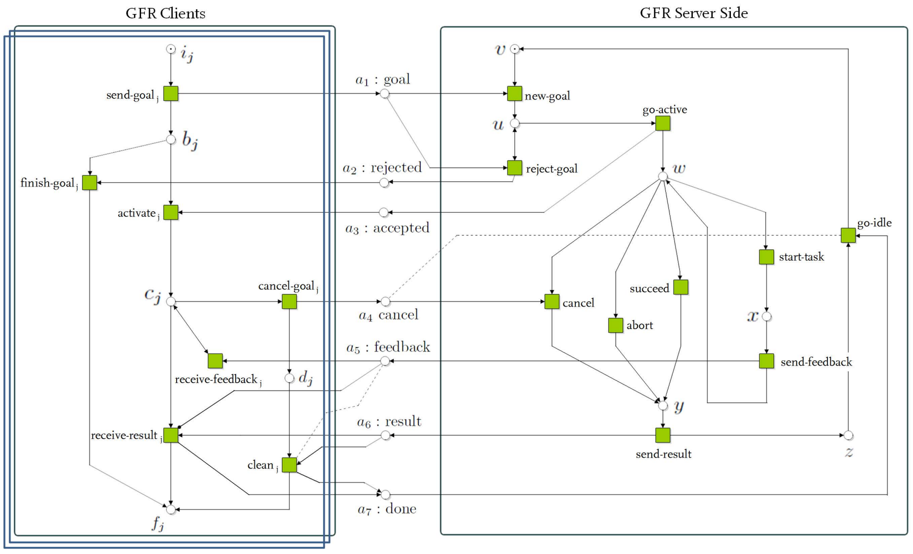


Fig. 2. GFR Pattern with n clients

becomes the pending goal of the server. In case the server already has a pending goal, this goal is canceled, and a reject message is sent to the corresponding client.

On rejection, the corresponding client terminates. When the server accepts a pending goal, the server starts processing it. Depending on the goal, the server may send feedback messages. Finally, the server will send its result and returns to the initial state to process a new pending goal. When the client has received the accept message for a goal, it needs to process the feedback sent by the server before it may accept the result message of the server. The client is also allowed to cancel the goal at any moment after it has been activated. In case a client sends a cancel message to the server, the server will report that it has canceled by sending a result message. Fig. 2 depicts the pattern.

The server of a GFR pattern executes a procedure in a loop. This procedure is modeled by a place in the server labeled x. A procedure can be further detailed by first modeling its control flow as an S-WFN and then performing a refinement of place x.

Definition 6 (GFR pattern). *We will use the notations depicted in Fig. 2 to define the* GFR pattern *consisting of a single server M and clients $\mathcal{C} = \{C_1, \ldots, C_n\}$. The* GFR pattern *is the pair $(\mathcal{C}, \{M\})$ with an* idle *place v.*

We define the system $GFR(\mathcal{C}, \{M\}) = (N, M_0, M_f)$ with $N = \mathcal{N}(\mathcal{C}, \{M\})$, M_0 is defined by $m \in M_0$ iff $m(v) = 1$, $m(i_j) \leq 1$ for each client C_j with $1 \leq j \leq n$ and all other places are empty, the set of all final markings M_f is defined by $m \in M_f$ iff $m(v) = 1$, $m(i_j) + m(f_j) = m_0(i_j)$ for each client C_j with $1 \leq j \leq n$ and initial marking $m_0 \in M_0$, and all other places are empty.

Based on the properties of the net, we show that for any number of clients, the GFR pattern is weakly terminating.

Theorem 7 (Weak termination of the GFR pattern). *Let $(\mathcal{C}, \{M\})$ be the GFR pattern with clients $\mathcal{C} = \{C_1, \ldots, C_n\}$ as defined in Def. 6. Then, the system $GFR(\mathcal{C}, \{M\})$ is weakly terminating.*

Proof. (sketch) Let $GFR(\mathcal{C}, \{M\}) = (N, M_0, M_f)$ and let $m_0 \in M_0$. From the structure of the GFR pattern it is easy to verify that the following place invariants hold, i.e., for any marking m reachable from (N, m_0), we have

1. The single goal policy: $m(v) + m(u) \leq 1$;
2. Only one goal is active: $m(u) + m(v) + m(w) + m(x) + m(y) + m(z) = 1$;
3. A single result is returned: $m(w) + m(x) + m(y) + m(a_6) + m(a_7) \leq 1$;
4. The skeleton of the clients is safe: $\forall 1 \leq j \leq n : m(i_j) + m(b_j) + m(c_j) + m(d_j) + m(f_j) = m_0(i_j)$;
5. Each goal is accepted or rejected: $\sum_{j=1}^{n} m(b_j) = m(a_1) + m(a_2) + m(a_3) + m(u)$;
6. A single instance is handled by the server, and the server returns to the idle state: $m(v) + m(u) + m(a_3) + m(a_7) + \sum_{j=1}^{n}(m(c_j) + m(d_j)) = 1$;
7. For each goal there exists at most one cancel message: $m(a_4) \leq m(a_7) + \sum_{j=1}^{n} m(d_j) \leq 1$

Based on these invariants we conclude:

– Only one client can be active at the same time.
– All places except a_1, a_2 and a_5 are safe.
– If place a_3 is marked then $j \in \{1...n\}$ exists such that place b_j is not empty.

Let C_j be this active client. Then either place c_j or place d_j is marked with a single token. Places a_4 and a_5 do not influence the desired behavior of N, as all tokens in places a_4 and a_5 are respectively removed by transition *go-idle*, and by transitions *receive-feedback*$_j$ or *clean*$_j$.

To analyze the behavior note that subnet $N_1 = \{P_1, T_1, F_1\}$ defined by $P_1 = \{u, v, w, x, y, z, a_3, a_7\} \cup \bigcup_{j \in \{1,\ldots,n\}} \{c_j, d_j\}$, $T_1 = {}^\bullet P_1 \cup P_1{}^\bullet$ and $F_1 = F \cap ((P_1 \times T_1) \cup (T_1 \times P_1))$ is initially marked with a single token in place v. Subnet N_1 corresponds to the goal handling of server M. From the invariants it follows that net N_1 is a strongly connected state machine. Thus, it is always possible to reach the idle place v. Note that place a_4 only influences the order in which transitions *send-result* on the one hand and *receive-result* or *clean* on the other hand fire.

Next, observe that the server autonomously may fire transition *send-result* and that client C_j can either fire transition *clean*$_j$ or transition *receive-result*$_j$. After either one of these two transitions, transition *go-idle* fires, returning the subnet to its initial marking. Remark that in each non-empty cycle from and to v in N_1, transition *go-active* fires only once. Since only either one of the transitions *clean*$_j$ and *receive-result*$_j$ fires once, only a single token is added to place f_j in this cycle.

As each initially marked client C_j always receives either a *reject* or *accept* message after firing its transition *send-goal*$_j$, and by invariant 4, each client ends with a single token either in place f_j or i_j if and only if initially place i_j was marked with a single token. □

In any reachable marking for a GFR pattern, a client is either waiting to be executed, under execution, or already finished. The proof of the theorem above shows that clients that are not under execution can be added or removed from the pattern.

Corollary 8 (Dynamic reconfiguration of the GFR pattern). *Given a GFR pattern $(\mathcal{C}, \{M\})$ as defined in Def. 6 with $\mathcal{C} = \{C_1, \dots C_n\}$, let m be a reachable marking of GFR$(\mathcal{C}, \{M\})$. Let $X, Y, Z \subseteq \mathcal{C}$ be a partitioning of $\mathcal{C}$ such that $C_j \in X$ iff $m(i_j) = 1$, $C_j \in Y$ iff either $m(f_j) = 1$ or the net C_j is unmarked, and $C_j \in Z$ otherwise. Then GFR$(\mathcal{C}', \{M\})$ is weakly terminating for any set of clients $\mathcal{C}'$ with $Z \subseteq \mathcal{C}' \subseteq \mathcal{C}$.*

4.4 The RPC Pattern

The RPC pattern consists of a single server and multiple clients. The *server* of an RPC pattern comprises of a set of *procedures* and *internal behavior*. Each procedure is started on the request of a client, and will in due time send a response. A server can only execute one procedure at a time. Each client calls a single procedure and multiple clients may invoke the same procedure. Fig. 3 presents the *logical model* of an RPC pattern.

The *busy* places labeled e and c_i, $i \in \{1...k\}$ belonging to the server of a RPC pattern, can be refined. The place e models the internal behavior of the server and each place c_i models the internal behavior of procedures. Similar to the GFR pattern, busy places can be refined by an S-WFN.

Definition 9 (RPC pattern). *We use the notations of Fig. 3 to define the* RPC pattern $(\mathcal{C}, \{S\})$*, where $S = W \cup \bigcup_{Q \in R} Q$ denotes the server and $\mathcal{C} = \{C_1, \dots, C_n\}$ the set of clients, with $R = \{R_1, \dots, R_k\}$ a non-empty set of k* procedures *and $r : \mathcal{C} \to \{1 \dots k\}$ is a total surjective function denoting the procedure each client invokes. The* RPC pattern *is the pair $(\mathcal{C}, \{S\})$ with an* idle *place v.*

For the RPC pattern, we define the system RPC$(\mathcal{C}, \mathcal{S}) = (N, M_0, M_f)$ by $N = \mathcal{N}(\mathcal{C}, \{S\})$, M_0 is defined by $m \in M_0$ iff $m(v) = 1$, $m(i_j) \leq 1$ for each client C_j with $1 \leq j \leq n$ and all other places are empty, and the set of all final markings M_f is defined by $m \in M_f$ iff $m(v) = 1$, $m(i_j) + m(f_j) = m_0(i_j)$ for each client C_j with $1 \leq j \leq n$ and initial marking $m_0 \in M_0$, and all other places are empty.

Next, we show that the RPC pattern is weakly terminating.

Theorem 10 (Weak termination of the RPC pattern). *Let $(\mathcal{C}, \{S\})$ be an RPC pattern as defined in Def. 9. Then RPC$(\mathcal{C}, \{S\})$ is weakly terminating.*

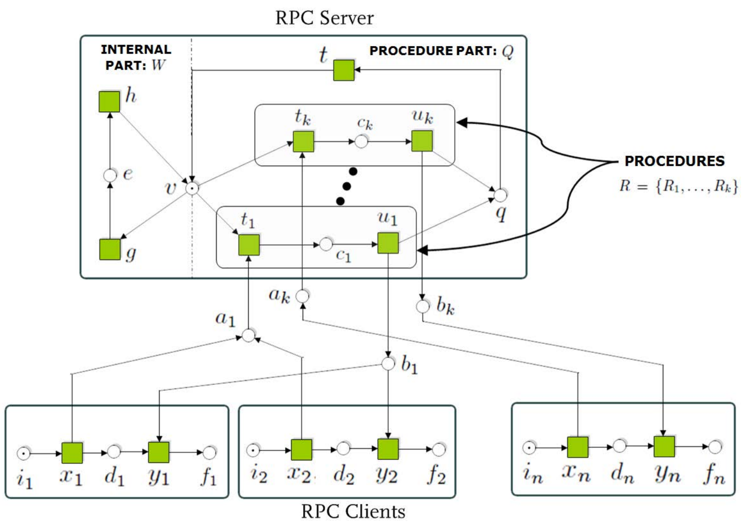


Fig. 3. RPC Pattern with n clients

Proof. (sketch) Let $\mathcal{C} = \{C_1, \ldots, C_n\}$, i.e., the RPC pattern consists of n clients, let $R = \{R_1, \ldots, R_k\}$ be the k procedures handled by server S, and $r : C \to \{1, \ldots, k\}$ be the total surjective function defining for each client which procedure is taken. Let $RPC(\mathcal{C}, \mathcal{S}) = (N, M_0, M_f)$ and let $m_0 \in M_0$.

From the structure of the RPC pattern it is easy to verify the following place invariants, i.e., for any marking m reachable from (N, m_0), we have:

1. The skeleton of the server is safe: $m(v) + m(q) + m(e) + \sum_{j=1}^{k} m(c_j) = 1$
2. Each procedure R_i, $i \in \{1, ..., k\}$ is called at most $|r^{-1}(i)|$ times, i.e., $|r^{-1}(i)|$ clients call the procedure R_i: $\sum_{C_j \in r^{-1}(i)} m(d_j) = m(a_i) + m(b_i) + m(c_i)$

Based on these invariants, and the observation that server S is a component with idle place v, it follows that a client that requests a procedure is always able to receive a response. Hence, the system weakly terminates. □

A logical consequence of the definition of the RPC pattern is that clients can be attached and detached at runtime, as shown in the next corollary.

Corollary 11 (Dynamic reconfiguration of the RPC pattern). *Given a RPC pattern $(\mathcal{C}, \{S\})$ as defined in Def. 9 with $\mathcal{C} = \{C_1, \ldots C_n\}$, let m be a reachable marking of $RPC(\mathcal{C}, \{S\})$. Let $X, Y, Z \subseteq \mathcal{C}$ be a partitioning of $\mathcal{C}$ such that $C_j \in X$ iff $m(i_j) = 1$, $C_j \in Y$ iff either $m(f_j) = 1$ or the net C_j is unmarked, and $C_j \in Z$ otherwise. Then $RPC(\mathcal{C}', \{M\})$ is weakly terminating for any set of clients $\mathcal{C}'$ with $Z \subseteq \mathcal{C}' \subseteq \mathcal{C}$.*

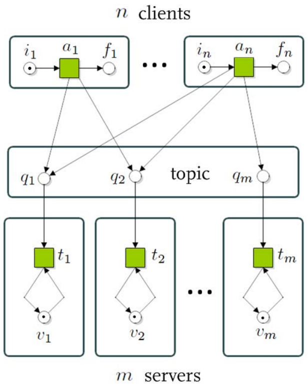


Fig. 4. Publish-Subscribe Pattern with n clients and m servers

4.5 The Publish-Subscribe Pattern

The PS pattern consists of multiple servers and multiple clients. The client and servers of a PS pattern communicate over one topic. A *topic* is modeled as a set of interface places. A *server* is a component with one transition connected to an input place. A *client* is a OWN with one transition, firing which produces copies of the same message in the input place of each server. Fig. 4 presents the *logical model* of the publish-subscribe pattern. Note that places of a PS pattern are not refinable.

Definition 12 (Publish-subscribe pattern). *We use the notations of Fig. 3 to define the* PS pattern $(\mathcal{C}, \mathcal{S})$*, where* $\mathcal{C} = \{C_1, \ldots, C_n\}$ *denotes the set of* n *clients and* $\mathcal{S} = \{S_1, \ldots S_m\}$ *denotes the set of* m *servers. The* idle *place of each server* S_i*,* $1 \leq i \leq m$ *is labeled as* v_i*.*

For the publish-subscribe pattern we define the system $PS(\mathcal{C}, \mathcal{S}) = (N, M_0, M_f)$ *by* $N = \mathcal{N}(\mathcal{C}, \mathcal{S})$*,* M_0 *is defined by* $m \in M_0$ *iff* $m(v_j) = 1$ *for all* $1 \leq j \leq m$*,* $m(i_j) \leq 1$ *for all* $1 \leq j \leq n$ *and all other places are empty, and the set of final markings* M_f *is defined by* $m \in M_f$ *iff* $m(v_j) = 1$ *for all* $1 \leq j \leq m$*,* $m(i_j) + m(f_j) = m_0(i_j)$ *for each client* C_j *with* $1 \leq j \leq n$ *and initial marking* $m_0 \in M_0$*, and all other places are empty.*

Note that if an instance of a PS pattern has no servers, then the output places of each client yields the empty set. On the other hand, if a PS pattern has no clients, then all servers are dead, i.e. no transition in any client can fire.

Although the PS pattern does not satisfy the proper completion property [7], it is easy to verify that any publish-subscribe pattern weakly terminates.

Theorem 13 (Weak termination of the PS pattern). *Let* $PS(\mathcal{C}, \mathcal{S})$ *be a PS pattern as defined in Def. 12. Then* $PS(\mathcal{C}, \mathcal{S})$ *is weakly terminating.*

Proof. Let m be a reachable marking of $PS(\mathcal{C}, \mathcal{S})$. As for each client $C \in \mathcal{C}$ either place i or place f is marked in m, we only need to consider the interface place

$q \in I_S$ for each server $S \in \mathcal{S}$. By the structure of S, we can fire transition t, $m(q)$ times, after which place q is empty. This is possible for each server. Hence, $PS(\mathcal{C}, \mathcal{S})$ is weakly terminating. □

Like for the first two interaction patterns, the PS pattern can be changed at run-time, by adding or removing clients in their initial marking. In addition, the server of a PS pattern can also be added or removed if its *topic* place is empty.

Corollary 14 (Dynamic reconfiguration of publish-subscribe pattern). *Given a PS pattern* $(\mathcal{C}, \mathcal{S})$ *as defined in Def. 12, let* m *be a reachable marking of* $PS(\mathcal{C}, \mathcal{S})$*. Let* $X, Y \subseteq \mathcal{S}$ *be a partitioning of* $\mathcal{S}$ *such that* $S_j \in X$ *iff* $m(q_j) > 0$ *and* $S_j \in Y$*, otherwise. Then* $PS(\mathcal{C}', \mathcal{S}')$ *is weakly terminating for any set of clients and servers with* $\mathcal{C}' \subseteq \mathcal{C}$ *and* $X \subseteq \mathcal{S}' \subseteq \mathcal{S}$*.*

5 The Construction Method

In this section we will present a construction method to derive weakly terminating systems starting from an architectural diagram. An architectural diagram gives the blue print of the system. From this diagram we proceed in a bottom up manner consisting of two phases: In the first phase, all RPC and GFR patterns are introduced in the right order by successive applications of *simultaneous refinement*. In the second phase, successive insertions of the PS patterns are carried out. We will first describe the architecture diagram.

5.1 Architectural Diagram

Fig. 5 describes the graphical notation to specify the component architecture of a system. The concept is similar to other component models like SCA [1], Koala [11] and UML [6] but is extended with pattern and component types.

An architectural diagram is a directed graph with components as nodes and interaction patterns as edges. To guarantee weak termination, the graph must be acyclic, without taking into account the PS pattern. It is easy to check that a cyclic path indicates a deadlock in the system: to handle its clients, the server needs to finish itself, using a client that can only finish if the server itself finishes, which clearly is a deadlock.

A *component* has a type: being either a GFR, a RPC or a *basic* component. A GFR or RPC component is a server of the corresponding interaction pattern. All other components are referred to as *basic components*. A *component* is denoted by a solid rectangle, and labeled with its name and type. Clients and servers of an interaction pair are depicted by an arc with arrow heads both at its foot and head. Clients communicating to the same server are connected to the same arrow head. A filled arrow head indicates the RPC pattern, an unfilled arrow head the GFR pattern. A server of an RPC pattern has multiple procedures. For each procedure, an explicit incoming arrow head is drawn. A client and server of a PS pattern are denoted by a pair of *circles* that are connected by a directed arrow from the client to the server.

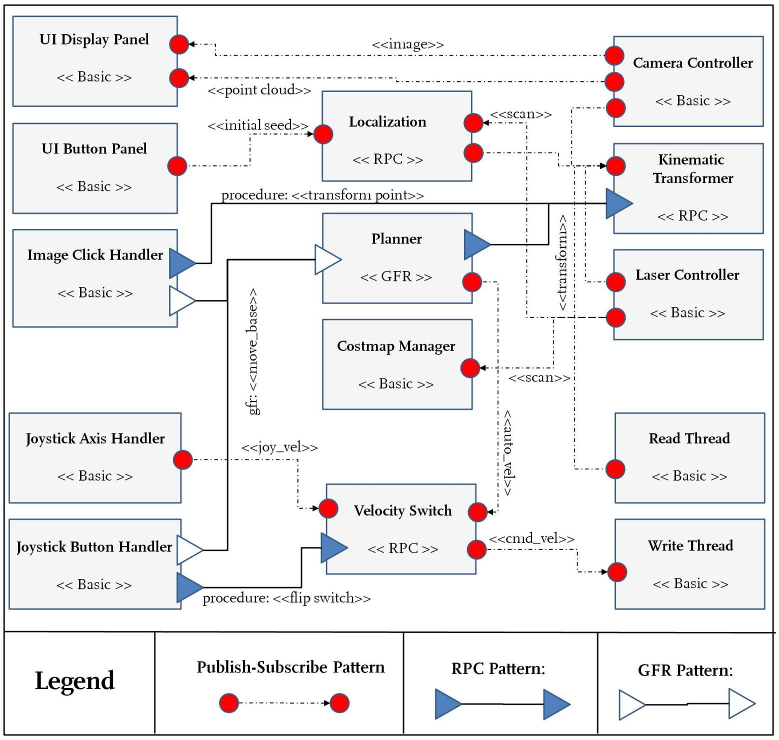


Fig. 5. Architectural Diagram of the Platform Control System (ROSE)

To construct a net from an architectural diagram, we use Algorithm 1. The method prescribes a structured way to introduce communication between components. The method progresses in two steps: first the GFR and RPC patterns are inserted, and next each of the PS patterns. A GFR and RPC pattern can be further refined with a client of an interaction pattern or S-WFN. All places except the initial and final place of a S-WFN and basic component are refinable. We do not require to design a system using this algorithm, it should only be possible to derive the system in this way. Further, it is easy to verify that with the construction method we can realize any architectural diagram that is acyclic.

Theorem 15 (Preservation of weak termination). *The construction method preserves weak termination.*

Proof. (sketch) The construction method starts with a set of basic components. From [7,8], we know that each basic component is weakly terminating. From Thm. 7 and Thm. 10, we conclude that each RPC and GFR pattern has the same behavior as their clients in isolation as classical workflow nets (i.e. without interface places). From [7], it is well known that refinement of a safe place by a weakly terminating workflow net preserves weak termination. Hence, in each step the refined system is weakly terminating.

```
   Input  : Architecture Diagram
   Output: Logical Model of a Weakly Terminating System
 1 Identify basic components and define their behavior;
 2 The initial system is the union of all basic components;
 3 Identify interaction patterns and the components they occur;
 4 while not all RPC or GFR patterns have been added do
 5     Choose an RPC or GFR pattern such that for all clients there exists
       components in the system so far;
 6     Identify the set of places to be refined (one for each client) and perform a
       simultaneous refinement with the chosen pattern;
 7     If necessary, refine a place with an S-WFN;
 8 end
 9 while not all publish-subscribe patterns have been added do
10     Choose a publish-subscribe pattern for which a set of refinable places (one
       for each client) and a set of idle places (one for each server) already exists in
       the system so far;
11     Perform an insertion of the chosen pattern (see Def. 5);
12     If necessary, refine a place with an S-WFN;
13 end
```

Algorithm 1. Construction Method

After all RPC and GFR patterns have been introduced, the PS patterns are added. From Def. 12, we know that a PS pattern is weakly terminating. The *insertion* of a PS pattern (see Def. 5) on top of an already weakly terminating system does not inhibit or extend the behavior of the original system. The only difference is that now clients of a PS pattern can put tokens in the topic input places of server transitions that are inserted in RPC or GFR servers. These transitions are connected to the idle place of servers with bi-directional arcs. Since the whole system so far was weakly terminating, it is able to reach the state with the idle place of a server marked. But then these inserted transitions can consume the tokens of the topic input places. □

ROS allows for dynamic reconfiguration. In order to guarantee weak termination for such systems at run-time, components can only be added or removed when they are in their initial state. Also interaction pattern can be changed at run-time. A client can be inserted in a component if the place to be refined is empty. It can be removed if it is not marked. Introducing a new RPC or GFR pattern, which is in fact the addition of a new component, can be done at any point in time, as long as in the other components the places to be refined with a client are empty. To remove the server of a PS pattern we must first ensure that its *topic* place is empty. To remove a server of an RPC or GFR pattern, its set of clients should be empty and it must be idle.

6 Case Study: Navigation System

In this section, we will discuss the design of a navigation system for the robot ROSE. This case study considers only a subset of the hardware capabilities available on ROSE: (a) four wheel drive mobile platform with individually steerable wheels, (b) stereo camera, (c) laser scanner and (d) joystick. We begin with a sketch of the functional requirements for the navigation system.

The user must be able to move the platform using a joystick. The images streaming from the camera must be displayed on the user interface. The user must be able to give goals by clicking on locations on this image and then the robot must be able to plan its path and move to the desired location while avoiding obstacles. The progress of the robot must be visualized on a map displayed on the user interface. The user always has the possibility to cancel the currently pursuing navigation goal. Furthermore, the user can always take over/give back the control from/to the navigation system by pressing a joystick button. During this switch, the navigation goal being pursued must not be canceled. This leads to the following set of components:

The *Laser Controller* is a basic component that reads and transforms range data from a laser and publishes them on the topic *scan*. The *Camera Controller* is a basic component that connects to a stereo camera and publishes one of the images on the topic *image*. The basic component also generates a point-cloud and publishes them on the topic *point cloud*. The two basic components also publish transformations between the coordinate frames of the respective devices and platform on the topic *transform*.

The motion of the *Platform* is controlled by two basic components. The *Read Thread* periodically reads encoder values from the four wheels. Using the current and last read encoder values, odometry data is published on the topic *transform*. The *Write Thread* has a simple control loop that reads the current set point and generates control signals based on changes in encoder values such that the platform achieves the desired set point as soon as possible. The *Write Thread* has a server of a PS pattern listening on the topic *cmd_vel*. Messages arriving on this topic carry set point values.

The *User Interface* consists of three basic components. The *UI Display Panel* subscribes to the topics *camera images* and *point cloud* published by the *camera controller* and displays them. The *UI Button Panel* waits for a button event from the user interface containing the initial position of the robot and publishes it on the topic *initial seed*. The *Image Click Handler* waits for a mouse click event on the cockpit image. The coordinates of the mouse click is correlated to a 3D point in the point cloud which is then kinematically transformed into world coordinates by invoking the service *transform point*. The transformed point is sent as a goal to the GFR server labeled *Planner*.

The events generated by the *Joystick* are controlled by two basic components. The *Joystick Axis Handler* transforms joystick movements into velocity commands for the platform and is then published on the topic *joy_vel*. The *Joystick Button Handler* waits for a joystick button press event. Two types of button

Phase 1: Stepwise Refinement with RPC and GFR patterns

Step No.	Refinement Type	Pattern Type	Refining Components	Components Added
1	Place Refinement	S-WFN (with choice)	Joystick Button Handler	None
2	Simultaneous Refinement	RPC	Joystick Button Handler	Velocity Switch
3	Place Refinement	S-WFN	Velocity Switch	None
4	Simultaneous Refinement	GFR	Image Click Handler, Joystick Button Handler	Planner
5	Place Refinement	S-WFN (Navigation Algorithm)	Planner	None
6	Simultaneous Refinement	RPC	Image Click Handler, Planner	Kinematic Transformer

Phase 2: Insertion of PP patterns

Refining Components	Server Components with idle place	Topic Name
Camera Controller, Laser Controller, Localization, Read Thread	Kinematic Transformer	transform
Velocity Switch	Write Thread	cmd_vel
Joystick Axis Handler	Velocity Switch	joy_vel
Planner	Velocity Switch	auto_vel
Camera Controller	UI Display Panel	image
Camera Controller	UI Display Panel	point cloud
Laser Controller	Costmap Manager, Localization	scan
UI Button Panel	Localization	initial seed

Fig. 6. Construction Steps

events are distinguished: one of the button events invokes the RPC service *flip switch* and the other invokes the GFR server *move_base* with a goal containing *home coordinates* for the robot to navigate to.

The *Navigation System* is made up of three components. The *localization* subscribes to the topic *transform* and *scan* and publishes an estimate of the robot's current location on the topic *transform*. The *Planner* is a GFR server that takes a goal describing the destination coordinates and in turn generates a sequence of velocity commands on the topic *auto_vel*. These commands drive the platform to the destination while avoiding obstacles. The planner also makes use of an RPC client to invoke the procedure *transform point* to perform kinematic transformations on a goal coordinate. The *Costmap Manager* maintains a representation of the immediate environment of the robot by subscribing to the topic *scan*.

The *Velocity Switch* is a RPC component with one procedure *flip switch*, two PS servers listening on the topics *joy_vel* and *auto_vel* and a PS client that publishes the last recorded command from either *joy_vel* or *auto_vel*. Every time the procedure *flip switch* is invoked by the *Joystick*, the PS client toggles between the two commands. Note that the PS client belongs to the internal part of the RPC server.

The *Kinematic Transformer* is a RPC component that listens to transformations between the different joints of the robot on the topic *transform* and builds and maintains a kinematic tree of joints. The RPC component has one procedure to carry out transformations on a coordinate point between coordinate frames.

An architectural diagram of the system is shown in Fig. 5. Fig. 6 describes the construction of the navigation system in a stepwise manner. The logical model of the resulting system is shown in Fig. 7. The implementation of the navigation module for robot ROSE conforms to this logical model.

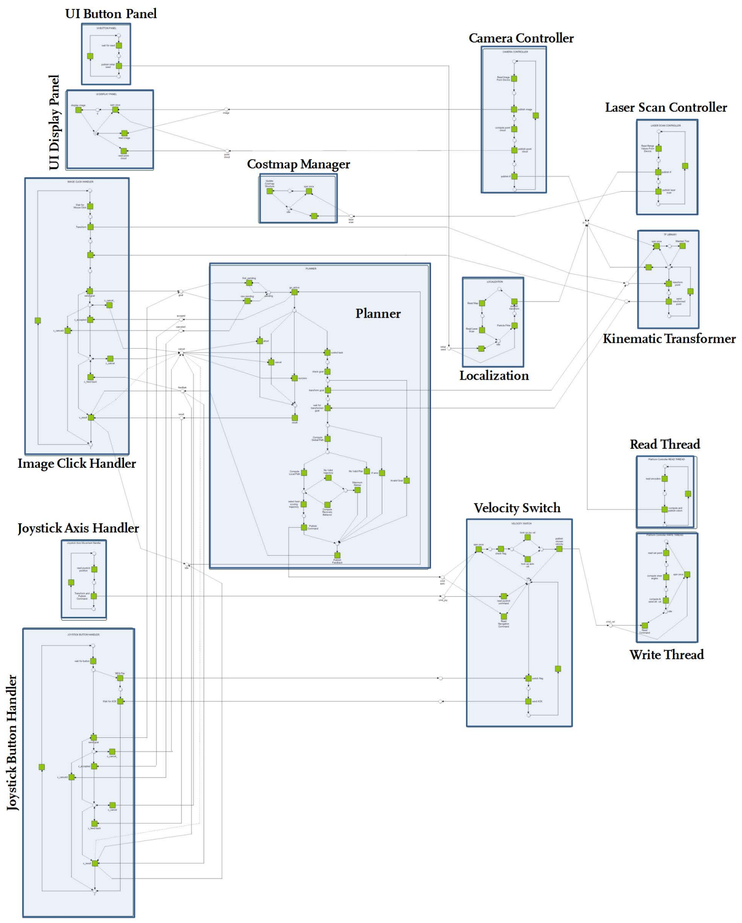


Fig. 7. Weakly Terminating Logical Model of the Navigation System

7 Conclusions

In this paper we presented a method to design ROS systems that are weakly terminating by construction. The method is based on familiar stepwise refinement principles. We used Petri nets to model ROS components as strongly connected state machines and the three interaction patterns provided by ROS as multi-workflows. We tested the method on a concrete robot system, ROSE, and it was straight forward to structure the program code according to this method. As ROSE is a standard service robot, we expect that the method works for other robot systems as well. A nice feature of our method is that it allows dynamic reconfiguration. Petri nets were a good choice because the systems we model are component-based with asynchronous communication and the weak termination property is well-studied within the framework of Petri nets. The case study shows our approach to be a practical method for structured programming of

ROS systems. We want to generalize the method to other interaction patterns that have the same structure of clients and servers, i.e., the servers cannot prevent clients to be weakly terminating. Although weak termination at the level of control flow is an important sanity check for systems, it is only a starting point for system verification. Data manipulation by transitions can destroy the weak termination property. We have to check this as well, but this can be done locally. Robot systems are typical real time systems, and the weak termination property we considered abstracts from time. Our aim is to extend the framework with time to be able to make statements about the time needed to reach a final state from an arbitrary state.

References

1. van der Aalst, W.M.P., Beisiegel, M., van Hee, K.M., Konig, D., Stahl, C.: A SOA-based Architecture Framework. International Journal of Business Process Integration and Management 2, 91–101 (2007)
2. Ando, N., Suehiro, T., Kotoku, T.: A Software Platform for Component Based RT-System Development: OpenRTM-AIST. In: Carpin, S., Noda, I., Pagello, E., Reggiani, M., von Stryk, O. (eds.) SIMPAR 2008. LNCS (LNAI), vol. 5325, pp. 87–98. Springer, Heidelberg (2008)
3. Bruyninckx, H.: Open Robot Control Software: the OROCOS project. In: IEEE Int. Conf. Robotics and Automation, pp. 2523–2528 (2001)
4. Dijkstra, E.W.: Letters to the editor: Go To Statement Considered Harmful. Commun. ACM 11, 147–148 (1968)
5. Gerkey, B.P., Vaughan, R.T., Howard, A.: The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor Systems. In: Proceedings of the 11th International Conference on Advanced Robotics, pp. 317–323 (2003)
6. Object Management Group. OMG Unified Modeling Language (OMG UML), Superstructure V2.3. Object Management Group (2010)
7. 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)
8. van Hee, K.M., Sidorova, N., van der Werf, J.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)
9. 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)
10. Murata, T.: Petri nets: Properties, analysis and applications. Proceedings of the IEEE 77(4), 541–580 (1989)
11. van Ommering, R., van der Linden, F., Kramer, J., Magee, J.: The Koala Component Model for Consumer Electronics Software. Computer 33, 78–85 (2000)
12. Quigley, M., Gerkey, B., Conley, K., Faust, J., Foote, T., Leibs, J., Berger, E., Wheeler, R., Ng, A.Y.: ROS: an open-source Robot Operating System. In: Open-Source Software Workshop of the International Conference on Robotics and Automation, ICRA (2009)
13. Robot ROSE website, `http://www.robot-rose.nl/`
14. ROS website, `http://www.ros.org/`

Nets-within-Nets to Model Innovative Space System Architectures

Frédéric Cristini and Catherine Tessier

ONERA, The French Aerospace Lab, Toulouse, France
{frederic.cristini,catherine.tessier}@onera.fr

Abstract. This paper focuses on a nets-within-nets approach to model, simulate and evaluate innovative space system architectures. Indeed, as military Earth observation space systems are more and more subjected to a wide spectrum of emerging space threats and attacks, it is necessary to devise new architectures and to assess their performance and their threat-tolerance. We will consider networked constellations of autonomous microsatellites - or Autonomous Networked Constellations (ANCs) - in which several heterogeneous interacting entities (satellites and ground stations), ruled by the space dynamics laws, rely on resources (functions or sub-systems) to achieve the overall Earth observation mission. Petri nets are well adapted for modeling ANCs as they feature event-triggered state changes and concurrent communication accesses. The ANC model is based on *Renew 2.2* which we use to deal with top-down, bottom-up and horizontal synchronizations so as to represent state propagations from an entity to its nested resources and vice-versa, and from an entity to another one. A model and simulation of a simple ANC subjected to threats are given and discussed.

Keywords: Experience with using nets, case study, nets-within-nets, Reference nets, space system design, satellite networks, threat-tolerance.

1 Introduction

Traditional Intelligence, Surveillance and Reconnaissance (ISR) space systems consist of a limited number of very expensive monolithic satellites, performing optical, infrared or radar observations, or electronic and communications intelligence. Satellite tasking is performed by dedicated Users Ground Centers (UGC), also known as mission centers. On a periodical basis, UGCs send their requests to a Command and Control Center (CCC) which computes mission plans. Those plans are then uploaded to the satellites thanks to a network of Tracking, Telemetry and Command (TT&C) ground stations. This network is also used by the CCC to monitor the state of the satellites. Once raw data have been gathered by the satellites, they are downloaded back to the UGC through specific Receiving stations.

ISR space systems are considered as particularly sensitive and require high availability and robustness. In order to meet those requirements despite the natural space environment hazards and the limited recovery options, the traditional design approach consists in implementing redundant equipments in each monolithic satellite, along with local Fault Detection, Isolation and Recovery (FDIR) algorithms [19]. This leads to complex integration, validation and verification processes and thus to very expensive satellites.

Despite those efforts, ISR satellites remain vulnerable to an emerging class of highly unpredictable threats: a wide range of *space negation capabilities*, including cyber-attacks and directed energy[1] or kinetic energy[2] weapons, are being developed throughout the world [10, p.149-160]. In the meantime, collision risks caused by orbital debris become a growing concern for operational satellites [10, p.27-43]. As traditional space systems are not designed to resist such threats, their chances to be damaged or destroyed are thus increasing.

In order to deal with this new paradigm, we propose to apply the traditional strategy (redundancies + FDIR) to a higher design level, in the context of fractionated spacecraft [2]. This leads us to define two complementary concepts to improve the robustness of space systems: *passive* and *active robustness*.

Passive robustness focuses on the physical architecture of the space segment of the system, *i.e.* the types, number and orbits of satellites. Its objective is to minimize immediate consequences of aggressions. Inspired by the fractionation concept [2], passive robustness is based on networked constellations of redundant small satellites, called "*networked constellations*" (NCs) [8]. In NCs, the military reconnaissance mission is achieved by a constellation of "payload (P/L) satellites" on very low Earth orbits (VLEO). This constellation is supported by another constellation of "support satellites" on higher orbits, enabling P/L satellites operations (communication relays, data handling, computation functions, etc.). P/L and support constellations are two layers of a dynamic space network. Each layer behaves like a distributed sub-system and the connections between P/L and supports, performed by intersatellite links (ISLs), create dynamic virtual satellites (Fig. 1).

Active robustness aims at increasing the system availability and efficiency in the aftermath of an aggression. This is achieved thanks to short-term recovery capabilities accomplished by autonomous on-board algorithms, ground control, or a combination of both. The detection of off-nominal conditions, the isolation of the failure to a specific subsystem and the recovery of nominal or degraded capabilities are performed thanks to FDIR strategies [5, 16].

NC configurations combined with some FDIR strategies have been modeled and simulated thanks to a set of classical computer simulations, using ephemerids generated with *Satellite Tool Kit 8* ®, the space mission simulator developed by *AGI*, and *Excel* ® and *MATLAB* ® computations. The operational performance of the NCs (revisit rate, accessibility, etc.), their communicability (network density, resource sharing conflicts, etc.) and their robustness (operational consequences of satellite failures) have been evaluated and compared [6–8]. Network metrics [1], such as routing delays or traffic overload caused by FDIR information exchange, have also been assessed. The preliminary results show that

[1] Lasers or micro-waves.
[2] *i.e.* antisatellite missiles.

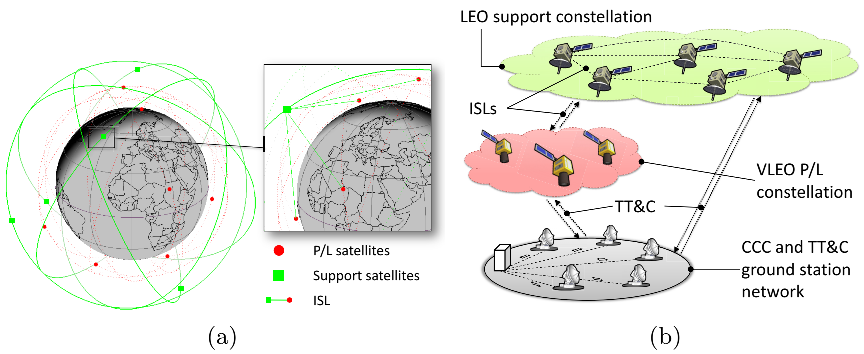


Fig. 1. Example of NC configuration: orbital configuration (1a) and functional configuration (1b)

threat-tolerance of sensitive space systems can be drastically improved thanks to NC architectures and autonomous FDIR strategies. However they also reveal plenty of concurrency issues between the satellites within NCs, that have to be modeled and assessed. Moreover, in the simulations described above, we only considered *satellite losses* to evaluate the degradation of operational performance of nominal systems. Even if this rough approach is sufficient for a preliminary analysis of threat tolerance of NCs with a large number of satellites, a more accurate description of consequences of aggressions is required, based on satellite functions or equipment losses, to design redundancy architectures and FDIR strategies.

In the remainder of the paper, we propose to call networked constellations equipped with on-board autonomy for mission achievement and FDIR *Autonomous Networked Constellations* (ANCs). An ANC will be considered as a *hierarchical and cooperative multiagent system*, including *entity-agents* (ground stations, P/L satellites, support satellites) and *resource-agents* (communication, Attitude and Orbit Control System (AOCS) or payload equipments, etc.)

The ultimate purpose of this study is to make a trade-off between several ANC hardware (number, types and distribution of entities, etc.) as well software (autonomy organization pattern, FDIR strategies, etc.) architectures. To this end, we must be able to simulate and evaluate nominal and degraded performance of various ANC configurations in order to rank ANCs' architectures, which requires to:

- easily create, handle and study several ANC hardware and software configurations, which are large networked space systems with heterogeneous entities;
- manage *transient communication sessions* between these entities, which are ruled by space dynamics laws; interfacing with realistic precomputed ephemerids is strongly considered;
- study *post-aggression failure propagation* within ANCs, from resource-agents to entity-agents;
- assess *several reconfiguration strategies*, which imply upward and downward interactions between entities and resources.

This paper is organized as follows: in section 2, we will detail the hierarchical multiagent description of ANCs. In section 3, we will describe the specific nets-within-nets approach chosen for our work, namely *Reference nets*, as well as the tool, *Renew 2.2*. Finally in section 4, we will assess a simplified example of an ANC configuration modeled with *Renew* as a set of *Reference nets* and we will discuss preliminary results.

2 ANC Model Features

In this section, we describe the detailed features of an ANC and specify what is needed for its modeling.

2.1 ANC Model Structure

The static structure of an ANC model should include the following levels (Fig.2):

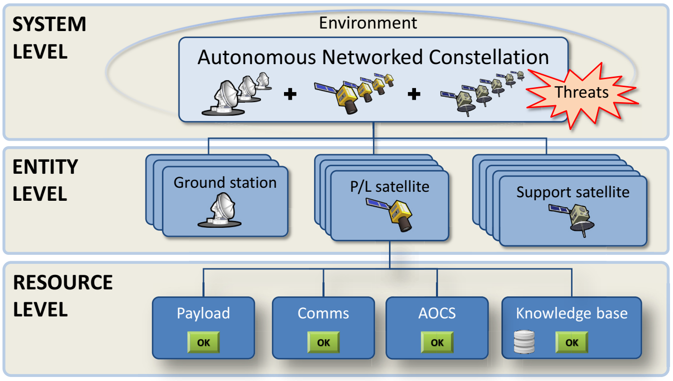


Fig. 2. Multilevel breaking down of the ANC model structure

System Level. It is the highest level of the ANC model. It represents the ANC in its environment and contains a reference to each satellite and ground station of the ANC. In this paper, we only consider a limited model of the environment of ANCs: the Earth, which provides its gravity field and enables orbital moves, the Earth surface, which is observed by the P/L satellites and on which ground stations are set up, and the near-Earth space where satellites orbit. Threats, that can strike any part of an ANC, are also part of the environment.

Entity Level. This level represents the state of each mobile (P/L or support satellites) or fixed entity (ground stations), within the "system". Each entity performs a part of the global ISR mission (data collection or storage, communication, mission planning, system monitoring, etc.) and requires low-level resources to achieve those goals. The number of P/L satellites, support satellites and ground stations is set for each considered ANC configuration [6].

Resource Level. This level describes the availability of low-level physical or logical components which are considered as resources for "entities": payload, communication, AOCS, etc. It only comprises the relevant functions to be conside-red for threat-tolerance analysis, *i.e.* the most likely to be targeted by aggression means. The "knowledge" of each entity is also stored in a resource called "knowledge base". This knowledge may be about its current state, its environment or the state of other entities.

2.2 ANC Model Dynamics

The dynamic features of the different levels are as follows:

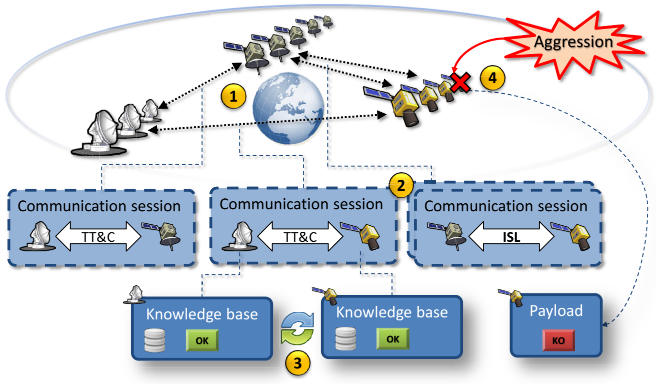


Fig. 3. Examples of interactions in an ANC.
①: inter-entity accesses ruled by the space dynamics laws.
②: ongoing communication sessions (TT&C and ISL).
③: synchronization of two knowledge bases thanks to a communication session.
④: aggression on a P/L satellite - its payload becomes unavailable.

Environment. As it is ruled by the space dynamics laws, the ANC environment may be considered as deterministic: access periods between entities are foreseeable and can be computed thanks to ephemerids that sample position and velocity of each entity over time (Fig.3.①). As for threats, their activity and their precise effects remain highly unpredictable. Any aggression may, however, result in either damages at the resource-level and thus modify the availability of the entity functions (Fig.3.④), or destroy an entire entity.

Entity and Resource-Agents. As described in section 2.1, an heterogeneous set of agents is considered in our model: three types of entity-level agents, also called *entities*, (ground stations, P/L and support satellites) and several types of resource-level agents, also called *resources*. Although those agents are cooperative to achieve the ISR mission, concurrency may emerge, especially between entities (P/L satellites *vs* support satellites, or satellites *vs* ground stations). For example, a ground station may have a simultaneous access with two satellites.

For this concurrent situation, either the station has two available antennas and can communicate with both satellites at the same time, or it has to "choose" between them, according to rules (order of priority, first-come/ first-served, etc.) or routing tables. It is noticeable that there is no concurrency between entities to use resources: we assume that entities, which are mainly satellites, are correctly designed and have enough resources to achieve their part of the mission in nominal conditions. In case of an aggression, those resources may however be affected and failures may propagate within the whole system: for instance, the jamming of the communication sub-system (a resource) of a satellite (an entity) may impair the whole system communication pattern. Among the other resources, one should note the specific role of the "knowledge base": it is used to store the current state of each entity with regard to their other resources, as well as the state of the other entities. This knowledge is updated through communication sessions between entities (Fig.3.③).

Communications. By definition, ANCs form dynamic (but deterministic) mesh networks and each entity is a node of this network [6]. Like in any low Earth orbit space system, communications between the ANC entities are necessarily transitory. Those connections are established either by ISLs for communications between satellites, or by TT&C links for communications between satellites and ground stations (Fig.3.②).

Entities can interact in a deterministic and periodical manner [6]. In nominal situations[3], communication periods are determined by space dynamics as well as communication rules and protocols: in order to exchange messages, two satellites must firstly be in mutual visibility, which is determined by their orbits, and secondly be planned to communicate. As we will only focus on those access/no-access periods, we will consider ANCs as discrete-event dynamic systems, without taking into account continuous-time dynamics.

Interactions between entities mainly consist in message passing (concerning the entity current state, operational information, new assignments, communication relays, etc.) Those messages may be used to update each entity's knowledge base, to assign tasks to entities or to broadcast reconfiguration orders.

2.3 Requirements for a Modeling Tool

As shown in the previous two paragraphs an ANC model includes several types of entities with embedded resources and subjected to different kinds of events coming from themselves or from the environment. Moreover different ANC configurations including several tens of entities must be modeled easily so that their performance and robustness to aggressions should be assessed and compared. Therefore a modeling tool for ANCs should have the following features:

- allow easy entity creation or removal (*i.e.* when a satellite is destroyed by an aggression);
- allow a compact representation of the state of an ANC, given the states of the entities, resources and communications;

[3] *i.e.* with no failures and no aggressions.

- represent communicating objects;
- represent event-driven state changes (begin/end of visibility; begin/end of communication; aggressions and failures; reconfiguration orders...)
- represent synchronization and concurrency;
- allow a hierarchical representation of objects (*i.e.* entity-embedded resources) with state propagation within the hierarchy:
  - horizontal state propagation: entity $\longrightarrow$ entity;
  - vertical state propagation: top-down (entity $\longrightarrow$ its resource(s)) and bottom-up (resource $\longrightarrow$ its entity).

Given these requirements, *Petri nets*, and especially their *high-level* extensions, appear to be an obvious and particularly relevant modeling formalism for ANCs. Traditional high-level Petri net extensions, like colored Petri-nets [11], are however not entirely suitable for ANC modeling and simulation as they do not explicitly allow tokens to be nets themselves and do not implement net instantiation. We finally chose the "nets-within-nets" Petri net extension which will be reminded in the next section.

It should be noted that although *Petri nets* have been widely used to study event-driven and concurrent complex systems [9] as well as multiagent and multirobot systems [4, 20] or to develop software with agent-oriented paradigms [3], very few references are available concerning space system design [16].

3 From Nets-within-Nets to Reference Nets

Before modeling and simulating an ANC as a set of nets-within-nets, we have to determine the precise nets-within-nets formalism as well as the software we will use.

3.1 Reference Nets and *Renew*

Over the past decade several "nets-within-nets" approaches have been introduced to model hierarchical multiagent distributed systems: Elementary Object Petri Nets [21], Nested Petri Nets [18], Reference Nets [13] or Mobile Object Net Systems [12].

Each formalism features different inter-net connection schemes: indeed the number of hierarchical levels may be limited (*e.g.* two levels [21]), the ability to communicate or synchronize transition firings may be restricted in order to preserve some formal properties (*e.g. decidability* [18]), or specific semantics may be considered (*e.g. value semantics* [12]). Those limitations or specific features may appear problematic to model ANCs. Moreover there are few available software tools to design and run such "nets-within-nets".

As for Reference nets, they implement the nets-within-nets concept thanks to a special inscription language using Java expressions, which may control transition enabling and can also create new instances of subnets. A given net instance can communicate with another one provided it has a reference of this instance and that those nets have linked transitions thanks to so-called "synchronous channels" inscriptions.

Those features have been implemented in an academic open source Petri net simulator, called *Renew*[4], whose current version, *Renew 2.2*, was released in August 2009 [14, 15]. The developer team still provides support.

3.2 Some *Renew* Specific Features

Net Instances. *Renew* differentiates typical static nets that are drawn in the graphical editor and considered as templates, and net instances that are dynamically created from net templates during the simulation [14, p.43].

As presented previously (section 2.1), ANCs are composed of several "copies" of a limited number of different entities (and resources): P/L satellites, support satellites, ground stations, etc. Therefore each type of entity will be modeled as a specific Petri net template, and their instances will represent the different P/L satellites, support satellites or ground stations within the considered ANC configuration.

Transitory communication links (section 2.2) may also be considered as transitory Petri nets: thanks to *Renew* features, an unlimited number of communication nets can be instantiated "on the fly".

Finally, the hierarchical structure of ANCs is immediately obtained when instantiating nets within other nets.

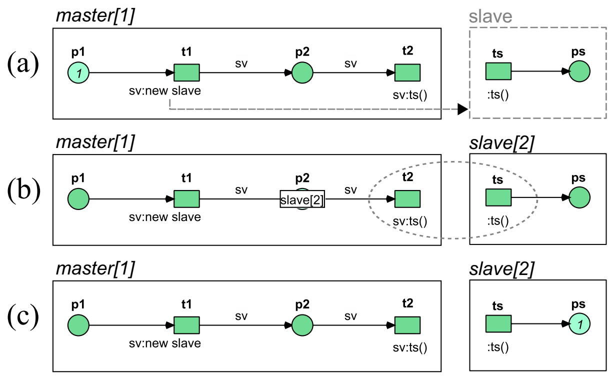
However instances cannot be erased or removed from the simulation: if all transitions of an instance are disabled, this instance is "forgotten" by the simulator, thanks to a garbage collector. Consequently we have to design nets very carefully in order to be sure that unused instances will not remain active and disturb the simulation run.

Synchronization Schemes. *Synchronous channels* are one of the specific features of reference nets: they enable nets to influence each other [14, p.43-47]. This feature is mandatory to simulate communicating entities or event-driven net evolutions, or to synchronize net activities. Synchronizations will be extensively used to model ANCs.

Within the hierarchical architecture of ANCs, we have to consider top-down, bottom-up and horizontal synchronizations. Even if they are made possible by *Renew*, they require different types of techniques:

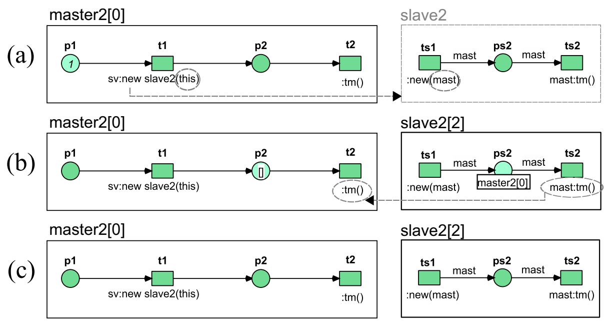
– *Top-down synchronization* (Fig.4) is the simplest one as the master "knows" the slave-instance it creates;
– *Bottom-up synchronization* (Fig.5): the slave net must "know" its master: a reference to the master must be passed to the slave instance;
– *Horizontal synchronization* (Fig.6) is quite similar to the bottom-up synchronization; each slave must "know" the other one.

[4] For REference NEts Workshop.



(a): Thanks to the inscription sv:new slave, firing t1 creates one instance of the slave net (*slave[2]*). A reference to *slave[2]* is stored in variable sv.
(b): t2 and ts are synchronized thanks to inscriptions sv:ts() and :ts(). Note the net-token *slave[2]* in p2.
(c): Final marking.

Fig. 4. Top-down synchronization



(a): Thanks to inscription sv:new slave2(this), firing t1 creates one instance of slave2 net (*slave2[2]*) and passes a reference to *master2[0]* to *slave2[2]* thanks to keyword this. *slave2[2]* stores this reference in variable mast.
(b): t2 and ts2 are synchronized thanks to inscriptions :tm() and mast:tm(). Note the net-token *master2[0]* in ps2.
(c): Final marking.

Fig. 5. Bottom-up synchronization

4 Application

In this paper, our purpose is to validate our choices about formalism and modeling tools. In this section, we thus present an example of a simplified ANC modeled with *Renew* as a set of Reference nets. In further works, we will implement bigger and more complex ANCs[5].

4.1 A Simplified ANC

The simplified ANC configuration we are considering here is composed of the following entities: 2 P/L satellites, 1 support satellite and 1 TT&C ground station.

[5] *Renew* have been successfully tested with ANCs composed of up to 25 entities and 80 resources.

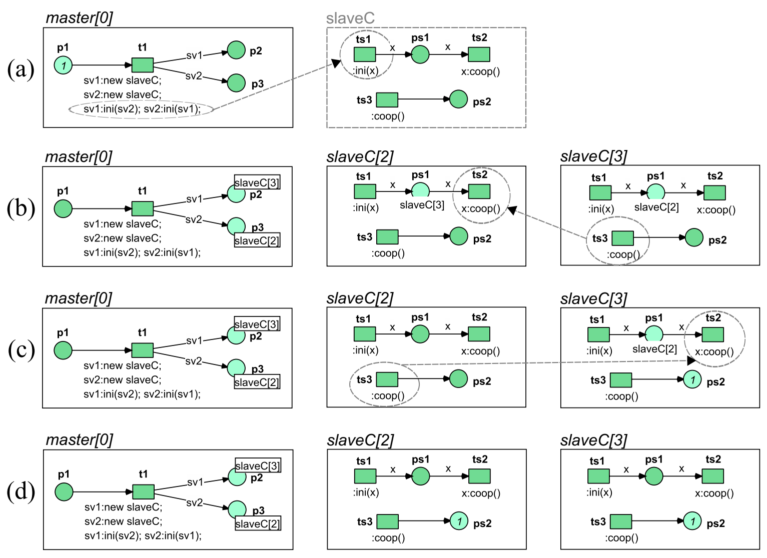


Fig. 6. Horizontal synchronization between two slave-nets.
(a): Thanks to inscriptions sv1:ini(sv2) and sv2:ini(sv1), each slaveC net instance receives a reference to the other slaveC net instance (transition ts1, :ini(x)).
(b): Transitions *slaveC[2]*.ts2 and *slaveC[3]*.ts3 are synchronized thanks to inscriptions x:coop() and :coop().
(c): Transitions *slaveC[3]*.ts2 and *slaveC[2]*.ts3 are synchronized thanks to inscriptions x:coop() and :coop().
(d): Final marking.

Each entity embeds the following resources: 1 communication resource + 1 knowledge base; for P/L and support satellites: 1 AOCS resource; for P/L satellites only: 2 payload resources.

Concerning the dynamic behavior of the ANC, we have made the following simplifying assumptions:

- neither time nor real space dynamics are implemented in our model yet; accesses between entities are thus triggered randomly while complying with the following rules:
  - inter P/L satellite communications are not allowed;
  - the support satellite is a sharable entity: it can simultaneously establish ISLs with the two P/L satellites;
  - the TT&C ground station is an unsharable entity: it can only establish a link with one satellite at a time;
- communication sessions are ruled by a simple exchange protocol: mutual identification of entities with no acknowledgment;
- threats can either aggress any kind of resource, or destroy an entire entity. They are triggered manually.

Although these simplifications make objective performance analysis of ANCs impossible, they enable us to check the ANC hierarchical model as a set of nets-within-nets. Time and accurate space dynamics will be considered in further works so as to perform a thorough performance evaluation of several ANC configurations.

4.2 Nets Description

Each ANC item described in section 2 is modeled as a Petri net.

System Net (Fig.7). It is the highest level net of the ANC and has two main functions: setting up the simulation by creating entity-net instances (static structure of the ANC) and managing the system's dynamics (aggression triggering, ISL and TT&C accesses).

Simulation initialization (Fig.7, frame 1). The manual transition Init_Simu creates instances of entity nets of the considered ANC configuration thanks to several *creation inscriptions*. For example, creation inscription sat1:new satellitePL(this,1) creates a new instance of the P/L satellite net and passes the parameter list (this,1) to this instance, which is assigned to variable sat1. sat0, sat2 and gst0 are instances of, respectively, satelliteSupp net, satellitePL net and groundStation net. An instance of the net init[6] distributes the references of those instances so that they can interact. Those references are stored in places Satellite_Indices, Satellite_List, Station_Indices and Station_List. Those places are then used to manage accesses via *virtual places* which can be seen at the bottom of the system net in ISL and TT&C accesses management areas (frames 3 and 4), with the V. prefix. When Start_Simu transition is fired, the simulation run starts.

Threat management (Fig.7, frame 2). This area is used to manually trigger aggressions on resources or entities. The upper part is dedicated to satellite destruction management, which is an irreversible process (*e.g.* attack of an antisatellite missile). As net instances cannot be removed (see section 3.2), P/L satellites and support satellites are dealt with differently: the payload activity in satellitePL instances is disabled thanks to inscription listSat[sat]:stopPL(). The lower part is dedicated to resource threat management. It is composed of three frames, respectively for threats on communications (*e.g.* jamming), payload (*e.g.* dazzling) and AOCS (*e.g.* GPS jamming). Transitions with inscriptions thrC(com), thrP(pl) and thrA(aocs) are fired during the initialization phase. Then a reference to each resource net is stored in the central place of each net. When transition :agress is fired manually, one resource is made unavailable until reconfiguration (manual transition in resource nets).

Access management (Fig.7, frames 3 and 4). Those nets simulate access and communication periods between entities. As neither time nor real space dynamics are implemented yet, a very simple dynamic model is considered: for ISLs, place Possible_Access_ISL contains a list of possible ISL accesses, determined externally thanks to satellite ephemerids; as for TT&C links, accesses to ground stations are

[6] Not detailed here; this net has only a functional role.

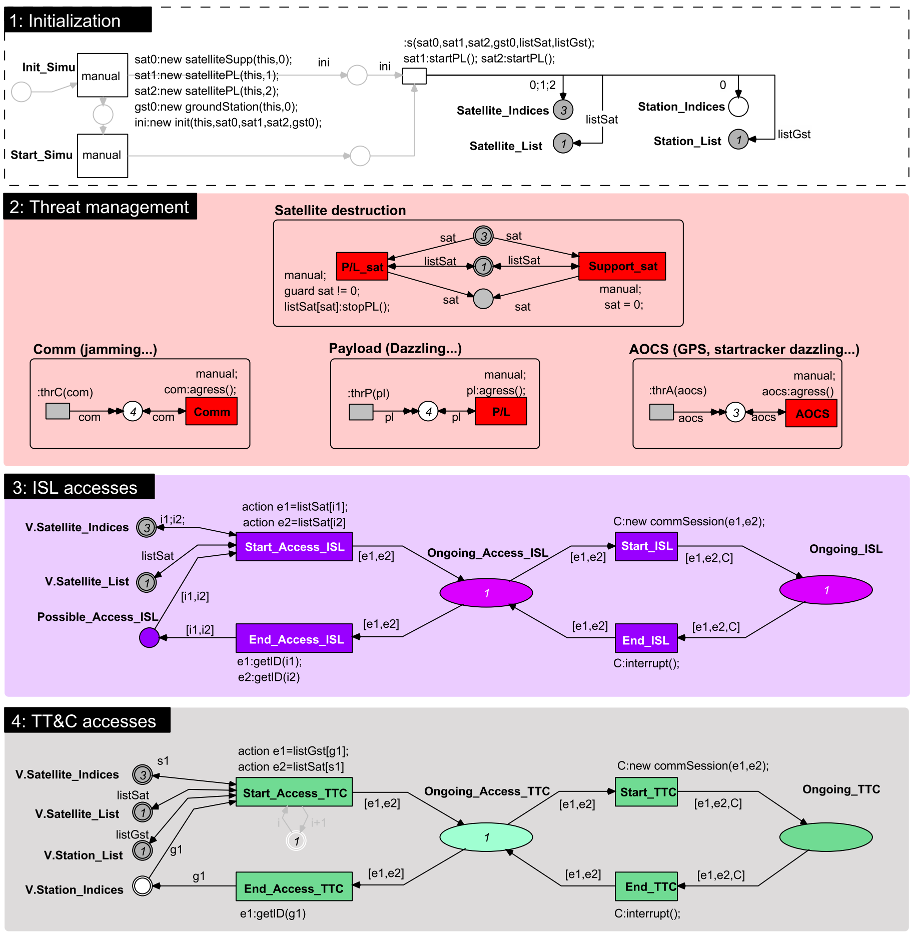


Fig. 7. An instance of *system net*. The marking is composed of colored tokens which may be integers (*e.g.* place Satellite_Indices, 3 tokens in the current marking), tuples (*e.g.* place Ongoing_ISL, 1 token), Java objects (*e.g.* place Satellite_List, 1 token) or net references (*e.g.* places in payload, communications or AOCS threat management, respectively with 4, 4 and 3 tokens). Only the number of tokens is displayed, not their values.

considered as concurrent. When transition Start_Access_ISL fires, an access period between two satellites (e1 and e2) begins (place Ongoing_Access_ISL) and transition Start_ISL is enabled. If it fires, a communication session between e1 and e2 is established (inscription C:new commSession(e1,e2)). Transitions End_Access_ISL and End_ISL put an end to ISL communications and access periods respectively and disable the ongoing communication session (inscription C:interrupt()). This description is also suitable for TT&C access management.

Entity Nets have a common structure:

1. an *initialization area* that instantiates the suitable number and type of the entity embedded resources and stores their references in variables and places;
2. a *functional area* that manages the operational activities of the entity, *i.e.* how the resources are used.

The size of the functional area depends on the type of the considered entity. According to section 4.1, functional areas of support satellite and ground station nets are subsets of the P/L satellite net functional area, which is composed as follows: knowledge base, communication function, AOCS function and payload function (Fig.8). Support satellite and ground station nets are not detailed here. They have the same structure with less embedded resources: knowledge base, communications and AOCS for support satellite nets, knowledge base and communications for ground station nets.

Knowledge base: it is instantiated thanks to inscription kb:new resourceKB(this) and one token kb is put in place Knowledge_Base. Each function of the entity can update the knowledge base according to its state or its activity thanks to top-down synchronous channels (*e.g.* transition Emit with inscription kb:read(msg) on Fig.8). When a communication session is established (inscription :initSession()), the knowledge base is accessed and a message is prepared.

Communication function: the communication resource is instantiated thanks to inscription c01:new resourceCO(this). Token co1 in place Comm_OK sets the resource as "available" for operational use. The right hand part of the communication resource area is dedicated to communication management implemented by transitions Emit and Receive, and to synchronous channels (top-down, with knowledge base: kb:rec(msg) and kb:read(msg); bottom-up, with communication session net: emit(msg) and receive(msg)).

Token co1 can be removed in case of an aggression on the communication resource: a bottom-up synchronous channel (:KO_Comm(), from resource to entity) is activated and the token moves to place Comm_KO. This firing also updates the knowledge base thanks to kb:updateS(0,0). This marking prevents further operational use of the communication function. The resource can be made available again thanks to channel :recovComm(). The availability of the AOCS and payload functions are managed in the same manner.

Payload Function: in the example, it relies on two P/L resources assigned to variables pl1 and pl2 in place PL_OK. Transition Use_PL only requires one P/L resource: this is how we simulate a hot redundancy[7] reconfiguration scheme. The firing of this transition updates the knowledge base thanks to channel kb:updateA(1): the integer value 1 is sent to the knowledge base and increments a counter of operational activity, called *activity index*.

[7] The second resource is immediately available in case the first one becomes unavailable.

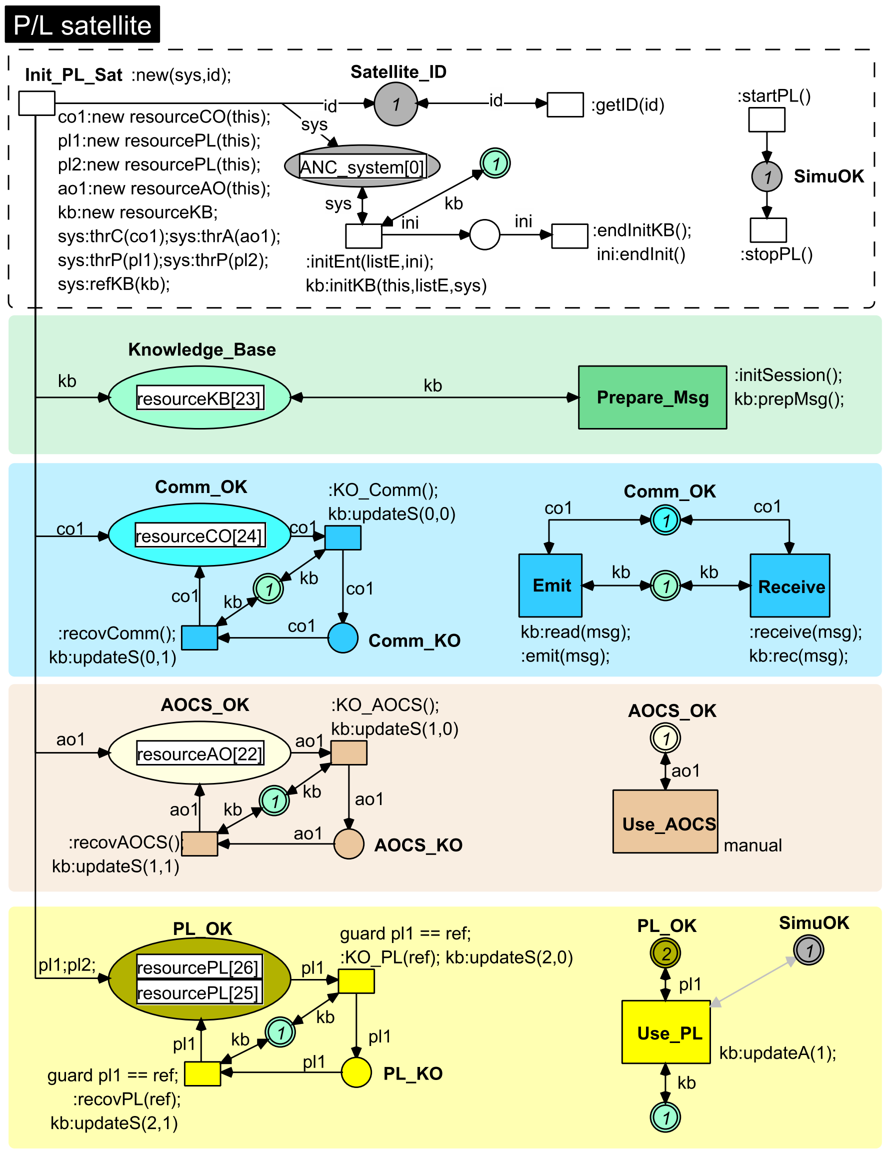
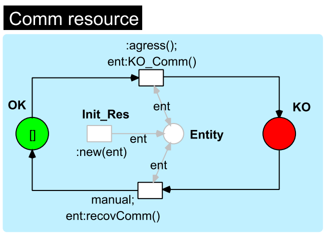


Fig. 8. An instance of *P/L satellite net.*
At the top: initialization area. Below: functional areas. The current marking means that this instance is the number "1" P/L satellite (place Satellite_ID) and that it was created by instance ANC_system[0]. Functions are provided by resources resourceKB[23], resourceCO[24], resourceAO[22], resourcePL[25] and resourcePL[26]. They are all available.

Resource Nets (Fig. 9). In the considered simplified ANC configuration, communication, payload and AOCS resources have the same net structure. The main places of the nets are OK, which is initially marked with a black token, and KO. This token moves to KO when an aggression occurs (inscription :agress()) and moves back to OK when recovery actions are carried out (*e.g.* cold redundancy activation[8], reset of an electronic component, etc.).

In further works, more complex resource models will be implemented (more possible states, various behaviors depending on the resource type, etc.).



The place Entity stores the reference to the upper entity in order to enable bottom-up synchronous channels ent:KO_Comm() and ent:recovComm().

Fig. 9. A *communication resource net.*

Knowledge Base Net (Fig. 10). Its main role is to store the entity's local knowledge about the other entities in a set of tuples that contains the states of all the entities of the ANC. Each "knowledge tuple" (*e.g.* [satellitePL[11],[I@8f5944,420] in Fig. 10) is structured as follows:

- the identification of an entity (*e.g.* satellitePL[11])
- the state of its resources stored in a Java list object (*e.g.* [I@8f5944)
- the value of its *activity index* (*e.g.* 420). This index measures the operational activity of an entity over time: for example, each time a P/L satellite activates its payload, the activity index is increased by 1. It is particularly used during the synchronization phase to compare knowledge freshness between received and stored data.

This net enables interactions with other nets: top-down synchronization with the upper entity to update its activity index (transition Update_Activity) or resource state (transition Update_State) or bottom-up synchronization when a communication session is established between two entities to pass messages (*e.g.* transitions Prepare_Msg or Receive_Msg).

Knowledge is synchronized between entities when they communicate. Emission and reception activities are carried out by the lower part of the net. For emission, the formatted message is only read and sent to the communication function (:read(msg)). For reception, the received message is stored in a buffer (place Input_Buffer) and its data are sorted: the knowledge base only keeps

[8] The second resource is powered off; it needs to be powered up to become available.

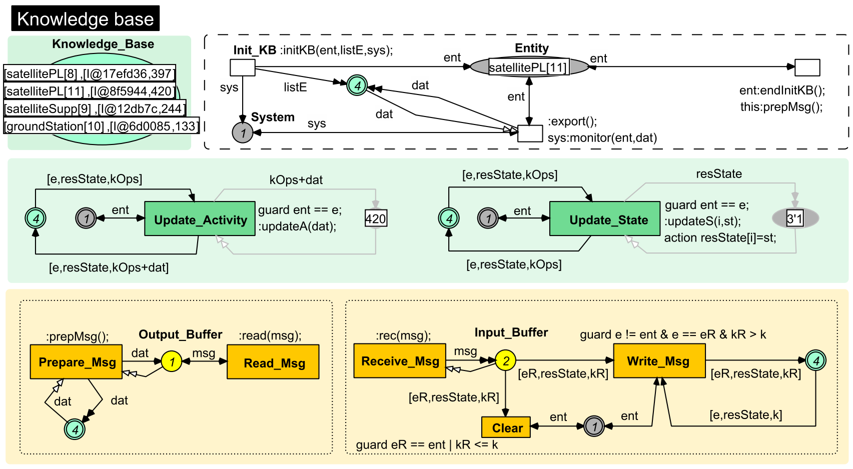


Fig. 10. An instance of a *knowledge base net*. This instance was created by satellite[11] (place Entity). Note the structure of the 4 tokens in place Knowledge_Base. The value of the activity index is 420, i.e. the payload of satellite[11] has been activated 420 times since the beginning of the simulation. The Output_Buffer is ready for the next emission. The Input_Buffer is processing a received message (2 tokens left, i.e. two tuples concerning two entities).

up-to-date data about other resources (transition Write_Msg with guard inscription); out-of-date data or data about its own entity, which is necessarily out-of-date, are cleared (transition Clear).

Communication Session Net (Fig. 11). It is used to model bidirectional communications between two entities. When a communication session starts, each entity receives the identity of the other one. Then, each entity sends the content of its knowledge base (transitions Emit_1 and Emit_2) only once. The ongoing communication session is disabled when the transition :interrupt() is triggered.

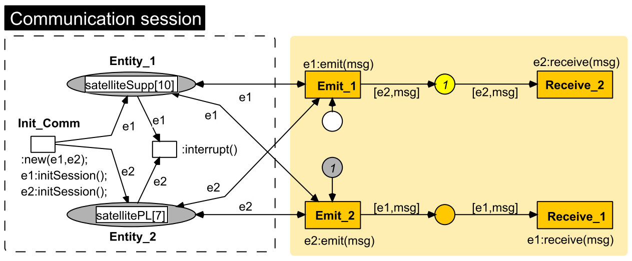


Fig. 11. An instance of a *communication session net* between satelliteSupp[10] and satellitePL[7]. One message msg is being sent by satelliteSupp[10].

4.3 Simulation and First Results

As our main goal is to validate our methods and models, we propose to run the simplified ANC described above with a basic scenario. At this stage of the study, we expect to confirm some earlier results [6–8]. This section focuses on first results obtained with a scenario involving an aggression on the communication resource of the support satellite.

Scenario Description. The simulation starts with the nominal simplified ANC. After a given duration, the communication resource of the support satellite is jammed and permanently made unavailable. Therefore ISL communication sessions no longer enable knowledge base synchronization: P/L satellites keep on emitting messages but the support satellite remains quiet; knowledge base updates cannot be relayed by the support satellite anymore.

Metrics. In ANCs, operational activity of entities is measured through an *activity index* (see description of *knowledge base net*). Due to space dynamics, knowledge bases can only be updated during communication sessions: consequently there always remains residual differences between the knowledge bases of each entity at any simulation step. Those differences are measured through *knowledge discrepancy factors.*

Let $A_{i,j}(s)$ be the activity index of satellite j stored in satellite i at step s and $A_{i,i}(s)$ the "real" activity index of satellite i at step s. For the simplified ANC, we have defined *three knowledge discrepancy* factors KD (≤ 0) between P/L satellites "1" and "2" and ground station "g":

$$(1)\begin{cases} KD_{sat-sat}(s) = \frac{1}{2} \cdot [(A_{1,2}(s) - A_{2,2}(s)) + (A_{2,1}(s) - A_{1,1}(s))] \\ KD_{sat-grd}(s) = \frac{1}{2} \cdot [(A_{1,g}(s) - A_{g,g}(s)) + (A_{2,g}(s) - A_{g,g}(s))] \\ KD_{grd-sat}(s) = \frac{1}{2} \cdot [(A_{g,1}(s) - A_{1,1}(s)) + (A_{g,2}(s) - A_{2,2}(s))] \end{cases}$$

In other words, we calculate the difference between the "real" activity index of each entity, stored and updated within each entity, and the "believed" activity index, stored in the other entities' knowledge bases.

One may consider KD as a performance criterion of ANCs; the less (in absolute value), the better. For a multiagent system, *stability* can be defined as its ability to maintain its performance stationary for a given variation in the system [17]. It is considered as a major property in multiagent system design and evaluation. The stability of our ANCs may thus be assessed as: how does the aggression on the communication resource modify the KD factors?

Experimental design. As time is not implemented in our model yet, it is approximated by a discrete counter of ground station accesses, which are regularly spread over time in the real world. In the following, one simulation *step* will be equivalent to one ground station access, disregarding other Petri net activities.

The knowledge bases of all entities are recorded every 10 steps thanks to both a specific part of the system net (not displayed on figure 7) and log files of simulation traces exported by *Renew* and computed in *Excel* ®. For the considered scenario, the loss of the communication session occurs after 1000 steps. 10 simulation runs have been performed and recorded.

Results. The values of KD factors (average values) are displayed in Fig.12 and Tab.1.

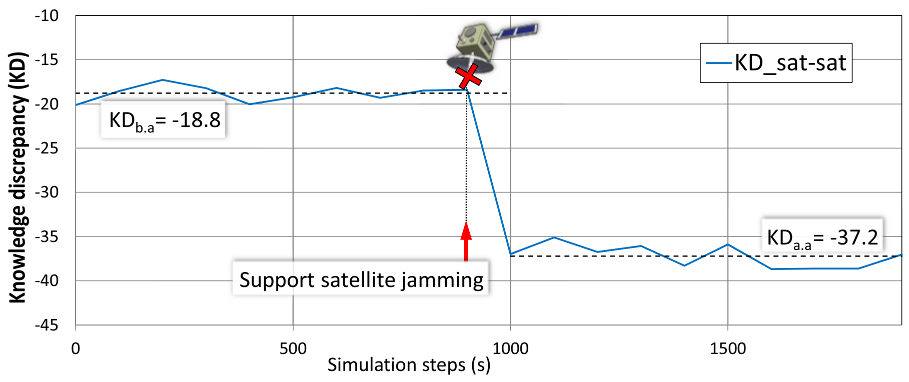


Fig. 12. Knowledge discrepancies over time ($KD_{b.a}$: KD before aggression; $KD_{a.a}$: KD after aggression.)

Table 1. Knowledge discrepancies stability (average values, before and after the aggression)

Knowledge discrepancy	Before aggression	After aggression	Deviation
$KD_{sat-sat}(s)$	-18.8	-37.2	-98.1%
$KD_{sat-grd}(s)$	-2.6	-2.8	-7.7%
$KD_{grd-sat}(s)$	-16.4	-18.1	-10.3%

One may first notice that the aggression on the communication resource results in a *significant deterioration of the knowledge within the whole ANC*: KDs do not remain stable. From the P/L satellites point-of-view, mutual knowledge ($KD_{sat-sat}$) drops of 98.1% on average, and knowledge of the ground station activity ($KD_{sat-grd}$) drops of 7.7%. As for the ground station, its knowledge about P/L satellite activity ($KD_{grd-sat}$) also decreases of 10.3%.

Those preliminary results are coherent and comply with previous results: communication jamming on support satellites results in a significant degradation of the overall performance of the ANC (here the knowledge discrepancy).

5 Conclusion and Further Work

The nets-within-nets approach allows us to design and simulate clearly organized models of ANCs that are well adapted for further assessment of the performance and robustness of different ANC configurations subjected to different kinds of threats. Indeed nets-within-nets allow hierarchical influences to be represented and the entity-resource hierarchy that we have shown paves the way for a multiple-level hierarchy with *e.g.* more detailed resources for a more precise assessment of threat impacts. The assessment metrics are directly linked to the semantics of tokens, that carry explicit knowledge of the ANC state.

Moreover reference nets, which are derived from object oriented programing, enable us to envisage quite easy manipulations as the whole structure of the model will not be affected by changes within nets, provided we manage to preserve interfaces with other nets.

Further work will focus on the improvement of the ANC model and on the setting up of more comprehensive simulation scenarios in order to assess several ANC configurations thoroughly, with various combinations of P/L and support satellites:

– Larger ANCs: more relevant ANC configurations, with more entities (up to 12 P/L satellites, 12 supports and 5 ground stations);
– Time: the nets need to be modified in order to make them suitable with the timed Petri net formalism which is a feature of *Renew*. Time will enable us to perform more realistic simulations and thus evaluate ANC performance more accurately;
– Real space dynamics: this will require the development of a plugin for *Renew* in order to fire some transitions according to real ephemerids read from external files;
– Knowledge bases: more complex data structures will be implemented, including knowledge on the environment or on the mission plan of the other entities;
– Communication protocols: different protocols will be implemented to manage communication session creations.

References

1. Barnhart, C., Ziemer, R.: Topological analysis of networks composed of multiple satellites. In: Tenth Annual International Phoenix Conference on Computers and Communications, US Naval Res. Lab., Washington, DC (1991)
2. Brown, O., Eremenko, P.: Fractionated space architectures: a vision for responsive space. In: 4th Responsive Space Conference. AIAA, Los Angeles (2006)
3. Cabac, L., Moldt, D., Rölke, H.: A Proposal for Structuring Petri Net-Based Agent Interaction Protocols. In: van der Aalst, W.M.P., Best, E. (eds.) ICATPN 2003. LNCS, vol. 2679, pp. 102–120. Springer, Heidelberg (2003)
4. Celaya, J.R., Desrochers, A.A., Graves, R.J.: Modeling and analysis of multi-agent systems using Petri nets. Journal of Computers 4(10), 981–996 (2009)

5. Cougnet, C., Gerber, B., Dufour, J.-F.: New technologies for improving satellite maintainability, flexibility and responsiveness. In: Toulouse Space Show 2010, TechnoDis Symposium, Toulouse, France (2010)
6. Cristini, F.: Robust satellite networks: solutions against emerging space threats. In: 18th IFAC Symposium on Automatic Control in Aerospace, Nara, Japan (2010)
7. Cristini, F., Tessier, C., Bensana, E.: Satellite network architectures against emerging space menaces. In: Toulouse Space Show 2010, TechnoDis Symposium, Toulouse, France (2010)
8. Cristini, F., Tessier, C., Bensana, E.: Satellite network architectures against emerging space threats. In: 10th International Symposium on Artificial Intelligence, Robotics and Automation in Space (iSAIRAS), Sapporo, Japan (2010)
9. Girault, C., Valk, R.: Petri Nets for System Engineering: A Guide to Modeling, Verification, and Applications. Springer-Verlag New York, Inc., Secaucus (2001)
10. Jaramillo, C. (ed.): Space Security Index 2011, Executive summary. Project Ploughshares and the McGill University Institute of Air and Space Law, Canada (2011)
11. Jensen, K., Kristensen, L.M.: Hierarchical coloured Petri nets. In: Coloured Petri Nets, pp. 95–125. Springer, Heidelberg (2009)
12. Köhler, M.: Mobile object net systems: Petri nets as active tokens. Technical Report 320, University of Hamburg, Department of Computer Science (2002)
13. Kummer, O.: Introduction to Petri Nets and Reference Nets. Sozionik Aktuell 1, 1–9 (2001)
14. Kummer, O., Wienberg, F., Duvigneau, M., Cabac, L.: Renew - User Guide. Release 2.2. Technical report, Theoretical Foundations Group, Departement of Informations, University of Hamburg (2009)
15. 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)
16. Laborde, B., Castel, C., Gabard, J.-F., Soumagne, R., Tessier, C.: FDIR strategies for autonomous satellite formations - A preliminary report. In: AAAI 2006 Fall Symposium "Space Autonomy: Using AI to Expand Human Space Exploration", Washington DC, USA (2006)
17. Lee, L., Nwana, H., Ndumu, D., Wilde, P.D.: The stability, scalability and performance of multi-agent systems. BT Technology Journal 16(3), 94–103 (1998)
18. Lomazova, I.A.: Nested Petri Nets: a formalism for specification and verification of multi-agent distributed systems. Fundamenta Informaticae 43, 195–214 (2000)
19. NASA. Glossary - NASA Crew Exploration Vehicle, SOL NNT05AA01J, Attachment J-6 (2005), http://www.spaceref.com/news/viewsr.html?pid=15201
20. Sánchez-Herrera, R., Villanueva-Paredes, N., López-Mellado, E.: High-Level Modelling of Cooperative Mobile Robot Systems. In: Alami, R., Chatila, R., Asama, H. (eds.) Distributed Autonomous Robotic Systems 6, pp. 431–440. Springer, Heidelberg (2007)
21. Valk, R.: Petri Nets as Token Objects: An Introduction to Elementary Object Nets. In: Desel, J., Silva, M. (eds.) ICATPN 1998. LNCS, vol. 1420, pp. 1–24. Springer, Heidelberg (1998)

P- and T-Systems in the Nets-within-Nets-Formalism

Frank Heitmann and Michael Köhler-Bußmeier

University of Hamburg, Department for Informatics
Vogt-Kölln-Straße 30, D-22527 Hamburg
{heitmann,koehler}@informatik.uni-hamburg.de

Abstract. We are concerned with (strongly deterministic) *generalised state machines* (GSMs), a restricted formalism of the nets-within-nets-family, and further restrict the involved nets to P- and T-nets. While GSMs with these restrictions are likely to be of little use in modelling applications, understanding them better might help in future attempts to analyse more sophisticated formalisms.

We show that, given a strongly deterministic GSM where the system net and all object nets are P-nets, it is PSpace-complete to decide the reachability of a given marking. In past work we have already shown that the same restriction to T-nets remains solvable in polynomial time. We discuss this work in the context given here. At last we give some initial results concerning other combinations of restricting the system and/or the object nets to P- and/or T-nets. Throughout we also discuss the effect of dropping the restriction to strongly deterministic GSMs.

Keywords: Higher-level net models, nets-within-nets, reachability problem.

1 Introduction

Many formalisms are known by now which in one way or the other apply the idea of nesting to Petri nets, that is, formalisms which allow to interpret the tokens of an ordinary p/t net as p/t nets again. Formalisms of these kind are for example object nets [28], recursive nets [8], nested nets [24], PN^2 [11], hypernets [1], Mobile Systems [23], AHO systems [12], adaptive workflow nets [25], and Hornets [20]. Another line of research also dealing with nesting, but not in the field of Petri nets, is concerned with process calculi. Arguably most prominently there are the Ambient Calculus of Gordon and Cardelli [2] and the Seal Calculus [3] among many others.

All these formalisms are usually quite helpful to model mobility of and interaction between different objects or agents. However, despite the success in modelling a variety of applications which are rather awkward to model with ordinary Petri nets to say the least, the borderline between modelling power on the one hand and complexity of the algorithms applied for verification issues on the other is by far not so well understood as for Petri nets where Free Choice

Petri nets can be seen as the formalism that allows to model a high diversity of applications while retaining a modest and manageable degree of complexity.

A rather constrained specimen of the above mentioned Petri net formalisms are generalised state machines (GSM). They are a restriction of elementary object systems (Eos), which were originally proposed by Valk [27] for a two levelled structure and later generalised for arbitrary nesting structures (see [17,18]). Despite being suited for modelling (see e.g. [16]), even with their restriction of the nesting depth they are Turing-powerful (cf. [15]). However, in most cases certain aspects of elementary net systems are not needed for modelling. Generalised state machines retain the ability to describe nesting of objects, but the duplication or destruction of them is not allowed. Therefore they are nicely suited to model physical entities. While problems like reachability and many others are now decidable (see Theorem 1) the above mentioned borderline is not well understood for GSMs, i.e. what is an appropriate net class for the involved nets such that we enjoy a high modelling capability, yet also have the possibility to verify the models with affordable resources?

To understand this boundary better, we first restrict GSMs to deterministic and strongly deterministic GSMs, limiting the interaction between the involved nets, and then restrict the nets to P- and T-nets. While such nets considered individually are well understood (cf. [4]), viewed in the context here they become far more intricate.

After presenting elementary object systems and generalised state machines in Section 2, we show in Section 3 that the reachability problem for strongly deterministic GSMs is PSpace-complete if all object nets and the system net are P-nets. We also repeat the formerly obtained result that the same problem is solvable in polynomial time if all object nets and the system net are T-nets. In Section 4 we give initial results for the "mixed cases", i.e. the cases where the system net is a T-net and the object nets are all P-nets or vice versa. The paper end with a conclusion and an outlook.

2 Fundamentals

An elementary object system (Eos) is composed of a system net, which is a p/t net $\widehat{N} = (\widehat{P}, \widehat{T}, \mathbf{pre}, \mathbf{post})$, and a set of object nets $\mathcal{N} = \{N_1, \ldots, N_n\}$, which are p/t nets given as $N_i = (P_{N_i}, T_{N_i}, \mathbf{pre}_{N_i}, \mathbf{post}_{N_i})$ (cf. Example 1 below). We assume $\widehat{N} \notin \mathcal{N}$ and the existence of the object net $N_\bullet \in \mathcal{N}$ which has no places or transitions and is used to model black tokens. Moreover we assume that all sets of nodes (places and transitions) are pairwise disjoint and set $P_\mathcal{N} = \cup_{N \in \mathcal{N}} P_N$ and $T_\mathcal{N} = \cup_{N \in \mathcal{N}} T_N$. The system net places are typed by the mapping $d : \widehat{P} \to \mathcal{N}$ with the meaning, that if $d(\widehat{p}) = N$, then the place $\widehat{p}$ of the system net may contain only net-tokens of the object net type N. The transitions in an Eos are labelled with synchronisation channels by the synchronisation labelling l. For this we assume a fixed set of channels C. In addition we allow the label τ which is used to describe that no synchronisation is desired (i.e. autonomous firing). The synchronisation labelling is then a tuple $l = (\widehat{l}, (l_N)_{N \in \mathcal{N}})$ where

$\widehat{l} : \widehat{T} \to (\mathcal{N} \to (C \cup \{\tau\}))$ and $l_N : T_N \to (C \cup \{\tau\})$ for all $N \in \mathcal{N}$. All these functions are total. The intended meaning is as follows: $l_N(t) = \tau$ means that the transition t of the object net N may fire (object-)autonomously. $l_N(t) = c \neq \tau$ means that t synchronises via the channel c with the system net. $\widehat{l}(\widehat{t})(N) = \tau$ means that the system net transition $\widehat{t}$ may fire independently (or autonomously) from the object net N. $\widehat{l}(\widehat{t})(N) = c \neq \tau$ means that $\widehat{t}$ synchronises via the channel c with the object net N. In case of a synchronous event the system net and the object net transitions have to be labelled with the same channel. A system net transition $\widehat{t}$ may fire system-autonomously, if $\widehat{l}(\widehat{t})(N) = \tau$ for all $N \in \mathcal{N}$.

A *marking* of an Eos is a *nested* multiset, denoted $\mu = \sum_{k=1}^{n} \widehat{p}_k[M_k]$, where $\widehat{p}_k$ is a place in the system net and M_k is a marking of the net-token of type $d(\widehat{p}_k)$. The set of all markings is denoted $\mathcal{M}$. We define the partial order $\leq$ on nested multisets by setting $\mu_1 \leq \mu_2$ iff $\exists \mu : \mu_2 = \mu_1 + \mu$.

$\Pi^1(\mu)$ denotes the projection of the nested marking μ to the system net level and $\Pi_N^2(\mu)$ denotes the projection to the marking belonging to the object net N, i.e. $\Pi^1(\sum_{k=1}^{n} \widehat{p}_k[M_k]) = \sum_{k=1}^{n} \widehat{p}_k$ and $\Pi_N^2(\sum_{k=1}^{n} \widehat{p}_k[M_k]) = \sum_{k=1}^{n} \mathbf{1}_N(\widehat{p}_k) \cdot M_k$, where $\mathbf{1}_N : \widehat{P} \to \{0, 1\}$ with $\mathbf{1}_N(\widehat{p}) = 1$ iff $d(\widehat{p}) = N$.

Definition 1 (Eos). *An elementary object system (*Eos*) is a tuple $OS = (\widehat{N}, \mathcal{N}, d, l)$ such that:*

1. *$\widehat{N}$ is a p/t net, called the* system net.
2. *$\mathcal{N}$ is a finite set of disjoint p/t nets, called* object nets.
3. *$d : \widehat{P} \to \mathcal{N}$ is the typing of the system net places.*
4. *$l = (\widehat{l}, (l_N)_{N \in \mathcal{N}})$ is the labelling.*

An Eos *with initial marking is a tuple $OS = (\widehat{N}, \mathcal{N}, d, l, \mu_0)$ where $\mu_0 \in \mathcal{M}$ is the initial marking.*

The synchronisation labelling generates the set of system events Θ, which consists of the disjoint sets of synchronous events Θ_l, object-autonomous events Θ_o, and system-autonomous events Θ_s. An event is a pair, denoted $\widehat{t}[\vartheta]$ in the following, where $\widehat{t}$ is a transition of the system net or $\widehat{\epsilon}$ if object-autonomous firing is desired and ϑ maps each object net to one of its transitions or to ϵ if no firing is desired in this object net, that is $\vartheta : \mathcal{N} \to T_{\mathcal{N}} \cup \{\epsilon\}$ where $\vartheta(N) \neq \epsilon$ implies $\vartheta(N) \in T_N$ for all $N \in \mathcal{N}$. If $\vartheta(N) = \epsilon$ for all N the system net transition fires autonomously. We also use the shortcut ϑ_ϵ for this function. The labelling functions are extended to $l_N(\epsilon) = \tau$ and $\widehat{l}(\widehat{\epsilon})(N) = \tau$ for all $N \in \mathcal{N}$.

We now distinguish three cases: For a synchronous event $\widehat{t}[\vartheta] \in \Theta_l$, the system net transition $\widehat{t} \neq \widehat{\epsilon}$, fires synchronously with all the object net transitions $\vartheta(N), N \in \mathcal{N}$. Thus at least one $N \in \mathcal{N}$ must exist with $\widehat{l}(\widehat{t})(N) \neq \tau$ and $\vartheta(N) \neq \epsilon$. We demand $\vartheta(N) \neq \epsilon \Leftrightarrow \widehat{l}(\widehat{t})(N) \neq \tau$ and that the channels have to match, i.e. $\widehat{l}(\widehat{t})(N) = l_N(\vartheta(N))$ for all $N \in \mathcal{N}$. Note that for object nets which do not participate in the event (either because they are not in the preset of the system net transition or because no object net transitions fires synchronously) $\widehat{l}(\widehat{t})(N) = \tau$ holds, which forces $\vartheta(N) = \epsilon$ and thus $l_N(\vartheta(N)) = l_N(\epsilon) = \tau = \widehat{l}(\widehat{t})(N)$.

In the case of a system-autonomous event $\widehat{t}[\vartheta] \in \Theta_s$, $\widehat{t} \neq \widehat{\epsilon}$ fires autonomously. Therefore we demand that $\widehat{l}(\widehat{t})(N) = \tau$ for all $N \in \mathcal{N}$ and $\vartheta = \vartheta_\epsilon$, that is $\vartheta(N) = \epsilon$ for all $N \in \mathcal{N}$.[1]

In the third case of an object-autonomous event $\widehat{\epsilon}[\vartheta] \in \Theta_o$, $\vartheta(N) \neq \epsilon$ for exactly one object net N. Moreover the transition $\vartheta(N)$ must not use a channel, that is $l_N(\vartheta(N)) = \tau$ has to hold.[2]

If we write $\widehat{t}[\vartheta] \in \Theta$ in the following, this includes the possibility that the event is an system- or object-autonomous event, i.e. $\vartheta = \vartheta_\epsilon$ or $\widehat{t} = \widehat{\epsilon}$ is possible. Moreover, since the sets of transitions are all disjoint, we usually write $\widehat{t}[\vartheta(N_1), \vartheta(N_2), \ldots]$ and also skip the object nets which are mapped to ϵ, that is, we simply list the object net's transitions with which a system net transition synchronises.

Example 1. Figure 1 below shows an Eos consisting of a system net $\widehat{N}$ and two object nets $\mathcal{N} = \{N, N'\}$. The typing of the system net is given by $d(\widehat{p}_1) = d(\widehat{p}_2) = d(\widehat{p}_4) = N$ and $d(\widehat{p}_3) = d(\widehat{p}_5) = d(\widehat{p}_6) = N'$.

For now, ignore the net-tokens on $\widehat{p}_4, \widehat{p}_5$, and $\widehat{p}_6$. These places are initially empty and the system has thus four net-tokens: two on place $\widehat{p}_1$ and one on $\widehat{p}_2$ and $\widehat{p}_3$ each. The net-tokens on $\widehat{p}_1$ and $\widehat{p}_2$ share the same structure, but have independent markings. The initial marking is thus given by

$$\mu = \widehat{p}_1[\mathbf{0}] + \widehat{p}_1[a+b] + \widehat{p}_2[a] + \widehat{p}_3[a'+b'].$$

We have two channels ch and ch'. The labelling function $\widehat{l}$ of the system net is defined by $\widehat{l}(\widehat{t})(N) = ch$ and $\widehat{l}(\widehat{t})(N') = ch'$. The object net's labellings are defined by $l_N(t) = ch$ and $l_{N'}(t') = ch'$. Thus there is only one (synchronous) event: $\Theta = \Theta_l = \{\widehat{t}[N \mapsto t, N' \mapsto t']\}$. The event is also written shortly as $\widehat{t}[t, t']$.

To explain firing we distinguish two cases: Firing a system-autonomous or synchronous event $\widehat{t}[\vartheta] \in \Theta_l \cup \Theta_s$ removes net-tokens together with their individual internal markings. The new net-tokens are placed according to the system net transition and the new internal markings are determined by the internal markings just removed and ϑ. Thus a nested multiset $\lambda \in \mathcal{M}$ that is part of the current marking μ, i.e. $\lambda \leq \mu$, is replaced by a nested multiset ρ.

The enabling condition is expressed by the *enabling predicate* ϕ_{OS} (or just ϕ whenever OS is clear from the context):

$$\begin{aligned} \phi(\widehat{t}[\vartheta], \lambda, \rho) \iff\; & \Pi^1(\lambda) = \mathbf{pre}(\widehat{t}) \wedge \Pi^1(\rho) = \mathbf{post}(\widehat{t}) \wedge \\ & \forall N \in \mathcal{N} : \Pi^2_N(\lambda) \geq \mathbf{pre}_N(\vartheta(N)) \wedge \\ & \forall N \in \mathcal{N} : \Pi^2_N(\rho) = \Pi^2_N(\lambda) - \mathbf{pre}_N(\vartheta(N)) + \mathbf{post}_N(\vartheta(N)), \end{aligned} \tag{1}$$

where $\mathbf{pre}_N(\epsilon) = \mathbf{post}_N(\epsilon) = \mathbf{0}$ for all $N \in \mathcal{N}$.

[1] Note that this implies $\vartheta(N) \neq \epsilon \Leftrightarrow \widehat{l}(\widehat{t})(N) \neq \tau$, the equivalence we had to demand in the case above. Moreover $l_N(\vartheta(N)) = \widehat{l}(\widehat{t})(N)$ follows, too.

[2] Note that the labels match again for all N, i.e. $\widehat{l}(\widehat{t})(N) = \widehat{l}(\widehat{\epsilon})(N) = \tau = l_N(\vartheta(N))$ for all $N \in \mathcal{N}$, but the equivalence $\vartheta(N) \neq \epsilon \Leftrightarrow \widehat{l}(\widehat{t})(N) \neq \tau$ does not hold for exactly one N, namely for the N for which $\vartheta(N) \neq \epsilon$ holds. $\vartheta(N) \in T_N$ is the transition intended to fire object-autonomously.

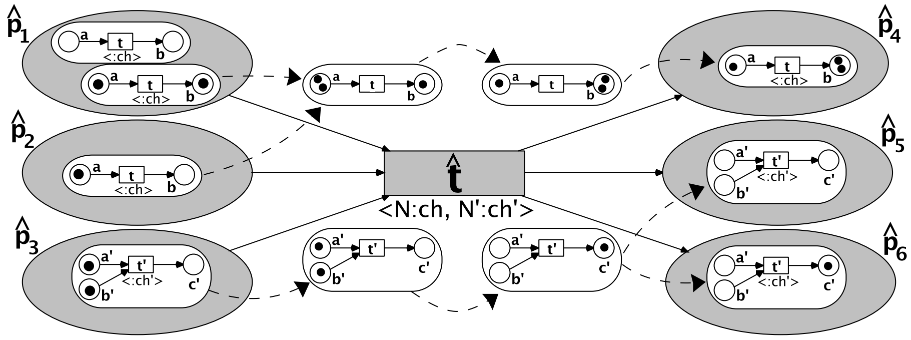


Fig. 1. An EOS firing the synchronous event $\widehat{t}[t, t']$

For an object-autonomous event $\widehat{\epsilon}[\vartheta] \in \Theta_o$ let N be the object net for which $\vartheta(N) \neq \epsilon$ holds. Now $\phi(\widehat{\epsilon}[\vartheta], \lambda, \rho)$ holds iff $\Pi^1(\lambda) = \Pi^1(\rho) = \widehat{p}$ for a $\widehat{p} \in \widehat{P}$ with $d(\widehat{p}) = N$ and $\Pi^2_N(\lambda) \geq \mathbf{pre}_N(\vartheta(N))$ and $\Pi^2_N(\rho) = \Pi^2_N(\lambda) - \mathbf{pre}_N(\vartheta(N)) + \mathbf{post}_N(\vartheta(N))$. In case of an object-autonomous event λ and ρ are thus essentially markings of an object net, but 'preceded' by a system net place typed with this object net.

Definition 2 (Firing Rule). *Let OS be an* EOS *and $\mu, \mu' \in \mathcal{M}$ markings. The event $\widehat{t}[\vartheta] \in \Theta$ is* enabled in μ for the mode $(\lambda, \rho) \in \mathcal{M}^2$ *iff $\lambda \leq \mu \wedge \phi(\widehat{t}[\vartheta], \lambda, \rho)$ holds.*

An event $\widehat{t}[\vartheta]$ that is enabled in μ for the mode (λ, ρ) can fire: $\mu \xrightarrow[OS]{\widehat{t}[\vartheta](\lambda,\rho)} \mu'$. The resulting successor marking is defined as $\mu' = \mu - \lambda + \rho$. Firing is extend to sequences $w \in (\Theta \cdot \mathcal{M}^2)^*$ in the usual way. The set of reachable markings from a marking μ is denoted by $RS_{OS}(\mu)$. The reachability problem asks given an EOS *OS with initial marking μ_0 and a marking μ, if $\mu \in RS_{OS}(\mu_0)$ holds.*

We omit the mode and the EOS *in the notations above if they are not relevant or clear from the context. We also say that $\widehat{t}[\vartheta]$ is* enabled in μ *or simply* active*, if a mode (λ, ρ) exists such that $\widehat{t}[\vartheta]$ is enabled in μ for (λ, ρ). This again is extended to sequences in the usual way.*

Example 2. To illustrate the firing rule, we return to the example of Figure 1. Note that the current marking μ enables $\widehat{t}[t, t']$ in the mode (λ, ρ), where

$$\begin{aligned}
\mu &= \widehat{p}_1[\mathbf{0}] + \widehat{p}_1[a + b] + \widehat{p}_2[a] + \widehat{p}_3[a' + b'] = \widehat{p}_1[\mathbf{0}] + \lambda \\
\lambda &= \widehat{p}_1[a + b] + \widehat{p}_2[a] + \widehat{p}_3[a' + b'] \\
\rho &= \widehat{p}_4[a + 2 \cdot b] + \widehat{p}_5[\mathbf{0}] + \widehat{p}_6[c']
\end{aligned}$$

The net-tokens' markings are added by the projections Π^2_N and $\Pi^2_{N'}$ resulting in the markings $\Pi^2_N(\lambda)$ and $\Pi^2_{N'}(\lambda)$. Firing the object net's transitions generates the (sub-)markings $\Pi^2_N(\rho)$ and $\Pi^2_{N'}(\rho)$. This is illustrated above and below transition $\widehat{t}$ in Figure 1, where the left net on top is $\Pi^2_N(\lambda)$ and the right net on top is $\Pi^2_N(\rho)$. Similar for the nets below $\widehat{t}$ for the object net N'. After the synchronisation we obtain the successor marking μ' with new net-tokens on $\widehat{p}_4$, $\widehat{p}_5$, and $\widehat{p}_6$:

$$\begin{aligned}\mu' &= (\mu - \lambda) + \rho = \widehat{p}_1[\mathbf{0}] + \rho \\ &= \widehat{p}_1[\mathbf{0}] + \widehat{p}_4[a + 2 \cdot b] + \widehat{p}_5[\mathbf{0}] + \widehat{p}_6[c']\end{aligned}$$

2.1 Generalised State Machines

A *generalised state machine* (GSM), first introduced in [19], is an Eos such that every system net transition has either exactly one place in its preset and one in its postset typed with the same object net or there are no such places. Additionally the initial marking has at most one net-token of each type.

Definition 3. *Let $G = (\widehat{N}, \mathcal{N}, d, l, \mu_0)$ be an* Eos. *G is a* generalised state machine *(GSM) iff G satisfies for all $N \in \mathcal{N} \setminus \{N_\bullet\}$*

1. $\forall \widehat{t} \in \widehat{T} : |\{\widehat{p} \in {}^\bullet\widehat{t} \mid d(\widehat{p}) = N\}| = |\{\widehat{p} \in \widehat{t}^\bullet \mid d(\widehat{p}) = N\}| \leq 1$
2. $\sum_{\widehat{p} \in \widehat{P}, d(\widehat{p}) = N} \Pi^1(\mu_0)(\widehat{p}) \leq 1$

Note that, the second item holds for all reachable markings, due to the first.

Since there is no restriction on $N_\bullet$, each p/t net is also a GSM. Moreover, for each GSM G a p/t net, the reference net $\mathrm{Rn}(G)$, can be easily constructed (see [19]). It is obtained by taking as set of places the disjoint union of all places of G and as set of transitions the events of G. Since the places of all nets in $\mathcal{N}$ are disjoint, given a marking μ of G the projections $(\Pi^1(\mu), (\Pi^2_N(\mu))_{N \in \mathcal{N}})$ can be identified with the multiset

$$\mathrm{Rn}(\mu) := \Pi^1(\mu) + \sum_{N \in \mathcal{N}} \Pi^2_N(\mu),$$

denoting the markings in the reference net.[3]

Definition 4. *Let $G = (\widehat{N}, \mathcal{N}, d, l, \mu_0)$ be a GSM. The* reference net, *denoted by* $\mathrm{Rn}(G)$, *is defined as the p/t net:*

$$\mathrm{Rn}(G) = \left(\left(\widehat{P} \cup \bigcup\nolimits_{N \in \mathcal{N}} P_N\right), \Theta, \mathbf{pre}^{\mathrm{Rn}}, \mathbf{post}^{\mathrm{Rn}}, \mathrm{Rn}(\mu_0)\right)$$

where $\mathbf{pre}^{\mathrm{Rn}}$ *and* $\mathbf{post}^{\mathrm{Rn}}$ *are defined for an event $\widehat{t}[\vartheta]$ by:*

$$\mathbf{pre}^{\mathrm{Rn}}(\widehat{t}[\vartheta]) = \mathbf{pre}(\widehat{t}) + \sum\nolimits_{N \in \mathcal{N}} \mathbf{pre}_N(\vartheta(N))$$
$$\mathbf{post}^{\mathrm{Rn}}(\widehat{t}[\vartheta]) = \mathbf{post}(\widehat{t}) + \sum\nolimits_{N \in \mathcal{N}} \mathbf{post}_N(\vartheta(N)),$$

with $\mathbf{pre}(\widehat{\epsilon}) = \mathbf{post}(\widehat{\epsilon}) = \mathbf{0}$ *and* $\mathbf{pre}_N(\epsilon) = \mathbf{post}_N(\epsilon) = \mathbf{0}$ *for all* $N \in \mathcal{N}$.

[3] The term *reference net* stems from an analogous definition for an Eos OS where $\mathrm{Rn}(OS)$ behaves as if the object nets (in OS) would have been accessed via pointers and not like values. Since in a GSM each object-net exists at most once, the difference between references and values does not truly exist (cf. [19]).

We repeat two easy to prove statements (cf. [15] and [19]) which allow to carry over results for p/t nets to generalised state machines:

Lemma 1. *Let G be a generalised state machine. An event $\widehat{t}[\vartheta]$ is activated in G for (λ, ρ) iff it is in* $\textrm{R}{\small \textrm{N}}(G)$*:*

$$\mu \xrightarrow[G]{\widehat{t}[\vartheta](\lambda,\rho)} \mu' \quad \iff \quad \textrm{R}{\small \textrm{N}}(\mu) \xrightarrow[\textrm{R}{\small \textrm{N}}(G)]{\widehat{t}[\vartheta]} \textrm{R}{\small \textrm{N}}(\mu')$$

Theorem 1. *The reachability problem is decidable for GSMs.*

In the definition of the reference net the set of events is present. Unfortunately, given a GSM $G = (\widehat{N}, \mathcal{N}, d, l, \mu_0)$ the number of events may become huge. To see this, let T_i be the set of transitions of the object net N_i. Let $\widehat{l}(\widehat{t})(N_i) = c_i$ for each i and a fixed system net transition $\widehat{t}$, where the c_i are channels. Let $l_{N_i}(t) = c_i$ for all $t \in T_i$ and all i. Now $\widehat{t}$ may fire synchronously with each transition in N_1, each in N_2 and so on. Each of these possibilities results in a different event, so we already have at least $|T_1| \cdot |T_2| \cdot \ldots \cdot |T_n|$ events, a number exponential in the number of object nets and thus in the size of the GSM. Note that this is possible for each system net transition resulting in an even larger number of events.

Lemma 2. *Let $|T| := \max\{|T_N| \mid N \in \mathcal{N}\}$ then the space needed to store Θ is in $O(|\widehat{T}| \cdot |T|^{|\mathcal{N}|} \cdot e)$, where e is the space needed to store one event $\widehat{t}[\vartheta]$, that is $e = O(|\mathcal{N}|)$.*

Although the space to store Θ is exponential in $|\mathcal{N}|$, the space needed to store a single event is small. Since it is easy to check with a couple of table lookups, if a given input $\widehat{\tau}[\vartheta]$ is indeed an event, that is if $\widehat{\tau}[\vartheta] \in \Theta$ holds, it is thus possible to enumerate all elements of Θ one after another and only use a small amount of space. Furthermore the following proof shows that it is possible to compute, given a marking μ of a GSM (or an Eos), all immediate successors of μ.

Lemma 3. *Given an* Eos *OS and a marking μ of OS, it is possible to compute all immediate successor markings of μ.*

Proof. We enumerate all events and for each event $\widehat{\tau}[\vartheta] \in \Theta_l$ we enumerate all modes (λ, ρ) such that $\lambda \leq \mu$ holds and ρ is in accordance with the conditions in equation 1, that is ρ satisfies $\Pi^1(\rho) = \mathbf{post}(\widehat{\tau})$ and $\Pi^2_N(\rho) = \Pi^2_N(\lambda) - \mathbf{pre}_N(\vartheta(N)) + \mathbf{post}_N(\vartheta(N))$ for all $N \in \mathcal{N}$.[4] For each so chosen event $\widehat{\tau}[\vartheta]$ and mode (λ, ρ) we check if $\widehat{\tau}[\vartheta]$ is indeed enabled in μ for this mode and if so compute the successor marking. □

Note that Lemma 3 also implies that it is possible to compute all possible successor markings given a marking and an event, to test if a given event is active

[4] Note that for one λ several modes (λ, ρ) may exist, that is the markings of the object nets may be distributed in different ways in the successor marking of μ.

in a given marking, to test if a marking is a deadlock, and given two markings μ and μ' to check if an event exists that is active in μ and whose firing results in μ'.

Also note that the algorithm in the proof of Lemma 3 is not very efficient. Nonetheless due to the different distributions of the object nets markings that are usually possible, one in general has to deal with exponentially many successor markings in the size of the Eos.

Turning back to the reference net of a GSM G, according to Lemma 2 it might be very expensive to construct $\textsc{Rn}(G)$. Note that this is due to the nondeterminism introduced above by the labelling. All transitions of one object net are labelled with the same channel, so one of the transitions is chosen nondeterministically to fire synchronously with $\widehat{t}$. To prevent this, we introduced deterministic GSMs in [9]:

Definition 5. *A GSM G is called* deterministic *if for each $N \in \mathcal{N}$ the value of $l_N(t)$ differs for all $t \in T_N$ with $l_N(t) \neq \tau$. G is* strongly deterministic *if in addition for all $\widehat{t}$ and N with $\widehat{l}(\widehat{t})(N) \neq \tau$ the value of $\widehat{l}(\widehat{t})(N)$ differs.*

Thus, in a deterministic GSM each channel is used at most once in each object net. In a strongly deterministic GSM each channel is also used at most once in the system net.

On the one hand GSMs can be used to model mobility and communication of processes or agents - even if only to a smaller degree compared to Eos or general object nets. On the other hand a GSM G can be 'flatten' to the reference net $\textsc{Rn}(G)$, which is a p/t net and which thus makes it possible to use analysis techniques for p/t nets. Since GSMs are thus as powerful as p/t nets, it is interesting to investigate if certain restrictions known for p/t nets can be transferred to GSMs and if they retain their complexity. As stated in the introduction we will focus on P- and T-nets in the following.

Definition 6. *Let $N = (P, T, F)$ be a p/t net and $G = (\widehat{N}, \mathcal{N}, d, l, \mu_0)$ be a GSM.*

1. *If $|{}^\bullet t| = |t^\bullet| = 1$ holds for every transition $t \in T$, then N is a* P-net.
2. *If $|{}^\bullet p| = |p^\bullet| = 1$ holds for every place $p \in P$, then N is a* T-net.
3. *If $\widehat{N}$ is a P-net and all $N \in \mathcal{N}$ are P-nets, then G is a* ppGSM.
4. *If $\widehat{N}$ is a T-net and all $N \in \mathcal{N}$ are T-nets, then G is a* ttGSM.
5. *If $\widehat{N}$ is a P-net and all $N \in \mathcal{N}$ are T-nets, then G is a* ptGSM.
6. *If $\widehat{N}$ is a T-net and all $N \in \mathcal{N}$ are P-nets, then G is a* tpGSM.

For historical reasons P-nets are also called S-nets and T-nets are also called marked graphs.[5]

From the definition of a GSM and the restrictions imposed above, one can deduce that for ppGSMs and ptGSMs there can only be one object net:

[5] Sometimes a distinction is made between a P-net N and a *P-system* (N, m_0), where additionally the initial marking m_0 is given (analogous for T-nets). We do not use this distinction here and also do not introduce it for GSMs.

Lemma 4. *If $G = (\widehat{N}, \mathcal{N}, d, l, \mu_0)$ is a ppGSM or a ptGSM and the system net $\widehat{N}$ is connected*[6]*, then $|d(\widehat{P})| = 1$.*

Proof. Assume otherwise and let D be the set of tuples of system net places such that $(\widehat{p}, \widehat{p}') \in D$ iff $d(\widehat{p}) \neq d(\widehat{p}')$. Let $dist(\widehat{p}, \widehat{p}')$ be the distance of two system net places with respect to the relation $(\widehat{F} \cup \widehat{F}^{-1})^*$. Choose $(\widehat{p}_1, \widehat{p}_2) \in D$ such that $dist(\widehat{p}_1, \widehat{p}_2)$ is minimal.

Now, since $\widehat{N}$ is connected and since $dist(\widehat{p}_1, \widehat{p}_2)$ is minimal, there is a node x such that a directed path (possibly of length 0) with respect to the relation $\widehat{F}$ exists from $\widehat{p}_1$ to x and also from $\widehat{p}_2$ to x. (If no such x exists, there is a node x' on the path between $\widehat{p}_1$ and $\widehat{p}_2$ such that $(\widehat{p}_1, x') \in D$ or $(\widehat{p}_2, x') \in D$ holds and such that the distance is smaller than $dist(\widehat{p}_1, \widehat{p}_2)$, contradicting the choice of $\widehat{p}_1$ and $\widehat{p}_2$.) Since G is a ppGSM or a ptGSM, $|{}^\bullet\widehat{t}| = |\widehat{t}^\bullet| = 1$ holds for all $\widehat{t} \in \widehat{T}$ and thus x cannot be a transition and must be a place. But now, since every transition of the system net has exactly one place in its preset and one in its postset and since these must then be of the same type according to the definition of a GSM (item 1 in Definition 3), $d(x)$ must equal $d(\widehat{p}_1)$ due to the path from $\widehat{p}_1$ to x but must also equal $d(\widehat{p}_2)$ due to the path from $\widehat{p}_2$ to x, contradicting $d(\widehat{p}_1) \neq d(\widehat{p}_2)$. □

We assume that $\mathcal{N} = d(\widehat{P})$ in the following, since a $N \in \mathcal{N} \setminus d(\widehat{P})$ is never used. The above lemma thus says that in a connected ppGSM or ptGSM we always have $|\mathcal{N}| = 1$. Furthermore, we may usually assume connectedness in the algorithms below, because if a GSM is not connected the single parts do not effect each other and can be treated in isolation. Since in the case where $\mathcal{N} = \{N_\bullet\}$ holds, a ppGSM or ptGSM is simply a P-Net and thus the theory for P-Nets is applicable, we assume in the following that $\mathcal{N} \neq \{N_\bullet\}$ for GSMs.[7] Also note that if $|\mathcal{N}| = 1$ holds for a GSM G (and $\mu_0 \neq \mathbf{0}$), then G has exactly one object net all the time, because the initial marking has only one net-token of each type (see the second item in the definition of a GSM, Definition 3).

To sum up, we assume in the following that a given GSM G is connected, that $\mathcal{N} = d(\widehat{P})$, and that $\mathcal{N} \neq \{N_\bullet\}$ holds.

In the following we will focus on the complexity of the reachability problem for strongly deterministic GSMs which additionally fulfil one of the items 3 to 6 in Definition 6 above.

3 The Reachability Problem for ppGSMs and ttGSMs

While for P- and T-nets the reachability problem can be solved in polynomial time [4], the problem becomes more intricate in the context of ppGSMs or ttGSMs. It turns out that for ttGSMs reachability can still be decided in polynomial time, but for ppGSMs the problem becomes PSpace-complete. We

[6] A net (P, T, F) is *connected* if every two nodes x, y satisfy $(x, y) \in (F \cup F^{-1})^*$.

[7] ttGSMs and tpGSMs would be the same as T-Nets in this case and a GSM would be a standard p/t net, it is thus reasonable to exclude this case for all GSMs.

start with showing PSPACE-hardness for the last-mentioned problem and then continue to show three possibilities to decide the problem in polynomial space. We used one of these approaches in the past to prove the first mentioned result, i.e. that for ttGSMs reachability of a given marking can be decided in polynomial time (see [10]), a result we briefly discuss at the end of this section.

Lemma 5. *Given a strongly deterministic ppGSM $G = (\widehat{N}, \mathcal{N}, d, l, \mu_0)$ and a marking μ, it is* PSPACE*-hard to decide if μ is reachable from μ_0.*

Proof. We give a reduction from the problem to decide if, given a linear bounded automaton A and an input string x, whether A accepts x or not. This problem is PSPACE-complete [14], [7].

Let $A = (Q, \Sigma, \Gamma, K, q_0, F, \#)$ be a linear bounded automata where Z is the finite set of states, Σ the finite set of input symbols, $\Gamma \supseteq \Sigma \cup \{\#\}$ the finite set of tape symbols, $K \subseteq Q \times \Gamma \times \{L, R, H\} \times Q \times \Gamma$ the transition relation, q_0 is the initial state, F the set of final states and $\#$ the blank symbol. Without loss of generality we assume that A uses only the portion of the tape containing the input (due to the linear tape-compression theorem) and that if A accepts a word it clears the tape, moves the head to the leftmost cell and enters a unique final state (i.e. we also assume $|F| = 1$).

Given a LBA A and an input string $x \in \Sigma^*$ we now construct in polynomial time a strongly deterministic ppGSM G and a marking μ such that μ is reachable in G iff A accepts x.

The idea is to have an event θ for every transition $k \in K$ *and* every tape cell on which this transition k might occur. Tokens on places are used to memorize in which state A is, on which cell the read-/write-head resides and what tape symbol is written on each cell. Since a single transition with only one input and one output arc cannot do all these changes the information is partly stored in the system and partly stored in the object net. The system net will save in which state A is and on which cell the read-/write head currently resides. The object net will save for each cell used what tape symbol is written on it (see Figure 2).

Let $Q = \{q_0, \ldots, q_{n-1}\}$, where q_0 is the start and q_1 the final state, $K = \{k_1, \ldots, k_m\}$, $\Gamma = \{A_1, \ldots, A_s\}$, where $A_1 = \#$, $|x| = l$ the input string's length and $c_1, \ldots, c_l$ the tape cells used.

The system net consists of $n \cdot l$ places $\widehat{P} = \{\widehat{p}_{i,j} \mid 0 \leq i \leq n-1, 1 \leq j \leq l\}$, where $p_{i,j}$ is marked if A is in state q_i and the read-/write-head is on cell c_j.

The (single) object net consists of $s \cdot l$ places $P = \{p_{i,j} \mid 1 \leq i \leq s, 1 \leq j \leq l\}$, where $p_{i,j}$ is marked if the symbol A_i is written on the cell c_j.

Furthermore, for each cell c_j and each transition $k_i \in K$ of the LBA there is a system net transition $\widehat{t}_{i,j}$ and an object net transition $t_{i,j}$ which are labelled with the same channel $c_{i,j}$.[8] Let $k_i = (q_a, A_b, X, q_{a'}, A_{b'})$. The input arc of $\widehat{t}_{i,j}$ is from $\widehat{p}_{a,j}$, the place that encodes that A is in state q_a and the read-/write-head is on cell c_j. The input arc of $t_{i,j}$ is from $p_{b,j}$, the place that encodes that the cell c_j is currently marked with the tape symbol A_b. The output arc of

[8] Aside from the cases where the read-/write-head would move from the tape. This is discussed below.

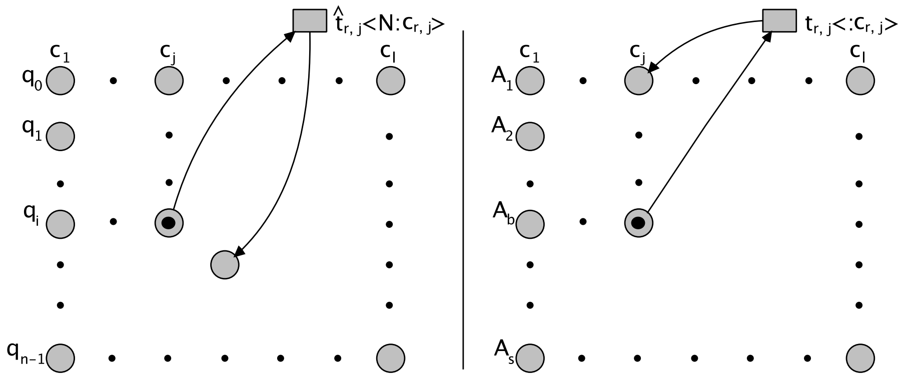


Fig. 2. System net (left) and object net (right) of the ppGSM from Lemma 5

$t_{i,j}$ is $p_{b',j}$. At last, the output arc of the system net transition depends on X, the movement of the LBA's head. If $X = H$ then the output arc of $\widehat{t_{i,j}}$ is to $p_{a',j}$. If $X = L$ then the output arc is to $p_{a',j-1}$ and if $X = R$ the output arc is to $p_{a',j+1}$. In the cases where $X = L$ and $j = 0$ or $X = R$ and $j = l$, i.e. in the cases where the LBA's head would move off the tape no transitions exist. The situation in Figure 2 is therefore as follows: The LBA is in state q_i, reading cell c_j onto which currently symbol A_b is written. The content of the other cells is not shown in the figure. The pictured transitions - synchronised via the channel $c_{r,j}$ - correspond to a transition $k_r = (q_i, A_b, R, q_{i+1}, A_1)$ of A.

Since we have only one object net, all places of the system net are typed with this net.

For the initial marking μ_0 let $x = A_{i_1} A_{i_2} \ldots A_{i_l}$ be the input of length l. Now μ_0 is given by $\widehat{p}_{0,1}[p_{i_1,1} + p_{i_2,2} + \ldots + p_{i_l,l}]$.

The marking tested for reachability is given by $\mu_e := \widehat{p}_{1,1}[p_{1,1}+p_{1,2}+\ldots+p_{1,l}]$.

By construction exactly one system net place is marked in each reachable marking. Also from the object net's places $p_{1,i}, p_{2,i}, \ldots, p_{s,i}$ exactly one place is marked (and l places are marked altogether in the object net).

Now, if A accepts the input x then there is a finite sequence $C_1, C_2, \ldots$ of configurations such that for each C_i there is a transition $k_i \in K$ that is possible in C_i and that changes the configuration of A to C_{i+1}. It is easy to prove inductively that this sequence of configurations corresponds to a sequence of markings in the constructed net system and that the transition $k_i \in K$ correspond to exactly one event consisting of one system net and one object net transition. Firing this event yields the marking corresponding to the next configuration in the sequence. If A accepts x then the last configuration is the unique accepting configuration and so the reached marking in the net system is μ_e.

Conversely if μ_e is reachable in G then by construction the sequence of markings correspond to a sequence of configurations of A and each transition in G corresponds to a transition in A. (A more rigorous proof would again use induction.) Thus if μ_e is reachable in G, so is the accepting configuration in A and hence μ_e is reachable in G iff A accepts x.

Since the constructed net system is clearly a strongly deterministic ppGSM and the construction can be done in polynomial time[9] (actually even in logarithmic space on the work tape), we conclude that the reachability problem for ppGSMs is PSpace-hard. $\square$

The negative result above carries over to deterministic ppGSMs and general ppGSMs, thus the reachability problem for all of them is at least PSpace-hard. We now prove that PSpace is enough.

In our first approach to prove that the reachability problem for ppGSMs can be decided in polynomial space we use a technique that dates back to Savitch's proof of PSpace = NPSpace [26] and that was in particular used successfully in the proofs that CTL model checking of 1-safe p/t nets and also of safe Eos is in PSpace [5], [21], [22]. The technique is based on the following idea: Given a finite state space of size $2^{p(n)}$ where p is a polynomial and n the input size, to decide if a given state s is reachable from the start state s_0 in m steps a deterministic machine iterates through all other states s' and tests if s' is reachable from s_0 and s from s' by $m/2$ steps, i.e. by half the steps. These tests are done recursively applying the same technique and reusing space. Since the number of steps is halved each time and at most $2^{p(n)}$ steps are possible without entering a loop, only $\log 2^{p(n)} = p(n)$ states need to be stored on a stack and thus the space needed is polynomial.

This technique works here, too, even if the state space might be rather big.

Lemma 6. *Given a strongly deterministic ppGSM $G = (\widehat{N}, \mathcal{N}, d, l, \mu_0)$ and a marking μ, it is decidable in polynomial space if μ is reachable from μ_0.*

Proof. Let $N = (P, T, F)$ be the sole object net (see Lemma 4). If $\mu_0 = \mathbf{0}$ then the only reachable marking is $\mathbf{0}$, since $\widehat{N}$ is a P-Net and thus no transition is active. We thus assume that $\mu_0 \neq \mathbf{0}$. In this case N is not only the sole object net type, but actually there is only one object net in any reachable marking μ due to the second item in Definition 3, i.e. $|\Pi^1(\mu)| = 1$.

Let $|\Pi^2_N(\mu_0)| = m$, i.e. in the initial marking the object net is marked with m black tokens. Since N is a P-net, the number of tokens in N remains constant and thus the number of reachable markings of N is bounded by $(m+1)^{|P|}$.[10] Since N can move around in the system net, but only one place of the system net is marked at any reachable marking, we have a upper bound of $|\widehat{P}| \cdot (m+1)^{|P|}$ for the number of reachable markings of G. Let n be the size of the input. From

[9] Note that with $s := \max\{|Q|, |\Gamma|, |K|, |x|\}$ we have $|\widehat{P}|, |P|, |\widehat{T}|, |T| \leq s^2$ and $|\widehat{F}|, |F| \leq 2s^2$. The labelling assigns to each transition one channel and can thus be stored in $\mathcal{O}(s^2)$ space. Also μ_0 and μ_e can be stored in $\mathcal{O}(s)$ space. Altogether the output is in $\mathcal{O}(s^2)$ space and since only a constant amount of computation is necessary for each output bit, the whole computation is possible in $\mathcal{O}(s^2)$ time.

[10] On each place between 0 and m tokens can reside, resulting in the above bound, which could actually be strengthen, but is sufficient here.

$n > \log(m+1), |P|, |\widehat{P}|$ and $|\widehat{P}| \cdot (m+1)^{|P|} = 2^{\log|\widehat{P}|} \cdot 2^{\log(m+1)\cdot|P|} \leq 2^{n^2+n}$ it follows that we have a finite state space of size $2^{p(n)}$ where p is a polynomial and n the input size and thus the technique outlined above is applicable.

A NPSpace-Algorithm A works as follows. Given G, μ_0 and μ we first guess the number i of steps it takes to reach μ. Since we have an upper bound for the size of the state space, i is known to be between 0 and 2^{n^2+n}.[11]

Now A guesses a marking μ' and verifies recursively that μ' is reachable from μ_0 and μ from μ' with $j = \lceil i/2 \rceil$ resp. $j = \lfloor i/2 \rfloor$ steps. The recursion ends if j is 0 or 1. In the first case two markings have to be tested for equality. In the second case we have to test if one marking is reachable from the other in one step. For this we can iterate through all events, consistently reusing space, test if the event is active and, if so, test if it has the desired effect (see Lemma 3). A accepts if it reaches μ and rejects if it does not reach μ in i steps.

Since the nesting depth of the recursive calls is $\log i$ it is at most $\log 2^{n^2+n} = n^2+n$ and thus polynomial in the input size. Furthermore, since only a finite number of data items (e.g. markings, counters) need to be stored and all these only require polynomial space and since it is possible to test in polynomial space if a marking is reachable from another marking and also if a marking is identical to another, i.e. all subroutines only require polynomial space, polynomial space is sufficient for the whole computation. □

Alternatively, one can exploit the fact that the total number of tokens does not change in a ppGSM. This idea was also used in [13] to prove that the reachability problem can be decided in polynomial space for 1-conservative p/t nets (i.e. nets with $|{}^\bullet t| = |t^\bullet|$ for all t). We only sketch the proof here.

Alternative Proof of Lemma 6 (Sketch). Since $\widehat{N}$ and the sole object net N are both P-nets, the total number of tokens does not change. In a nondeterministic algorithm A one can thus maintain one counter for each place, where the size of each counter is bounded. By guessing a firing sequence A can thus solve the reachability problem again exploiting the fact that the state space is finite and the firing sequence has a length representable in polynomial space. The whole algorithm works in polynomial space. □

Both approaches presented above are working directly on a given GSM G. In the next approach we make use of the reference net $\textsc{Rn}(G)$ instead (see Definition 4), which allows us to use tools already available for p/t nets.

Lemma 7. *Let $G = (\widehat{N}, \mathcal{N}, d, l, \mu_0)$ be a strongly deterministic ppGSM. Then the p/t net* $\textsc{Rn}(G)$ *satisfies $|{}^\bullet t| = |t^\bullet| = 1$ or $|{}^\bullet t| = |t^\bullet| = 2$ for all $t \in T(\textsc{Rn}(G))$.*

Proof. Let $t \in T(\textsc{Rn}(G))$ and let N be the unique object net (see Lemma 4).[12] We want to show $|\mathbf{pre}^{\textsc{Rn}}(t)| = |\mathbf{post}^{\textsc{Rn}}(t)| \in \{1, 2\}$. Since the transitions of

[11] Alternatively i can be a loop index, reusing space in each iteration.

[12] Note that N is not only the sole object net type present but that actually only one object net exists in any reachable marking (if $\mu_0 \neq \mathbf{0}$; but otherwise $\mathbf{0}$ is the only reachable marking), due to the second item in Definition 3.

$\textsc{Rn}(G)$ are the events of G we distinguish the three cases $t \in \Theta_s$ (system-autonomous event), $t \in \Theta_o$ (object-autonomous event), and $t \in \Theta_l$ (synchronous event).

The idea is that in the first two cases of autonomous events the pre- and postset of the event are identical to the pre- resp. postset (of cardinality 1) of a single system or object net transition and that in the case of a synchronous event the event is just a combination of one system net and one object net transition so that the cardinality of the pre- and postset is 2.

Explicitly, we have $t = \widehat{t}[\vartheta_\epsilon]$ for some system net transition $\widehat{t}$ in the first case. Since $\vartheta_\epsilon(N) = \epsilon$, the preset of t and $\widehat{t}$ are identical according to Definition 4 as are the postsets. Thus we have $|\mathbf{pre}^{\textsc{Rn}}(t)| = |\mathbf{pre}(\widehat{t})| = 1 = |\mathbf{post}(\widehat{t})| = |\mathbf{post}^{\textsc{Rn}}(t)|$.

The case of an object-autonomous event is similar. We then have $t = \widehat{\epsilon}[\vartheta]$ with $\vartheta(N) \neq \epsilon$ (usually for exactly one object net, but we only have one object net here, since G is a ppGSM). Thus again according to Definition 4 we have that the preset of t and $\vartheta(N)$ are the same as are the postsets and their cardinality is one again.

In the third case of a synchronous event one system net transition $\widehat{t}$ fires synchronously with exactly one object net transition t_N. With Definition 4 and since $\mathbf{pre}(\widehat{t}) \neq \mathbf{pre}_N(t_N)$ and $\mathbf{post}(\widehat{t}) \neq \mathbf{post}_N(t_N)$, we have $|\mathbf{pre}^{\textsc{Rn}}(t)| = |\mathbf{post}^{\textsc{Rn}}(t)| = 2$ (where in the preset of t are exactly the two places in the preset of $\widehat{t}$ and t_N and analogously for the postset). □

A p/t net with the property $|{}^\bullet t| = |t^\bullet|$ for each transition t is called 1-conservative and it is shown in [13] that for these net class the reachability problem can be decided in polynomial space. Thus, since the above conversion is clearly possible in polynomial space, we again have that reachability is decidable in polynomial space.

Note that we did not use the fact that G was a strongly deterministic GSM in any of the above proofs and actually the proofs work all the same for deterministic ppGSMs and also for general ppGSMs. Thus we have the following corollary from Lemma 6 or Lemma 7 above:

Corollary 1. *The reachability problem for strongly deterministic ppGSMs, deterministic ppGSMs and ppGSMs is solvable in polynomial space.*

The main reason for the generalisation in Corollary 1 is that the exponential blow-up of events (see Lemma 2) cannot occur in a ppGSM – due to Lemma 4 there is only one object net and thus we have at most a quadratic number of events in the size of the input.

From Corollary 1 and from Lemma 5 we deduce the following theorem:

Theorem 2. *The reachability problem for strongly deterministic ppGSMs, deterministic ppGSMs and ppGSMs is* PSPACE*-complete.*

The ttGSM Case. In a ttGSM the system net and all object nets are T-nets. Unlike before in the case of ppGSMs, a variety of object nets might now be present and according to Lemma 2 the number of events might become huge.

Thus for general ttGSMs the approach from Lemma 7 is unlikely to be applicable. But for strongly deterministic ttGSMs the approach works. The proof of the following lemma is similar in nature to the proof of Lemma 7: it is not complicated, but rather tedious. It can be found in [10].

Lemma 8. *Let* $G = (\widehat{N}, \mathcal{N}, d, l, \mu_0)$ *be a strongly deterministic ttGSM. Then* $\textsc{Rn}(G)$ *is a T-net.*

Since in a (strongly) deterministic GSM the number of events is bounded by a polynomial[13] the reference net can be constructed in polynomial time. Since reachability can be decided in polynomial time for a marked graph [6], we have:

Theorem 3. *The reachability problem for strongly deterministic ttGSMs is solvable in polynomial time.*

It is interesting to note that for a strongly deterministic ttGSM G the reference net $\textsc{Rn}(G)$ is a T-net, while for deterministic ttGSMs and general ttGSMs this is in general not the case. So, not only the size of $\textsc{Rn}(G)$ becomes huge in these cases, also its structure becomes more intricate. This was actually the very reason why we introduced deterministic and strongly deterministic GSMs in [10] at all.

On the other hand, even if $\textsc{Rn}(G)$ is not a P-net for none of the ppGSM formalisms, due to Lemma 7 and the discussion following it, the reference net is always 1-conservative and thus reachability can be solved in polynomial space. One of the reasons might be that in a ppGSM G only one object net is present and thus the number of events is rather limited independently of G being strongly deterministic, deterministic or a general ppGSM.

The more sophisticated structure of $\textsc{Rn}(G)$ in the case of deterministic and general ttGSMs, the increasing number of events and the potential presence of several object nets, are the reasons why ttGSMs are not as well understood yet as ppGSMs.

4 The Reachability Problem for ptGSMs and tpGSMs

We now turn our attention to ptGSMs and tpGSMs, i.e. to GSMs which employ a mixture of P- and T-nets. While in the cases of ttGSM and ppGSM above we got some positive results in way of algorithms, for ptGSMs and tpGSMs we up to now only have negative results, i.e. hardness results, or no results at all. For ptGSMs we already showed in [10] that reachability is NP-hard if the GSM is deterministic (a result that carries over to general ptGSMs). Here we strengthen (or rather worsen) this result to PSPACE-hardness. Furthermore we show that for strongly deterministic ptGSMs the problem is NP-hard. For tpGSMs we conjecture that the problem is solvable in polynomial time and give some intuition why this might be so.

[13] There are at most $|\widehat{T}|$ system- and $\sum_{N \in \mathcal{N}} |T_N|$ object-autonomous events. The sum of these is an upper bound for the number of events, because a synchronous event can be thought of here as a combination of one system- and several object-autonomous events, reducing the total number of events below the aforementioned sum.

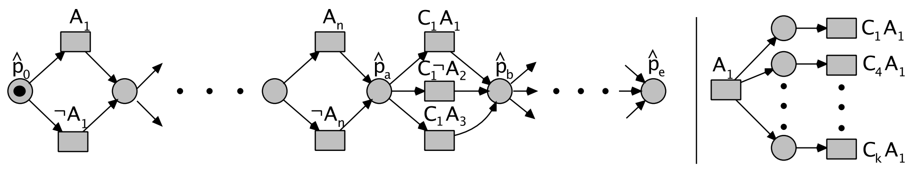


Fig. 3. System net (left) and object net (right) of the ptGSM from Theorem 4

Theorem 4. *It is* NP*-hard to decide the reachability problem for strongly deterministic ptGSMs.*

Proof. We give a reduction from 3-SAT. The ptGSM we construct is outlined in Figure 3, where only the transitions' channels are given. Assume we are given an instance of 3-SAT, i.e. a formula F with n variables $A_1, \ldots, A_n$ and m clauses $C_1, \ldots, C_m$, where each clause consists of exactly three literals.

The idea is that the object net moves through the system net and with its first n steps sets each variable to true or false. The next m moves check if in each clause one literal is true with this setting. All m moves are only possible if the setting is a satisfying assignment.

In more detail: For each variable A a pair of transitions exists in the system net (see Figure 3), one for the positive literal A and one for the negative literal $\neg A$. The transitions use the channels A and $\neg A$ respectively. To describe the object net N assume that L is a literal (e.g. A_1 or $\neg A_n$) that appears in the clauses $C_{i_1}, \ldots, C_{i_k}$. Then N has one transition t_L using channel L and with empty preset and k outgoing arcs. Each outgoing arc is connected with a place which is connected to a transition using a channel $C_{i_j} L$ corresponding to the clause. In Figure 3 (right side) this is only depicted for the positive literal A_1. Note that such a construct exists for each literal and that all used channels differ.

Further system net transitions exists to encode the clauses. Let $L_{i,1}, L_{i,2}, L_{i,3}$ be the literals that occur in clause C_i. Then three concurrent transitions exists in the system net using the channels $C_i L_{i,1}$ to $C_i L_{i,3}$ as depicted in Figure 3 for the clause C_1 (assumed to be $A_1 \vee \neg A_2 \vee A_3$).

The initial marking is $\mu_0 := \widehat{p}_0[\mathbf{0}]$.

It is easy to see that the object net can reach the place $\widehat{p}_e$ (to the right of the system net) iff the given formula is satisfiable. Unfortunately, this only shows that the marking $\mu_e := \widehat{p}_e[\mathbf{0}]$ can be *covered*.[14]

To get rid of the tokens in the object net a more sophisticated gadget needs to be constructed. The gadget is depicted in Figure 4. It replaces the part between the places $\widehat{p}_a$ and $\widehat{p}_b$ in Figure 3 and an identical structure only using different channels needs to be added for the other clauses. The gadget has the following property: At least one of the transitions with channel inscriptions needs to fire to reach $\widehat{p}_b$ and apart from the empty set every subset of these transitions is possible. (The unlabelled transitions fire system-autonomously.)

[14] Thus the coverability problem is NP-hard for ptGSMs.

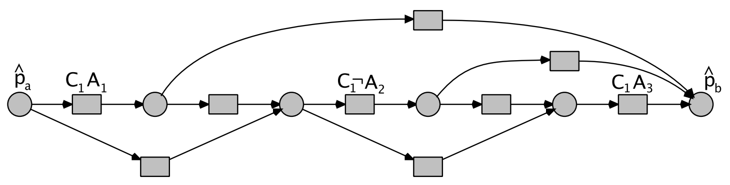


Fig. 4. Modification of the system net of Figure 3

With this gadget it is possible to fully rid the object net of all tokens. If for example the transition with channel A_1 has fired in the object net, then in each clause C_i in which A_1 appears the system and object net transitions using channel C_iA_1 may fire.

Thus the marking $\mu_e = \widehat{p}_e[\mathbf{0}]$ is reachable from $\mu_0 = \widehat{p}_0[\mathbf{0}]$ in the constructed net iff F is satisfiable. Since the construction is possible in polynomial time and the constructed net system is a strongly deterministic ptGSM, we conclude that the reachability problem is NP-hard for this net class. □

Dropping the strong determinism property and only requiring deterministic ptGSMs we can, by a slight modification of the construction in Lemma 5, show a stronger result for deterministic ptGSMs.

Theorem 5. *The reachability problem for deterministic and general ptGSMs is* PSPACE-*hard.*

Proof. We construct a ptGSM from a given LBA as in the proof of Lemma 5. The problem is that each place of the object net constructed there might have many incoming and many outgoing arcs, while here exactly one incoming and exactly one outgoing arc should exist. To circumvent this our new object net has the same set of places but for each place p it has one transition t_p^+ with an arc to p and one transition t_p^- with an arc from p. The transitions use the channels c_p^+ and c_p^+ respectively.

The transitions in the system net are replaced by two transitions and one place as follows. If before we had the transition $\widehat{t}_{r,j}$ and the two arcs $(\widehat{p}_{i,j}, \widehat{t}_{r,j})$ and $(\widehat{t}_{r,j}, \widehat{p}_{i+1,j+1})$ as depicted to the left of Figure 2, then we replace these with two transitions $\widehat{t}^-_{r,j}, \widehat{t}^+_{r,j}$ and the arcs $(\widehat{p}_{i,j}, \widehat{t}^-_{r,j}), (\widehat{t}^-_{r,j}, \widehat{t}^+_{r,j}), (\widehat{t}^+_{r,j}, \widehat{p}_{i+1,j+1})$. The channels used by the two new transitions depend on the LBA's transition k_r. If $k_r = (q_i, A_b, R, q_{i+1}, A_1)$ (again, as depicted in Figure 2), then the token denoting that the cell c_j is marked with A_b has to be removed and a token denoting that the cell c_j is marked with A_1 has to be added, i.e. the transition $\widehat{t}^-_{r,j}$ uses channel $c^-_{p_{b,j}}$ and the transition $\widehat{t}^-_{r,j}$ the channel $c^+_{p_{1,j}}$.

The construction is still possible in polynomial time, but the constructed net system is now a ptGSM. Thus the Theorem follows. □

The tpGSM Case. We now turn to tpGSMs, which on the contrary to ptGSMs above (see Lemma 4) may employ more than one object net.

At first glance this thus seems to be a complicated case, too. However, the structure of the system net is heavily limited by being a GSM *and* a T-net. In this setting each object net N resides on a circle C_N in the system net. While these circles may have transitions in common and while thus the object nets may interact with each other, this interaction is rather limited as are the reachable system net places for each object net.

For this reason, we suspect that the reachability problem for tpGSMs is solvable in polynomial time. So far we have been unable to prove this conjecture. If true, the results obtained would imply that the restriction of the system net to a T-net is far more severe than the restriction of the system net to a P-net. This would also be in line with the results presented here for tt- and ppGSMs, where reachability is solvable in polynomial time for a strongly deterministic ttGSM, but requires polynomial space for a ppGSM.

5 Conclusion and Outlook

Table 1 summarizes the results obtained so far concerning the complexity of the reachability problem for the the different variants of generalised state machines introduced in this paper. (Note that the problem is decidable for all of them due to Theorem 1.) While GSMs with the restrictions applied here will probably not be very useful in a modelling context, the theoretical results obtained might help in the analysis of less restricted models.

Table 1. The results obtained so far

	str. det. GSM	det. GSM	GSM
ttGSM	P[a]	?	?
ppGSM	PSPACE-complete	PSPACE-complete	PSPACE-complete
ptGSM	NP-hard	PSPACE-hard[b]	PSPACE-hard
tpGSM	?	?	?

[a] First proved in [10].
[b] Proved to be NP-hard in [10].

In retrospect we applied many long known techniques to solve problems in our new setting. For example the idea used in the proof of PSPACE-hardness in Lemma 5 was used in the late seventies by Jones, Landweber, and Lien [13] and they even state that these technique has already been used by Petri in his dissertation. Other ideas and techniques we used stem from CTL model checking and the theory of P- and T-nets. However, it is interesting to note that these techniques already seem to fail in the case of pt- and tpGSMs. Despite being just a combination of P- and T-nets and thus quite simple in structure, the possibility to interact seems to render the techniques used for pp- and ttGSMs useless. This is especially surprising in the case of ptGSMs where only one object net is present. In this regard the high complexity evident in the second and third row is also surprising, since both net classes employ only one object net.

Recalling the discussion at the end of Section 4 concerning tpGSMs, it seems that the restriction of the system net to a T-net results in easier cases than the restriction of the system net to a P-net, despite the fact that more than one object net might be present. While this speculation still needs to be confirmed, it is already certain, that the restriction of the system net to a P-net results in hard cases, regardless of the restriction imposed on the object net.

In future work we want to pinpoint the exact complexity of the open cases in Table 1. Also, we want to refine the restrictions imposed on the structure of the system and the object nets and so make the formalism more usable. We hope to arrive at a handy formalism eventually which serves as a helpful tool, allowing the modelling of many applications involving systems in systems, moving objects, and communication, while still permitting algorithmic verification and analysis of the model used employing only affordable resources.

References

1. Bednarczyk, M.A., Bernardinello, L., Pawłowski, W., Pomello, L.: Modelling Mobility with Petri Hypernets. In: Fiadeiro, J.L., Mosses, P.D., Orejas, F. (eds.) WADT 2004. LNCS, vol. 3423, pp. 28–44. Springer, Heidelberg (2005)
2. Cardelli, L., Gordon, A.D.: Mobile ambients. Theoretical Computer Science 240, 177–213 (2000)
3. Castagna, G., Vitek, J., Nardelli, F.Z.: The seal calculus. Information and Computation 201, 1–54 (2005)
4. Desel, J., Esparza, J.: Free choice Petri nets. Cambridge University Press, New York (1995)
5. Esparza, J.: Decidability and Complexity of Petri Net Problems – an Introduction. In: Reisig, W., Rozenberg, G. (eds.) APN 1998. LNCS, vol. 1491, pp. 374–428. Springer, Heidelberg (1998)
6. Esparza, J., Nielsen, M.: Decidability issues for petri nets - a survey. J. Inform. Process. Cybernet 30(3), 143–160 (1994)
7. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman and Company, San Francisco (1979)
8. Haddad, S., Poitrenaud, D.: Theoretical Aspects of Recursive Petri Nets. In: Donatelli, S., Kleijn, J. (eds.) ICATPN 1999. LNCS, vol. 1639, pp. 228–247. Springer, Heidelberg (1999)
9. Heitmann, F., Köhler-Bußmeier, M.: On defining conflict-freedom for object nets. In: Farwer, B. (ed.) Proceedings of the International Workshop on Logic, Agents, and Mobility, LAM 2011 (2011)
10. Heitmann, F., Köhler-Bußmeier, M.: Restricting generalised state machines. In: Szczuka, M., Czaja, L., Skowron, A., Kacprzak, M. (eds.) Proceedings of the Concurrency, Specification and Programming (CS&P 2011), Białystok University of Technology, Pułtusk, Poland (2011)
11. Hiraishi, K.: PN^2: An Elementary Model for Design and Analysis of Multiagent Systems. In: Arbab, F., Talcott, C. (eds.) COORDINATION 2002. LNCS, vol. 2315, pp. 220–235. Springer, Heidelberg (2002)
12. Hoffmann, K., Ehrig, H., Mossakowski, T.: High-Level Nets with Nets and Rules as Tokens. In: Ciardo, G., Darondeau, P. (eds.) ICATPN 2005. LNCS, vol. 3536, pp. 268–288. Springer, Heidelberg (2005)

13. Jones, N.D., Landweber, L.H., Lien, Y.E.: Complexity of some problems in Petri nets. Theoretical Computer Science 4, 277–299 (1977)
14. Karp, R.M.: Reducibility among combinatorial problems. In: Miller, R.E., Thatcher, J.W. (eds.) Complexity of Computer Computations, pp. 85–103. Plenum Press, New York (1972)
15. Köhler, M.: The reachability problem for object nets. Fundamenta Informaticae 79(3-4), 401–413 (2007)
16. Köhler, M., Moldt, D., Rölke, H.: Modelling Mobility and Mobile Agents Using Nets within Nets. In: van der Aalst, W.M.P., Best, E. (eds.) ICATPN 2003. LNCS, vol. 2679, pp. 121–139. Springer, Heidelberg (2003)
17. Köhler, M., Rölke, H.: Concurrency for mobile object-net systems. Fundamenta Informaticae 54(2-3) (2003)
18. Köhler, M., Rölke, H.: Properties of Object Petri Nets. In: Cortadella, J., Reisig, W. (eds.) ICATPN 2004. LNCS, vol. 3099, pp. 278–297. Springer, Heidelberg (2004)
19. Köhler, M., Rölke, H.: Reference and Value Semantics Are Equivalent for Ordinary Object Petri Nets. In: Ciardo, G., Darondeau, P. (eds.) ICATPN 2005. LNCS, vol. 3536, pp. 309–328. Springer, Heidelberg (2005)
20. Köhler-Bußmeier, M.: Hornets: Nets within Nets Combined with Net Algebra. In: Franceschinis, G., Wolf, K. (eds.) PETRI NETS 2009. LNCS, vol. 5606, pp. 243–262. Springer, Heidelberg (2009)
21. Köhler-Bußmeier, M., Heitmann, F.: Safeness for object nets. Fundamenta Informaticae 101(1-2), 29–43 (2010)
22. Köhler-Bußmeier, M., Heitmann, F.: Liveness of safe object nets. Fundamenta Informaticae 112(1), 73–87 (2011)
23. Lakos, C.A.: A Petri Net View of Mobility. In: Wang, F. (ed.) FORTE 2005. LNCS, vol. 3731, pp. 174–188. Springer, Heidelberg (2005)
24. Lomazova, I.A.: Nested Petri nets – a formalism for specification of multi-agent distributed systems. Fundamenta Informaticae 43(1-4), 195–214 (2000)
25. van Hee, K.M., Lomazova, I.A., Oanea, O., Serebrenik, A., Sidorova, N., Voorhoeve, M.: Nested Nets for Adaptive Systems. In: Donatelli, S., Thiagarajan, P.S. (eds.) ICATPN 2006. LNCS, vol. 4024, pp. 241–260. Springer, Heidelberg (2006)
26. Savitch, W.: Relationship between nondeterministic and deterministic tape complexities. J. on Computer and System Sciences 4, 177–192 (1970)
27. Valk, R.: Modelling concurrency by task/flow EN systems. In: 3rd Workshop on Concurrency and Compositionality. No. 191 in GMD-Studien, Gesellschaft für Mathematik und Datenverarbeitung, St. Augustin, Bonn (1991)
28. Valk, R.: Petri Nets as Token Objects: An Introduction to Elementary Object Nets. In: Desel, J., Silva, M. (eds.) ICATPN 1998. LNCS, vol. 1420, pp. 1–24. Springer, Heidelberg (1998)

Stochastic Modeling and Analysis Using QPME: Queueing Petri Net Modeling Environment v2.0

Simon Spinner, Samuel Kounev, and Philipp Meier

Karlsruhe Institute of Technology (KIT), 76131 Karlsruhe, Germany
{simon.spinner,kounev}@kit.edu, mail@philippmeier.com

Abstract. Queueing Petri nets are a powerful formalism that can be exploited for modeling distributed systems and analyzing their performance and scalability. By combining the modeling power and expressiveness of queueing networks and stochastic Petri nets, queueing Petri nets provide a number of advantages. In this paper, we present our tool QPME (Queueing Petri net Modeling Environment) for modeling and analysis using queueing Petri nets. QPME provides an Eclipse-based editor for building queueing Petri net models and a powerful simulation engine for analyzing these models. The development of the tool started in 2003 and since then the tool has been distributed to more than 120 organizations worldwide.

Keywords: Queueing Petri nets, stochastic modeling and analysis, simulation.

1 Introduction

Introduced in 1993 by Falko Bause [1], the Queueing Petri Net (QPN) formalism combines the modeling power and expressiveness of queueing networks and stochastic Petri nets. QPNs are commonly used for the performance evaluation of computer systems because they provide a number of benefits compared to traditional queueing networks and stochastic Petri nets. QPNs enable the integration of hardware and software aspects of system behavior in the same model. In addition to hardware contention and scheduling strategies, QPNs make it easy to model simultaneous resource possession, synchronization, asynchronous processing and software contention. Another advantage of QPNs is that they can be used to combine qualitative and quantitative system analysis. A number of efficient techniques from Petri net theory can be exploited to verify some important qualitative properties of QPNs. The latter not only help to gain insight into the behavior of the system, but are also essential preconditions for a successful quantitative analysis [4]. The main idea behind the QPN formalism was to add queueing and timing aspects to the places of Colored Generalized Stochastic Petri Nets (CGSPNs). This is done by allowing queues (service stations) to be integrated into places of CGSPNs. A place of a CGSPN that has an integrated queue is called a queueing place and consists of two components, the queue and a depository for tokens which have completed their service at the queue. For a detailed description of the QPN formalism see [1].

The major goal of QPME (Queueing Petri net Modeling Environment) is to support the modeling and analysis of QPN models. The presented tool provides user-friendly graphical editors enabling the user to quickly and easily construct QPN models. It offers a highly optimized simulation engine that can be used to analyze QPN models efficiently. The simulation engine enables the analysis of QPN models too large to be analyzable with analytical techniques due to the state space explosion problem [8]. QPME also offers advanced features for processing and visualizing the results of simulating a QPN model. In addition, being implemented in Java, QPME runs on all major platforms and is widely accessible. The QPN formalism can be used for stochastic modeling in many domains. One major area of application is the performance analysis of computer systems. The tool has been successfully used in several performance modeling studies, e.g. in [10, 11, 14, 16].

The development of QPME started in 2003 at the Technische Universität Darmstadt and has been continuously extended since then. Currently, the tool is developed and maintained by the Descartes Research Group[1] at Karlsruhe Institute of Technology (KIT). Since May 2011, QPME is available in version 2.0 under an open-source license (Eclipse Public License 1.0). QPME has been distributed to more than 120 universities and research organizations worldwide so far.

The rest of this paper is organized as follows: Section 2 provides an overview of the functionality provided by QPME. Section 3 gives some technical insights into its implementation. Section 4 describes typical use cases of the tool and Sect. 5 provides a comparison with other tools for QPNs. Finally, the paper is wrapped up with some concluding remarks in Sect. 7.

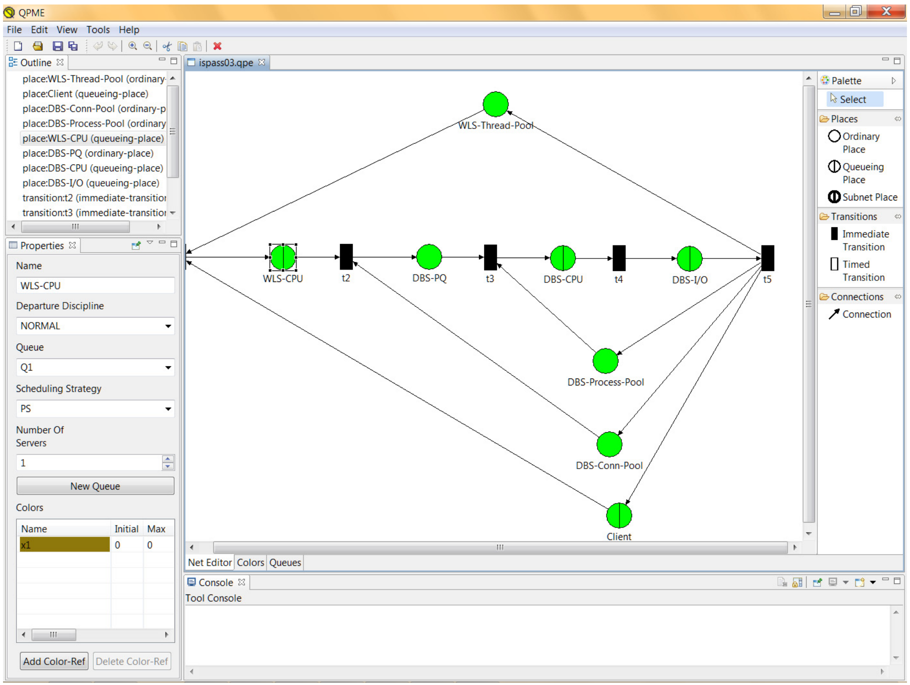
2 Functionality

2.1 Queueing Petri net Editor (QPE)

QPE is a graphical editor for QPNs. The user can create QPN models with a simple drag-and-drop approach. Figure 1 shows the QPE main window which is comprised of four views. The *Main Editor View* displays the graphical representation of the currently edited QPN. The palette contains the set of QPN elements that can be inserted in a QPN model by drag-and-drop, such as places, transitions, and connections. Furthermore, it provides editors for the central definition of colors and queues used in a QPN model. In the *Properties View* the user can edit the properties of the element currently selected in the QPN model. For queueing places, for instance, the user can specify a scheduling strategy and service time distributions for each color in this view. The *Outline View* shows a list of all elements in the QPN model. The *Console View* displays the output when simulating a QPN model.

In a QPN, a transition defines a set of firing modes. An incidence function specifies the behavior of the transition for each of its firing modes in terms of tokens destroyed and/or created in the places of the QPN. Figure 2 shows the

[1] `http://www.descartes-research.net`



place:WLS-Thread-Pool (ordinary-
place:Client (queueing-place)
place:DBS-Conn-Pool (ordinary-p
place:DBS-Process-Pool (ordinary
place:WLS-CPU (queueing-place)
place:DBS-PQ (ordinary-place)
place:DBS-CPU (queueing-place)
place:DBS-I/O (queueing-place)
transition:t2 (immediate-transitio
transition:t3 (immediate-transitio

Name: WLS-CPU
Departure Discipline: NORMAL
Queue: Q1
Scheduling Strategy: PS
Number Of Servers: 1

Colors

Name	Initial	Max
x1	0	0

Tool Console

Fig. 1. QPE Main Window

Incidence Function Editor, which is used to edit the incidence function of a transition. Once opened this editor displays the transition input places on the left, the transition firing modes in the middle and the transition output places on the right. Each place (input or output) is displayed as a rectangle containing a separate circle for each token color allowed in the place. The user can create connections from token colors of input places to modes or from modes to token colors of output places. If a connection is created between a token color of a place and a mode, this means that when the transition fires in this mode, tokens of the respective color are removed from the place. Similarly, if a connection is created between a mode and a token color of an output place, this means that when the transition fires in this mode, tokens of the respective color are deposited in the place.

In addition to the basic features described above, QPE has several characterizing features that improve the model expressiveness of QPNs and simplify the creation of complex QPN models. Special mention must be made of the following features:

– *Central color management.* The user can define token colors globally for the whole QPN instead of on a per place basis. Instead of having to define the color multiple times, the user can define it one time and then reference it in all places where it is used. This saves time, makes the model definition

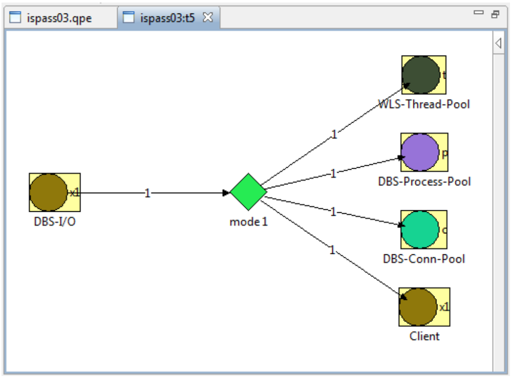


Fig. 2. QPE Incidence Function Editor

more compact, and last but not least, it makes the modeling process less error-prone since references to the same token color are specified explicitly.

– *Shared queues.* The user can specify that multiple queueing places share the same underlying physical queue[2]. In QPE, queues are defined centrally (similar to token colors) and once defined they can be referenced from inside multiple queueing places. This allows to use queueing places to represent software entities, e.g., software components, which can then be mapped to different hardware resources modeled as queues [16]. Shared queues are not supported in standard QPN models [16].
– *Hierarchical QPNs.* Subnet places can contain complete child QPN models. Hierarchical QPNs enable to model layered systems and improve the understandability of huge QPNs. QPE fully supports hierarchical QPNs.
– *Departure Disciplines.* Departure disciplines determine the order in which tokens arriving at an ordinary place or a depository of a queueing place become available for output transitions. QPE supports two disciplines: Normal (used by default) and FIFO. The former implies that tokens become available for output transitions immediately after arriving at an ordinary place or depository whereas in the latter case a token can only leave the place or depository if all tokens that have arrived before it have left. For an example of how this extension of the QPN formalism can be exploited and the benefits it provides we refer the reader to [7].

2.2 Simulation of QPN Model (SimQPN)

SimQPN is a discrete-event simulation engine specialized for QPNs. It simulates QPN models directly and has been designed to exploit the knowledge of the

[2] While the same effect can be achieved by using multiple subnet places mapped to a nested QPN containing a single queueing place, this would require expanding tokens that enter the nested QPN with a *tag* to keep track of their origin as explained in [3].

structure and behavior of QPNs to improve the efficiency of the simulation. Therefore, SimQPN provides much better performance than a general purpose simulator would provide, both in terms of the speed of simulation and the quality of output data provided.

Model Support. SimQPN implements most, but not all of the QPN elements that can be modeled in QPE. It currently supports three different scheduling strategies for queues: Processor-Sharing (PS), Infinite Server (IS) and First-Come-First-Served (FCFS). A wide range of service time distributions are supported including Beta, BreitWigner, ChiSquare, Gamma, Hyperbolic, Exponential, ExponentialPower, Logarithmic, Normal, StudentT, Uniform and VonMises as well as deterministic and empirical distributions. All of the characterizing features of QPE described in Sect. 2.1 are fully supported by the SimQPN simulator. A current limitation of SimQPN is the missing support for timed transitions[3] and immediate queueing places. The spectrum of scheduling strategies and service time distributions supported by SimQPN will be extended. Support for timed transitions and immediate queueing places is also planned for future releases.

Output Data. SimQPN offers the ability to configure what data exactly to collect during the simulation and what statistics to provide at the end of the run. This can be specified on a per *location* basis where location is defined to have one of the following five types: 1. ordinary places, 2. queue of queueing places (considered from the perspective of the place), 3. depository of queueing places, 4. queues (considered from the perspective of all places it is part of), and 5. probes.

A probe enables the user to specify a region of interest for which data should be collected during simulation. The region of a probe includes one or more places and is defined by one start and one end place. The goal is to evaluate the time tokens spend in the region when moving between its begin and end place. The probe starts its measurements for each token entering its region at the start place and updates the statistics when the token leaves at the end place. Probes are realized by attaching timestamps to individual tokens. With probes it is possible to determine statistics for the residence time of tokens in a region of interest.

For each location the user can choose between six modes of data collection . The higher the mode, the more information is collected and the more statistics are provided. Since collecting data costs CPU time, the more data is collected, the slower the simulation would progress. Therefore, with data collection modes the user can speed up the simulation by avoiding the collection of data that is not required. The six data collection modes are defined as follows:

– *Mode 0.* No data is collected.
– *Mode 1.* Only token throughput data is collected.

[3] In most cases a timed transition can be approximated by a serial network consisting of an immediate transition, a queueing place and a second immediate transition.

- *Mode 2.* Additionally, data to compute token population, token occupancy, and queue utilization is collected
- *Mode 3.* Token residence time data is collected (maximum, minimum, mean, standard deviation, steady state mean, and confidence interval of steady state mean).
- *Mode 4.* This mode adds a histogram of observed token residence times.
- *Mode 5.* Additionally token residence times are dumped to a file for further analysis with external tools.

Steady State Analysis. SimQPN supports two methods for the estimation of steady state mean residence times of tokens inside the various locations of the QPN. These are the well-known *Method of Independent Replications* (in its variant referred to as replication/deletion approach) and the classical *Method of Non-overlapping Batch Means (NOMB).* We refer the reader to [12, 15] for an introduction to these methods. Both of them can be used to provide point and interval estimates of the steady state mean token residence time.

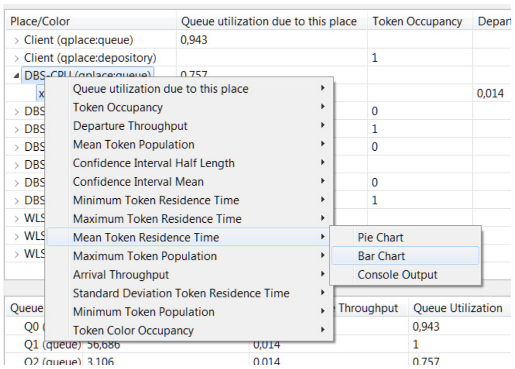
We have validated the analysis algorithms implemented in SimQPN by subjecting them to a rigorous experimental analysis and evaluating the quality of point and interval estimates [9]. Our analysis showed that data reported by SimQPN is very accurate and stable. Even for residence time, the metric with highest variation, the standard deviation of point estimates did not exceed 2.5% of the mean value. In all cases, the estimated coverage of confidence intervals was less than 2% below the nominal value (higher than 88% for 90% confidence intervals and higher than 93% for 95% confidence intervals).

Furthermore, SimQPN includes an implementation of the *Method of Welch* for determining the length of the initial transient (warm-up period). We have followed the rules in [12] for choosing the number of replications, their length and the window size.

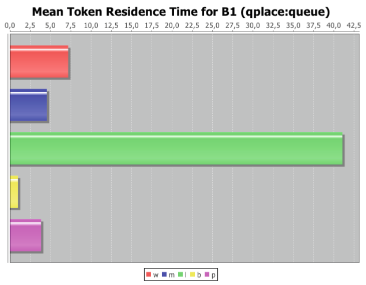
2.3 Processing and Visualization of Simulation Results

After a successful simulation run, SimQPN saves the results from the simulation in an XML file with a `.simqpn` extension. QPE provides an advanced query engine for the processing and visualization of simulation results. The query engine allows to define queries on the simulation results in order to filter, aggregate and visualize performance data for multiple places, queues and colors of the QPN. The results from the queries can be displayed in textual or graphical form. QPE provides the following two query editors:

- *Simple Query Editor.* The user can quickly filter and visualize metrics of a *single* location or token color with a few clicks. Currently, three visualization options are available: "Pie Chart", "Bar Chart" and "Console Output".
- *Advanced Query Editor.* The user can create complex queries including the aggregation of metrics over multiple locations and token colors with a powerful user interface. The following two aggregation operators are currently supported: "Average" and "Sum".


Place/Color
Queue utilization due to this place
Token Occupancy
Depart
Client (qplace:queue) 0,943
Client (qplace:depository) 1
DBS-CPU (qplace:queue) 0,757
0,014
DBS 0
DBS 1
DBS 0
DBS
DBS 0
DBS 1
WLS
WLS
WLS
Queue utilization due to this place
Token Occupancy
Departure Throughput
Mean Token Population
Confidence Interval Half Length
Confidence Interval Mean
Minimum Token Residence Time
Maximum Token Residence Time
Mean Token Residence Time
Maximum Token Population
Arrival Throughput
Standard Deviation Token Residence Time
Minimum Token Population
Token Color Occupancy
Pie Chart
Bar Chart
Console Output
Queue
Throughput
Queue Utilization
Q0 (0,943
Q1 (queue) 56,686 0,014 1
Q2 (queue) 3,106 0,014 0,757

(a) Metrics context menu



(b) Bar Chart

Fig. 3. Simple query editor

Figure 3(a) shows an area of the user interface of the simple query editor. The statistics for all QPN places are presented in the table in the background. By opening the context menu on one of the places a context menu with a set of metrics gets available. When choosing the bar chart for the mean token residence time, the diagram shown in Fig. 3(b) is displayed.

3 Architecture

QPME is based on the Eclipse OSGi framework and comes as a stand-alone *Rich Client Plattform (RCP)* application. QPME is written completely in Java making it widely accessible on different platforms. The architecture of QPME is plugin-based and consists of the following three core plugins: `qpe.base` contains the set of editors for building QPN models (QPE) and for analyzing simulation results (simple and advanced query editor), `qpe.simqpn.kernel` provides the core of the SimQPN simulator, and `qpe.simqpn.ui` contains the graphical wizard for calling the SimQPN simulator from within QPE. The core plugins are highly integrated with each other while at the same time it is possible to also use them separately.

The editors of QPE are based on the Graphical Editing Framework (GEF)[4]. Being a GEF application, QPE is based on the model-view-controller (MVC) architecture. The model in our case is the QPN being defined, the views provide graphical representations of the QPN, and finally the controller connects the model with the views, managing the interactions among them. The individual editors (net, incidence function, subnet, queues, and colors) are all realized as different views which work on one single model. Thus it is ensured that all editors working on the same QPN model are always up-to-date.

[4] `http://www.eclipse.org/gef/`

QPME knows two types of files. Files with the extension `.qpe` contain a complete QPN model. They are XML files with an own schema based on the Petri Net Markup Language (PNML) [6] with some changes and extensions to support the additional constructs available in QPN models. Files with the extension `.simqpn` contain the results of a simulation run. They are also XML files and can be opened with the simple or advanced query editor.

SimQPN is realized as a discrete-event simulator. It is very light-weight and optimized to exploit the knowledge of the structure and behavior of QPNs to improve the efficiency of the simulation. A simulation run consists of five phases. First the QPN model is loaded from a `.qpe` file or it is directly imported from an open editor in QPE. If the QPN contains subnet places, a model transformation is applied in phase two which integrates the subnets into the main net recursively resulting in a flat net. This simplifies the simulation of hierarchical QPNs. In phase three, the configuration of the simulation is processed and the simulation engine is configured accordingly. Especially, the simulation engine is set up to collect only the data that is necessary to calculate the requested statistics. By avoiding the collection of data that is not requested by the user, the simulation performance can be significantly sped up in a lot of cases. In phase four, the actual simulation is performed. An event loop advances the simulation clock and in each iteration all events that are scheduled at the current simulation time are processed. SimQPN utilizes the *Colt* open-source library, which is developed at CERN[5], for random number generation. The loop is running until the configured relative or absolute precision is reached or the configured maximum run length is reached. After the simulation, the requested statistics are calculated from the data collected during the simulation and the data is stored in a `.simqpn` file.

4 Use Cases

In [7], we have developed a methodology for performance modeling of distributed component-based systems using QPNs. The methodology has been applied to model a number of systems ranging from simple systems to systems of realistic size and complexity. Here, QPME can be used as a powerful tool for performance and scalability analyses. Some examples of modeling studies using QPME can be found in [10, 11, 14, 16].

Furthermore, QPME can be used at the design time of software systems for solving architecture-level performance models, e.g., the Palladio Component Model (PCM) [5]. In [13], we described how to derive QPN models from PCM models automatically using a formal mapping. In numerous representative case studies, we showed that SimQPN predicted all mean value metrics with high accuracy while the analysis overhead compared to the standard simulator of PCM could be significantly reduced, in many cases by an order of magnitude [13].

[5] `http://acs.lbl.gov/software/colt/`

5 Comparison with Other Tools

While the QPN modeling paradigm provides many important benefits, there are currently few tools that support the modeling and analysis of systems using QPNs. According to [17], apart from the QPME tool presented in this paper, the only tool that is available is the HiQPN-Tool [2] developed at the University of Dortmund. HiQPN can be used to build and analyze QPN models, however, it only supports analytical solution techniques. As demonstrated in [8], QPN models of realistic systems are too large to be analyzable using analytical techniques due to the state space explosion problem. Furthermore, it is only available on Sun-OS 5.5.x / Solaris 2, which significantly limits its accessibility. In contrast, QPME is implemented in Java and runs on all major platforms. It provides a highly optimized simulation engine capable of analyzing models of realistically sized systems.

6 Installation

The binaries of QPME for Windows, Linux and MacOS X and the source code can be obtained from `http://qpme.sourceforge.net` free-of-charge. The QPME binary drops are installed by extracting the zipped archive in an arbitrary location on the hard disk. QPME is open-source and is distributed under the Eclipse Public License (EPL) 1.0.

7 Conclusion

In this paper, we presented QPME 2.0, our tool for modeling and analysis using queueing Petri nets. QPME provides a user-friendly graphical interface enabling the user to quickly and easily construct QPN models. It offers a highly optimized simulation engine that can be used to analyze models of realistically-sized systems. In addition, being implemented in Java, QPME runs on all major platforms and is widely accessible. QPME provides a robust and powerful tool for performance analysis making it possible to exploit the modeling power and expressiveness of queueing Petri nets to their full potential. The tool is available free-of-charge under an open-source license.

Acknowledgements. This work was funded by the German Research Foundation (DFG) under grant No. KO 3445/6-1. We acknowledge the support of Frederik Zipp from KIT.

References

1. Bause, F.: Queueing Petri Nets - A formalism for the combined qualitative and quantitative analysis of systems. In: Proc. of 5th Intl. Workshop on Petri Nets and Perf. Models, Toulouse, France, October 19-22 (1993)

2. Bause, F., Buchholz, P., Kemper, P.: QPN-Tool for the Specification and Analysis of Hierarchically Combined Queueing Petri Nets. In: Beilner, H., Bause, F. (eds.) MMB/TOOLS 1995. LNCS, vol. 977. Springer, Heidelberg (1995)
3. Bause, F., Buchholz, P., Kemper, P.: Integrating Software and Hardware Performance Models Using Hierarchical Queueing Petri Nets. In: Proc. of the 9. ITG / GI - Fachtagung Messung, Modellierung und Bewertung von Rechen- und Kommunikationssystemen, Freiberg, Germany (1997)
4. Bause, F., Kritzinger, F.: Stochastic Petri Nets - An Introduction to the Theory. Vieweg Verlag (2002)
5. Becker, S., Koziolek, H., Reussner, R.: The Palladio Component Model for Model-Driven Performance Prediction: Extended version. Jour. of Sys. and Softw. (2008)
6. Billington, J., Christensen, S., van Hee, K., Kindler, E., Kummer, O., Petrucci, L., Post, R., Stehno, C., Weber, M.: The Petri Net Markup Language: Concepts, Technology, and Tools. In: Proc. of 24th Intl. Conf. on Application and Theory of Petri Nets, Eindhoven, Holland, June 23-27 (2003)
7. Kounev, S.: Performance Modeling and Evaluation of Distributed Component-Based Systems using Queueing Petri Nets. IEEE Trans. on Softw. Eng. 32(7), 486–502 (2006)
8. Kounev, S., Buchmann, A.: Performance Modelling of Distributed E-Business Applications using Queuing Petri Nets. In: Proc. of the 2003 IEEE Intl. Symp. on Performance Analysis of Systems and Software, Austin, USA, March 20-22 (2003)
9. Kounev, S., Buchmann, A.: SimQPN - a tool and methodology for analyzing queueing Petri net models by means of simulation. Performance Evaluation 63(4-5), 364–394 (2006)
10. Kounev, S., Nou, R., Torres, J.: Autonomic QoS-Aware Resource Management in Grid Computing using Online Performance Models. In: Proc. of 2nd Intl. Conf. on Perf. Evaluation Methodologies and Tools - VALUETOOLS, Nantes, France, October 23-25 (2007)
11. Kounev, S., Sachs, K., Bacon, J., Buchmann, A.: A Methodology for Performance Modeling of Distributed Event-Based Systems. In: Proc. of 11th IEEE Intl. Symp. on Object/Comp./Service-oriented Real-time Distr. Computing (ISORC), Orlando, USA (May 2008)
12. Law, A., Kelton, D.W.: Simulation Modeling and Analysis, 3rd edn. Mc Graw Hill Companies, Inc. (2000)
13. Meier, P., Kounev, S., Koziolek, H.: Automated Transformation of Palladio Component Models to Queueing Petri Nets. In: 19th IEEE/ACM Intl. Symp. on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS 2011) (July 2011)
14. Nou, R., Kounev, S., Julia, F., Torres, J.: Autonomic QoS control in enterprise Grid environments using online simulation. Jour. of Sys. and Softw. 82(3), 486–502 (2009)
15. Pawlikowski, K.: Steady-State Simulation of Queueing Processes: A Survey of Problems and Solutions. ACM Computing Surveys 22(2), 123–170 (1990)
16. Sachs, K.: Performance Modeling and Benchmarking of Event-based Systems. PhD thesis, TU Darmstadt (2010)
17. University of Hamburg. Petri Net Tool Database (2008), http://www.informatik.uni-hamburg.de/TGI/PetriNets/tools

Snoopy – A Unifying Petri Net Tool

Monika Heiner, Mostafa Herajy, Fei Liu, Christian Rohr, and Martin Schwarick

Computer Science Institute, Brandenburg University of Technology Cottbus
Postbox 10 13 44, 03013 Cottbus, Germany
snoopy@informatik.tu-cottbus.de
http://www-dssz.informatik.tu-cottbus.de

Abstract. The tool Snoopy provides a unifying Petri net framework which has particularly many application scenarios in systems and synthetic biology. The framework consists of two levels: uncoloured and coloured. Each level comprises a family of related Petri net classes, sharing structure, but being specialized by their kinetic information. Petri nets of all net classes within one level can be converted into each other, while changing the level involves user-guided folding or automatic unfolding. Models can be hierarchically structured, allowing for the mastering of larger networks. Snoopy supports the simultaneous use of several Petri net classes; the graphical user interface adapts dynamically to the active one. Built-in animation and simulation (depending on the net class) are complemented by export to various analysis tools. Snoopy facilitates the extension by new Petri net classes thanks to its generic design.

Keywords: hierarchical (coloured) qualitative/stochastic/continuous/ hybrid Petri nets, modelling, animation, simulation.

1 Overview

Petri nets may easily serve as a convenient umbrella formalism integrating qualitative and quantitative (i.e. stochastic, continuous, or hybrid) modelling and analysis techniques. Thus Petri nets are immediately ready to address distinctive modelling demands in systems and synthetic biology, which particularly include the dealing with biochemical reaction networks in several modelling paradigms.

Motivated by this application scenario, Snoopy is set up as a unifying Petri net framework (see Fig. 1) which can be divided into two levels: uncoloured [11] and coloured [17]. Each level comprises a family of related Petri net models, sharing structure, but being specialized by their kinetic information. Specifically, the uncoloured level contains qualitative (time-free) Place/Transition Petri nets ($\mathcal{QPN}$) as well as quantitative (time-dependent) Petri nets such as stochastic Petri nets ($\mathcal{SPN}$), continuous Petri nets ($\mathcal{CPN}$), and generalised hybrid Petri nets ($\mathcal{GHPN}$). The coloured level provides coloured counterparts of the uncoloured level, and thus consists of coloured qualitative Petri nets ($\mathcal{QPN}^\mathcal{C}$), coloured stochastic Petri nets ($\mathcal{SPN}^\mathcal{C}$), coloured continuous Petri nets ($\mathcal{CPN}^\mathcal{C}$) and coloured generalised hybrid Petri nets ($\mathcal{GHPN}^\mathcal{C}$).

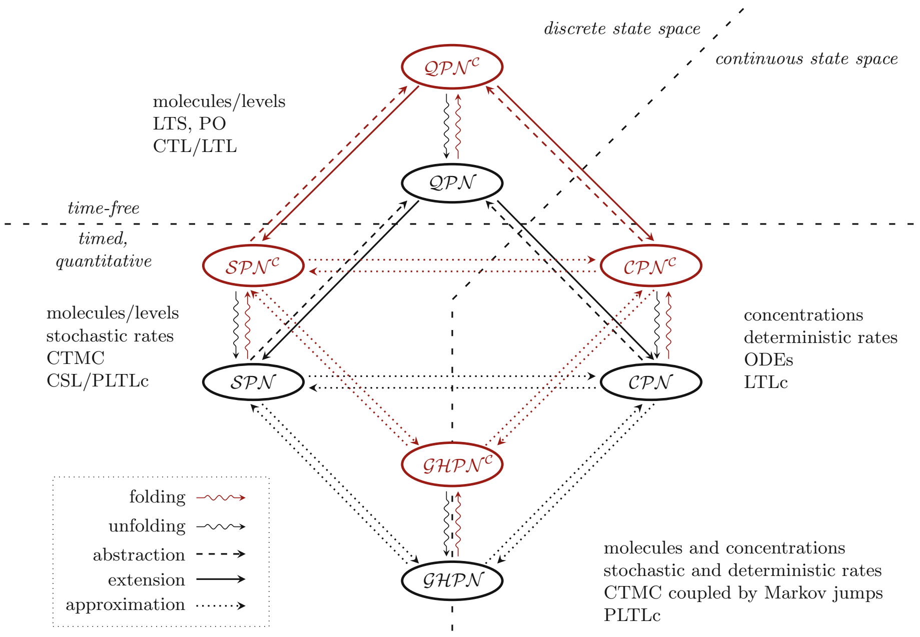


Fig. 1. Paradigms integrated in Snoopy's unifying framework

Petri nets of these net classes can be converted into each other. Obviously, there may be a loss of information in some directions (cf. arrows labelled with "abstraction" in Fig. 1). The conversion between coloured and uncoloured net classes is accomplished by means of user-guided folding or automatic unfolding (cf. arrows labelled with folding and unfolding in Fig. 1). Moving between the coloured and uncoloured level changes the style of representation, but does not change the actual net structure of the underlying reaction network. Therefore, all analysis techniques available for uncoloured Petri nets can be applied to coloured Petri nets as well.

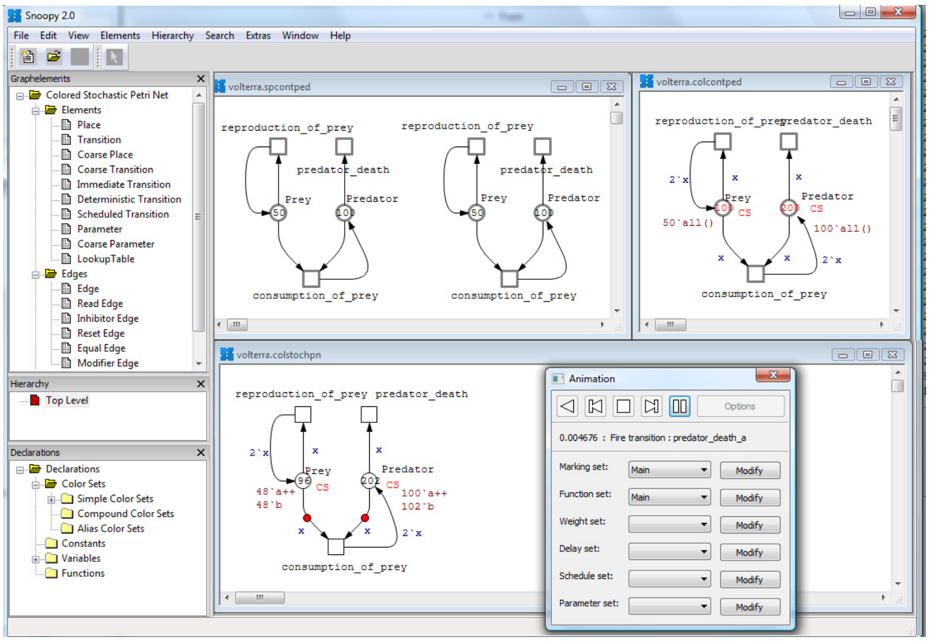
Snoopy supports the simultaneous use of different net classes, which provides the grounds to investigate one and the same case study with different modelling abstractions in various complementary ways [10], [11], [17].

2 Basic Functionalities

Snoopy offers for its net classes a graphical, unified modelling environment. See Fig. 2 for a snapshot of the user interface with a famous textbook example, the prey predator system. The user interface mainly consists of:

– graphical elements window (the top left tree control): listing all graphical elements, e.g. node elements and edge elements,
– hierarchy window (the middle left tree control): showing the model hierarchy,

– declarations window (the bottom left tree control): containing all declarations, e.g. colour sets, constants, and variables for coloured Petri nets,
– drawing canvas (the right window): drawing and showing models.



0.004676 : Fire transition : predator_death_a

Marking set: Main
Function set: Main
Weight set:
Delay set:
Schedule set:
Parameter set:

Fig. 2. User modelling interface

Basic modelling functions provided by Snoopy include:

– define declarations (only for coloured Petri nets),
– add graphical elements, i.e. places and transitions, from the graphical elements window to the canvas and connect them using edges,
– edit or modify properties of nodes (e.g. name, initial marking) and edges (multiplicity) in their property dialogues.

Snoopy supplies two features for the design and systematic construction of larger Petri nets. Logical nodes (places/transitions) serve as connectors, and coarse transitions (coarse places) help to hide transition-bordered (place-bordered) subnets in order to design hierarchically structured nets.

Further features consistently available for all Petri net classes include: editing (cut, copy, paste), colouring of individual net elements and of computed node sets (e.g. support of place/transition invariants, siphons, traps, Parikh vectors), layouting (mirror, flip, rotate, and automatic layouting using OGDF [5]), graphical export to eps, Xfig and FrameMaker (selected net classes), and print.

Additionally, Snoopy offers execution capabilities for each net class, see next section for details.

3 Net Class Specific Functionalities

3.1 Overview

The hierarchy of the Petri net classes supported by Snoopy is given in Fig. 3.

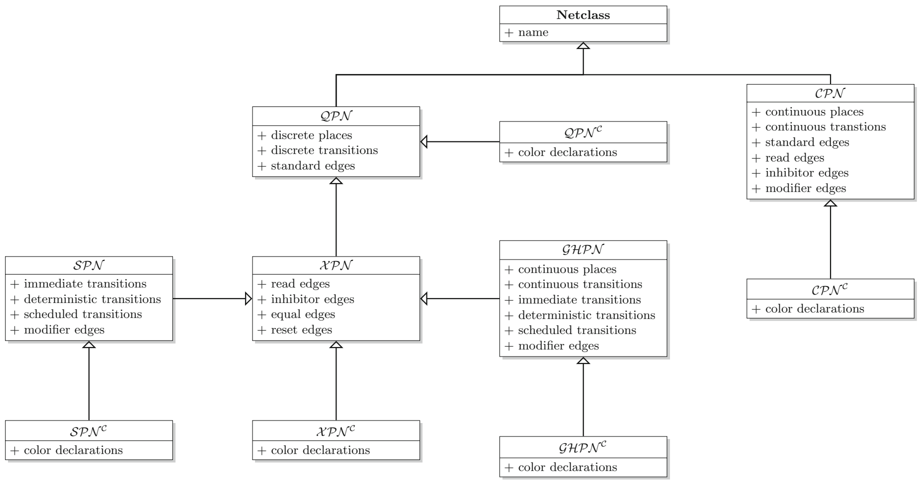


Fig. 3. Snoopy's class hierarchy

Qualitative Petri nets ($\mathcal{QPN}$). $\mathcal{QPN}$ contain standard *Place/Transition nets* (*P/T nets*) and *extended Petri nets* ($\mathcal{XPN}$). They do not involve any timing aspects; so they allow a purely qualitative modelling of, e.g., biomolecular networks. Tokens may represent molecules or abstract concentration levels [11]. $\mathcal{XPN}$ enhance standard Petri nets by four special edge types: read edges (often also called test edges), inhibitor edges, equal edges, and reset edges, see [13], [22] for details.

Stochastic Petri nets ($\mathcal{SPN}$). This net class extends $\mathcal{QPN}$ by assigning to transitions exponentially distributed waiting times, specified by firing rate functions. A rate function is generally state-dependent; it can be an arbitrary arithmetic function deploying the pre-places of a transition as integer variables and user-defined, real-valued constants (often called parameters). Pre-places can be associated with transitions by special modifier edges [22]. They may modify the transition's firing rate, but do not have an influence on the transition's enabledness.

Popular kinetics, e.g. mass-action semantics, level semantics [11], are supported by pre-defined function patterns. Each transition gets its own rate function, making up together a list of rate functions. Moreover, several rate functions lists and parameter lists as well as multiple initial markings can be maintained, allowing for quite flexible models and their systematic evaluation by series of related computational experiments.

$\mathcal{SPN}$ have been extended to *generalised stochastic Petri nets* ($\mathcal{GSPN}$) and *deterministic and stochastic Petri nets* ($\mathcal{DSPN}$). In our *extended stochastic Petri nets* ($\mathcal{XSPN}$)[12], there are the four special edge types as for $\mathcal{QPN}$, and three special transition types: immediate transitions (zero waiting time), deterministic transitions (deterministic waiting time, relative to the time point where the transition gets enabled), and scheduled transitions (scheduled to fire, if any, at single or equidistant, absolute points of the simulation time). The unrestricted use of these extended features destroys the Markov property, but the adaptation of the simulation algorithms is rather straightforward.

To simplify our life, Snoopy does not distinguish between these stochastic net classes. Thus, Snoopy's $\mathcal{SPN}$ net class is actually $\mathcal{XSPN}$; see Figure 3.

Continuous Petri nets ($\mathcal{CPN}$). Continuous Petri nets offer a graphical way to specify unambiguously systems of ordinary differential equations (ODEs) [10]. The real-valued tokens may denote concentrations. The continuous rate functions have to obey similar rules as for $\mathcal{SPN}$. Likewise, the concepts of function lists, parameter lists and initial marking lists are also applied to $\mathcal{CPN}$. Snoopy generates automatically the underlying system of ODEs. $\mathcal{CPN}$ and $\mathcal{SPN}$ provide an approximation of each other as it is depicted in Fig. 1.

Generalised hybrid Petri nets ($\mathcal{GHPN}$). Snoopy integrates all functionalities of its stochastic and continuous Petri nets into one net class, yielding generalised hybrid Petri nets [14]. $\mathcal{GHPN}$ are specifically tailored (but not limited) to models that require an interplay between stochastic and continuous behaviour. They provide a trade-off between accuracy and runtime of model simulation by adjusting the number of stochastic transitions appropriately, which can be done either statically (by the user) or dynamically (by the simulation algorithms). A typical application of $\mathcal{GHPN}$ is the hybrid representation of stiff biochemical reactions, where slow reactions are represented by stochastic transitions while fast reactions are modelled by continuous transitions.

Coloured extensions. Each uncoloured net class has a coloured counterpart [17] which inherits all features of its corresponding uncoloured net class, e.g., $\mathcal{SPN}^{\mathcal{C}}$ enjoy all special edge types and transition types of $\mathcal{SPN}$.

Snoopy provides various flexible ways to define declarations to be used in the annotations of coloured Petri nets. Data types for colour set definitions include: (1) simple types: dot, integer, string, boolean, enumeration and index, and (2) compound types: product and union. Variables, constants and functions can be defined to specify arc expressions, guards, and markings. By defining hierarchical colour sets using the product type, one can conveniently model a (biological) system evolving in multi-dimensional, e.g. 2- or 3-dimensional space [8]. Concise initial marking specifications for larger colour sets and individual rate function definitions for each transition instance are supported. Syntax checking ensures the syntactical correctness of constructed models.

3.2 Executability

Animation. Snoopy offers built-in animation for $\mathcal{QPN}$, $\mathcal{SPN}$, $\mathcal{QPN}^{\mathcal{C}}$ and $\mathcal{SPN}^{\mathcal{C}}$, see Fig. 2 for a snapshot of the animation of a $\mathcal{SPN}^{\mathcal{C}}$ model. Animation visualizes the token flow, which may give first insights in the behaviour and may help to better understand the inherent causality of the model. Animation can be triggered manually or be done in automatic mode with different firing strategies (single/intermediate/maximal step). Snoopy supports a similar animation within a standard web browser for $\mathcal{QPN}$; see Snoopy's website for a sampler.

Stochastic simulation. The underlying core semantics of $\mathcal{SPN}$ and $\mathcal{SPN}^{\mathcal{C}}$ are continuous time Markov chains (CTMC); so the simulation follows the standard Gillespie algorithm [9] enhanced by deterministic events of $\mathcal{XSPN}$. Simulation results are available as tables and can be visualized in diagrams, showing the evolution over time of the token numbers on selected places or the firing rates of selected transitions.

Simulation traces can be checked on-the fly for reachability of certain states specified by logical expressions over places. Additionally, simulation traces can be exported as averaged/single/exact traces, to be, e.g., evaluated by simulative model checking of PLTLc with the Monte Carlo Model Checker MC2 [4].

Continuous simulation. Snoopy provides 14 stiff/unstiff solvers for the numerical integration of $\mathcal{CPN}$ and $\mathcal{CPN}^{\mathcal{C}}$. These ODE solvers range from simple fixed-step-size unstiff solvers (e.g. Euler) to more sophisticated variable-order, variable-step, multi-step stiff solvers (e.g Backward Differentiation Formulas (BDFs)). In the latter case, we use SUNDIALS CVODE [15] to solve the underlying ODEs. Deterministic simulation traces are available as tables, can be visualized in diagrams, and written to files to be, e.g., checked against LTLc properties with MC2, see, e.g., [11].

Hybrid simulation. The simulation of $\mathcal{GHPN}$ can be carried out using either static or dynamic partitioning. In the former case, transition types are decided off-line by the user before the simulation starts, while in the latter case, the running simulation decides on-the-fly which transitions are considered as stochastic or continuous ones based on their current rates. In both cases, continuous transitions are simulated using an ODE solver with event detection, while stochastic transitions are simulated using Gillespie's direct method.

Moreover, the user can choose to simulate a net completely as stochastic or continuous one despite of the original place and transition types. Then transitions and places are automatically converted to the required type. This functionality gives the opportunity to experiment with different simulation algorithms without having to change the net. For instance, if the user's Petri net model is drawn to contain only stochastic transitions, later on, it could be simulated using stochastic (e.g. Gillespie) or continuous (e.g. BDFs) algorithms. In the latter case, stochastic transitions will be converted into continuous ones, while transitions of other types (immediate, deterministic, or scheduled) will remain unchanged.

Coloured extensions. Snoopy supports animation for $\mathcal{QPN}^{\mathcal{C}}$ and $\mathcal{SPN}^{\mathcal{C}}$, which can be run in automatic mode or manually controlled, e.g. with single-step animation by manually choosing a binding.

Simulation of coloured Petri nets ($\mathcal{SPN}^{\mathcal{C}}$, $\mathcal{CPN}^{\mathcal{C}}$, $\mathcal{GHPN}^{\mathcal{C}}$) is done on automatically unfolded Petri nets, and thus all simulation algorithms for uncoloured Petri nets are available for coloured Petri nets as well. In order to improve efficiency, the unfolding adopts a constraint satisfaction approach, which is implemented using the Gecode library [2]. Simulation results of coloured or uncoloured places/transitions can be shown separately or together. In-depth behaviour exploration is supported by auxiliary variables (observers) which depend on coloured places, allowing for extra measures, e.g. the sum of a group of related places.

3.3 Import/Export

All net classes can be converted into each other through export, which permits to easily switch from one net class to another one and thus to investigate a system under study with different modelling abstractions by deploying simultaneously several Petri net classes.

Additionally, there is export to numerous external analysis tools, among them Snoopy's close friends Charlie [24] and Marcie [23] ; see Snoopy's website for a complete list. Charlie's features for P/T nets include structural analysis, P/T invariants computation, and explicit CTL/LTL model checking. Marcie supports qualitative analysis and symbolic CTL model checking based on Interval Decision Diagrams. It also allows a quantitative investigation of $\mathcal{SPN}$ by means of CSL and PLTLc model checking based on numerical and simulative analysis engines.

A crucial point for the addressed main application area is Snoopy's import and export of the standard exchange format SBML, Level 2, Version 3 [7]. The Petri Net Markup Language [21] is not yet supported, but shortlisted for future plans. Complementary, we developed the Colored Abstract Net Definition Language (CANDL) [18] as a human-readable exchange format for our own toolbox.

The ODEs induced by a given $\mathcal{CPN}$ or $\mathcal{GHPN}$ can be written in Latex format and as plain ASCII text.

4 Architecture

Snoopy's architecture has been designed to gain three distinguished characteristics. (1) It is extensible; its generic design facilitates the implementation of new Petri net classes. (2) It is adaptive by supporting the simultaneous use of several models, with the graphical user interface adapting dynamically to the net class in the active window. (3) It is platform-independent.

Snoopy is written in the programming language C++ using the Standard Template Library and the cross-platform toolkit wxWidgets [6]. The main object in the data structure, see Fig. 4, is the graph object which contains modification methods and holds the associated node, edge and metadata classes. Every node class has one prototype and contains a number of nodes that are copied from

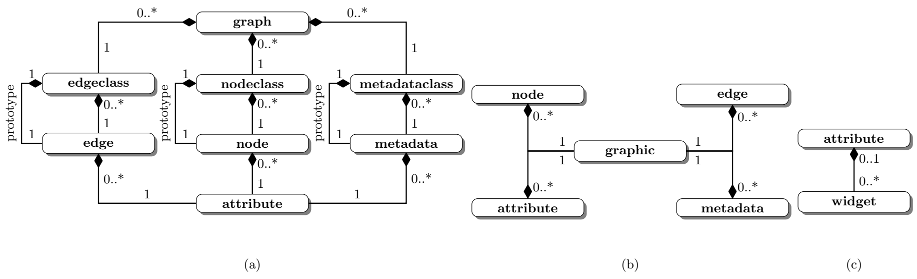


Fig. 4. (a) Internal data structure of Snoopy. (b) Graphics assigned to the graph elements. (c) Attributes connected with window interactions controls.

this prototype. The edge and metadata class are similarly structured, compare Fig. 4a. Every node, edge and metadata can have a list of attributes defining the properties of the graph elements. A graphics is assigned to every displayed element, see Fig. 4b. Attributes of graph elements may be manipulated with widgets as it is shown in Fig. 4c.

The object-oriented design uses several design patterns (Model View Controller, Prototype, and Builder), thus special requirements may be added easily. Due to a strict separation of internal data structures and graphical representation it is straightforward to extend Snoopy by a new graph class; see [13] for a demonstration how to do it. Fig. 3 gives the hierarchy of net classes which Snoopy currently supports.

5 Applications

Snoopy is in worldwide use for teaching (see, e.g., [16], [20]) and research (see, e.g., [8], [10], [11], [12], [14]); see Snoopy's website for more references. In the last year, Snoopy has been downloaded about 1,400 times.

Snoopy's coloured Petri nets have been applied to investigate a variety of large-scale biological systems, proving its capability to solve many challenges imposed by biological multi-scale modelling, e.g. repetition, variant, and organization of cells [17]. Case studies deploying coloured Petri nets usually require stochastic and/or continuous simulations over very large underlying uncoloured Petri nets; for specific case studies and related figures see [8].

6 Comparison with Other Tools

There is no tool on the market which supports a comparable family of Petri nets classes as Snoopy does. Usually, modelling tools confine themselves to a few net classes. Contrary, Snoopy provides a set of related net classes: time-free, stochastic, continuous, hybrid and their coloured extensions, as well as plenty of analysis techniques, e.g. built-in animation/simulation and export to external analysis tools. This provides an excellent approach to accomplish the analysis of a (biological) system from different perspectives by relating all these net classes.

Herein, we compare Snoopy with three popular tools providing similar functionalities for selected net classes: CPN Tools, GreatSPN, and Cell Illustrator.

CPN Tools [1] are tailored to coloured (timed) Petri nets. They established a landmark in modelling convenience. Their concept of fusion places inspired Snoopy's logical nodes, and the hierarchical organization of substitution transitions triggered Snoopy's coarse nodes. However, there are no special arcs, and CPN Tools do not explicitly support any of the quantitative net classes, which are mandatory for systems and synthetic biology, such as continuous, stochastic or hybrid Petri nets.

GreatSPN [3] supports modelling and analysis of $\mathcal{GSPN}$ and a coloured extension, but no other net classes. There are neither logical nodes nor hierarchy, but an interesting layer concept.

Cell Illustrator [19] is a commercially licensed software tool utilising Hybrid Functional Petri Nets with extensions (HFPNe) to model and simulate biochemical pathways. While Cell Illustrator combines discrete and continuous parts in one model, it does not offer the full interplay between continuous and stochastic transitions as it is given in Snoopy. Crucial features such as modifier edges and immediate or scheduled transitions are not supported. A model can only be simulated using static partitioning. Advanced modelling features like logical nodes, hierarchy, or colour – which are imperative when considering large scale models or models with repeated components – are not provided.

In summary, Snoopy's rich modelling capabilities make it competitive and particularly well suited for scenarios suggesting the simultaneous and consistent use of several modelling paradigms enabling different modelling abstractions.

7 Installation

Snoopy is available for Windows, Mac OS X and Linux. It can be obtained free of charge for academic use from its website `http://www-dssz.informatik.tu-cottbus.de/snoopy.html`. Installation packages contain all dependencies; no other libraries need to be manually installed. See Snoopy's website for more information how to install and use it on different platforms, for Petri net examples in Snoopy's proprietary file format, and for Snoopy's bibliography.

Snoopy comes with several further Petri net classes, including time(d) Petri nets and modulo Petri nets, as well as a couple of other graph types; see [13]. Snoopy is still evolving – we are open for suggestions.

Acknowledgement. Substantial contributions to Snoopy's development have been done by former staff members and numerous student projects at Brandenburg University of Technology, chair *Data Structures and Software Dependability.*

References

1. CPN Tools website, `http://cpntools.org/` (accessed: March 30, 2012)
2. Gecode website, `http://www.gecode.org/` (accessed: March 30, 2012)

3. GreatSPN website, http://www.di.unito.it/~greatspn/index.html (accessed: March 30, 2012)
4. MC2 website, http://www.brc.dcs.gla.ac.uk/software/mc2 (accessed: March 30, 2012)
5. OGDF - Open graph drawing framework website, http://www.ogdf.net/doku.php/start (accessed: March 30, 2012)
6. Wxwidgets website, http://www.wxwidgets.org (accessed: March 30, 2012)
7. Bornstein, B.J., Keating, S.M., Jouraku, A., Hucka, M.: LibSBML: an API library for SBML. Bioinformatics 24(6) (2008)
8. Gilbert, D., Heiner, M.: Petri nets for multiscale Systems Biology. Brunel University, Uxbridge/London (2011), http://multiscalepn.brunel.ac.uk/
9. Gillespie, D.T.: Exact stochastic simulation of coupled chemical reactions. Journal of Physical Chemistry 81(25), 2340–2361 (1977)
10. Heiner, M., Gilbert, D.: How Might Petri Nets Enhance Your Systems Biology Toolkit. In: Kristensen, L.M., Petrucci, L. (eds.) PETRI NETS 2011. LNCS, vol. 6709, pp. 17–37. Springer, Heidelberg (2011)
11. Heiner, M., Gilbert, D., Donaldson, R.: Petri Nets for Systems and Synthetic Biology. In: Bernardo, M., Degano, P., Zavattaro, G. (eds.) SFM 2008. LNCS, vol. 5016, pp. 215–264. Springer, Heidelberg (2008)
12. Heiner, M., Lehrack, S., Gilbert, D., Marwan, W.: Extended Stochastic Petri Nets for Model-Based Design of Wetlab Experiments. In: Priami, C., Back, R.-J., Petre, I. (eds.) Transactions on Computational Systems Biology XI. LNCS (LNBI), vol. 5750, pp. 138–163. Springer, Heidelberg (2009)
13. Heiner, M., Richter, R., Schwarick, M., Rohr, C.: Snoopy-a tool to design and execute graph-based formalisms. Petri Net Newsletter 74, 8–22 (2008)
14. Herajy, M., Heiner, M.: Hybrid representation and simulation of stiff biochemical networks through generalised hybrid Petri nets. Tech. Rep. 02–11, BTU Cottbus, Computer Science Institute (2011)
15. Hindmarsh, A., Brown, P., Grant, K., Lee, S., Serban, R., Shumaker, D., Woodward, C.: Sundials: Suite of nonlinear and differential/algebraic equation solvers. ACM Trans. Math. Softw. 31, 363–396 (2005)
16. Kafura, D., Tatar, D.: Initial experience with a computational thinking course for computer science students. In: Proc. SIGCSE 2011, pp. 251–256. ACM (2011)
17. Liu, F.: Colored Petri Nets for Systems Biology. Ph.D. thesis, BTU Cottbus, Computer Science Institute (January 2012)
18. Liu, F., Heiner, M., Rohr, C.: Manual for Colored Petri Nets in Snoopy. Tech. Rep. 02–12, BTU Cottbus, Computer Science Institute (March 2012)
19. Nagasaki, M., Saito, A., Jeong, E., Li, C., Kojima, K., Ikeda, E., Miyano, S.: Cell Illustrator 4.0: a Comp. Platform for Systems Biology. Silico Biology 10 (2010)
20. Petre, I.: Introduction to Computational and Systems Biology, Collection of Modelling Reports, Åbo Akademi, Department of IT (2011)
21. Petri Net Markup Language (PNML): Systems and software engineering – High-level Petri nets – Part 2: Transfer format, ISO/IEC 15909–2:2011 (2009)
22. Rohr, C., Marwan, W., Heiner, M.: Snoopy - a unifying Petri net framework to investigate biomolecular networks. Bioinformatics 26(7), 974–975 (2010)
23. Schwarick, M., Rohr, C., Heiner, M.: MARCIE - Model checking And Reachability analysis done effiCIEntly. In: Proc. QEST 2011. pp. 91–100 (2011)
24. Wegener, J., Schwarick, M., Heiner, M.: A Plugin System for Charlie. In: Proc. CS&P 2011, pp. 531–554. Białystok University of Technology (2011)

CPN Assistant II: A Tool for Management of Networked Simulations

Štefan Korečko, Ján Marcinčin, and Viliam Slodičák

Department of Computers and Informatics,
Faculty of Electrical Engineering and Informatics,
Technical University of Košice, Letná 9, 041 20 Košice, Slovakia
{stefan.korecko,viliam.slodicak}@tuke.sk, jano.marcincin@gmail.com

Abstract. The paper describes CPN Assistant II, a simulation management tool for CPN Tools software. CPN Assistant II cooperates with the version 3 of CPN Tools and allows to prepare, run and manage multiple simulation jobs in a networked environment. For each simulation job a net, a number of simulation runs and transition firings per run and a selection of monitors to be processed can be specified. The data acquired by each of the selected monitors during an execution of the simulation job are merged and stored as a text file at a specified location. If needed, the tool can also execute user-defined post-processing plug-ins to convert the text files to another format, compute some statistics or to perform other required tasks with the files.

Keywords: Coloured Petri nets, CPN Tools, CPN Assistant II, simulation management.

1 Introduction

The CPN Tools [7] software is a widely used and sophisticated tool for a design, analysis and simulation of coloured Petri net (CPN) models. Thanks to an incorporation of concepts such as time, randomness and monitors the tool can be also used for simulation-based performance analysis of discrete-event systems. The monitors are special sets of functions primarily used to collect data during simulations. One of such performance analysis tasks was an evaluation of various modifications [1] of a parallel raytracing implementation [2], [3] at the home institution of the authors, which required a large number of simulation runs, each with more than 260000 transition firings. It soon became obvious that this task cannot be performed "by hand", so we started to search for some simulation automation and management solution that fulfils the following requirements:

1. To be able to execute a desired number of simulation runs, each with a desired number of transition firings and to combine the data collected by monitors during the simulation runs.
2. To be able to run and manage multiple simulations in a networked environment.

As no available tool provided the required functionality, we developed a new tool, called *CPN Assistant* to be used with CPN Tools 2.2. This prototype tool uses a master-slave communication model and consists of two applications – CPN Assistant and CPN Assistant Node. The CPN Assistant serves as the master and provides simulation jobs preparation and assignment of the jobs to available slave nodes. The job preparation includes recording of a CPN from CPN Tools, setting a number of simulation runs and transition firings per run and selection of monitors to be processed. The net recording is a process of capturing a complete specification of the net when it is sent from the CPN Tools GUI to the CPN Tools simulator. Its implementation in CPN Assistant is inspired by a similar process of the *BRITNeY suite* software [4], [6]. The CPN Assistant Node (a slave) controls a simulation process on a given machine and collects data obtained by monitors of a simulated net. After the simulation job is finished the data are sent to the master. The CPN Assistant is briefly described in [1], together with simulation experiments performed with its aid.

In this paper we introduce a second version of the tool, called *CPN Assistant II*. Contrary to the prototype, the CPN Assistant II consists of only one application, which can be run in both master and slave mode. It also allows to run multiple simulations on a single machine and to post-process the data collected during the simulations by user-defined plug-ins. The presented version of CPN Assistant II has been designed and tested for cooperation with CPN Tools 3.2.2.

The rest of the paper is organized as follows: In section 2 we compare the tool with other extensions of CPN Tools, namely with *BRITNeY suite* and *Access/CPN* [5]. Section 3 describes the tool functionality from user's point of view and section 4 is devoted to its architecture and communication processes. Finally, we conclude with installation instructions and some remarks about a future development of the tool.

2 Comparison with Other Tools

As it was stated before, no tool has been found that offers the functionality implemented in CPN Assistant. The *CPN Tools* itself provides only so-called simulation replications, which allow to execute a specified number of simulation runs within a single instance of the tool.

The other tool considered was the *BRITNeY suite* [4], [6], which supports visualizations of CPN and other formal models and state-space analysis of CPN models. From our point of view its valuable features are a pluggable architecture [4] and a possibility to capture a communication between the CPN Tools GUI and simulator for later replay without a GUI assistance. The second feature allows basic simulation automation – even for CPNs that this tool cannot process (parse) itself. On the other hand, BRITNeY does not offer a management of multiple simulations and their outputs. We originally intended to re-implement CPN Assistant as a part of BRITNeY but its compatibility with new releases of CPN Tools has been dropped. Nevertheless, the BRITNeY suite has been important for our work as the net recording in CPN Assistant is based on the communication capturing process of BRITNeY.

Finally, the third software piece examined has been the *Access/CPN framework* [5]. It supports the development of extensions for CPN Tools and an integration of CPN models into external applications. It provides two interfaces to the CPN Tools simulator. The first one is an SML interface for efficient access to the simulator and its primary goal is to allow an implementation of new and more efficient analysis methods. The second one, a Java interface, offers an object-oriented representation of CPN models, an importer to load models from CPN Tools, and an implementation of a protocol for communication with the simulator [5]. The Access/CPN seems to be a good platform to implement an extension with the desired functionality, but for a long time, it lacked one important thing – a support of monitors. A work on the monitors support started only recently and the first note about it is from December 2011. When the monitors support will be finished, we will consider a re-implementation of CPN Assistant II as a part of the Access/CPN.

3 Functionality

CPN Assistant II is aimed at CPN Tools users that need to run a lot of simulation experiments with their CPN models. For them CPN Assistant II offers an automated management of simulation jobs execution – on a single computer or in a networked environment. CPN Tools and CPN Assistant II have to be installed on each computer utilized. Before the first use CPN Assistant II requires a single setup. It is also required to set the CPN Tools GUI to remote mode on a computer used for the CPN recording.

Preparation of simulation experiments consists of starting CPN Assistant II on each computer used and of a setup of simulation jobs (simulations). Setting up a simulation job includes selecting a net and specifying a number of simulation runs and transition firings per run. Of course, nets to be simulated have to be recorded first – to so-called replication files. Then the simulations themselves are handled by CPN Assistant II automatically - it distributes simulations to available computers (nodes), performs simulations by invoking the CPN Tools simulator and collects and post-processes data captured by monitors.

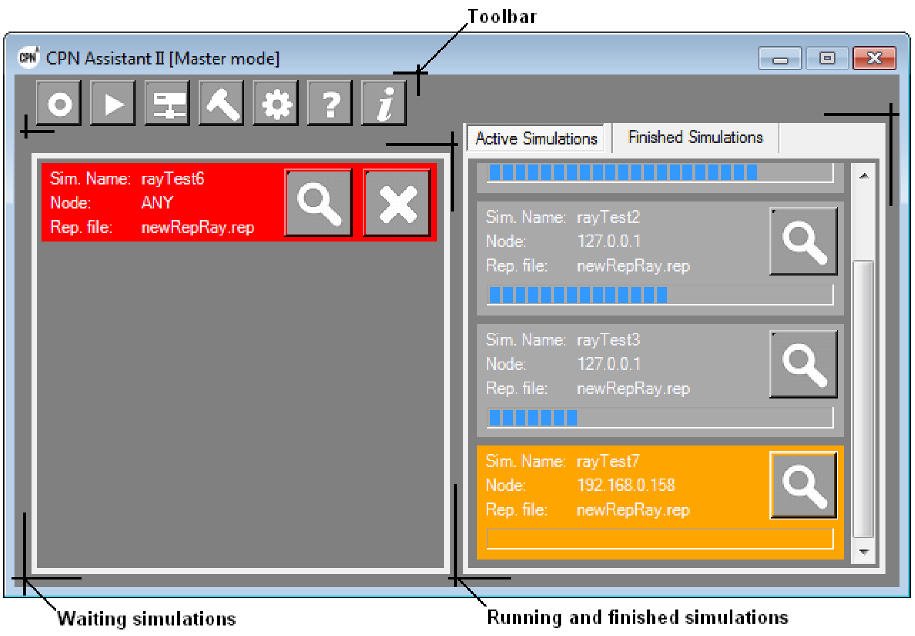
CPN Assistant II can work in three modes:

Slave mode (node mode). In this mode the tool is only able to perform simulations ordered by a master (i.e. the tool running in a master mode), collect results of simulations and send them back to the master. We call the tool running in this mode a slave or a simulation node.

Master mode. This mode provides the full functionality of the tool. The tool running in this mode, the master, can prepare simulations, distribute them to simulation nodes or execute them locally. All simulation results, captured by selected monitors, are sent from the slaves back to the master where they can be processed using the post-processing plug-ins. The tool running in the master mode broadcasts its presence in regular intervals allowing the slaves to identify and connect to the master without a need of manual specification and establishment of connection. Local execution of simulations can be disabled while in this mode.

Standalone mode. The idea behind this mode is to easily isolate the tool from the rest of a networked environment, where a presence of another instances of the tool can be expected. In fact, this mode is the same as the master mode without broadcasting its presence. So, only the slaves that know an IP address of the computer running the tool in this mode can connect to it.

A tool running in the slave mode can connect to the master in two ways. The first is an automatic connection of a simulation node after receiving a broadcast message from the master. The second way is to configure the node to ignore the broadcasts and specify the IP address of the master exactly. After setting up the address the node will try to connect to the specified master in regular intervals. With these settings, nodes can also connect to a tool instance running in the standalone mode. If the connection between the master and the simulation node breaks, the node reverts to listening for the master's presence broadcasts or to trying to connect to the specified master according to the node configuration.



Sim. Name: rayTest6
Node: ANY
Rep. file: newRepRay.rep

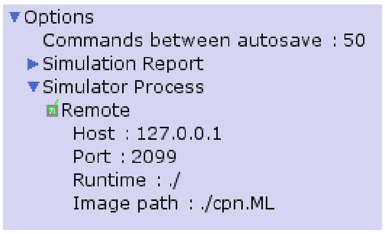
Sim. Name: rayTest2
Node: 127.0.0.1
Rep. file: newRepRay.rep

Sim. Name: rayTest3
Node: 127.0.0.1
Rep. file: newRepRay.rep

Sim. Name: rayTest7
Node: 192.168.0.158
Rep. file: newRepRay.rep

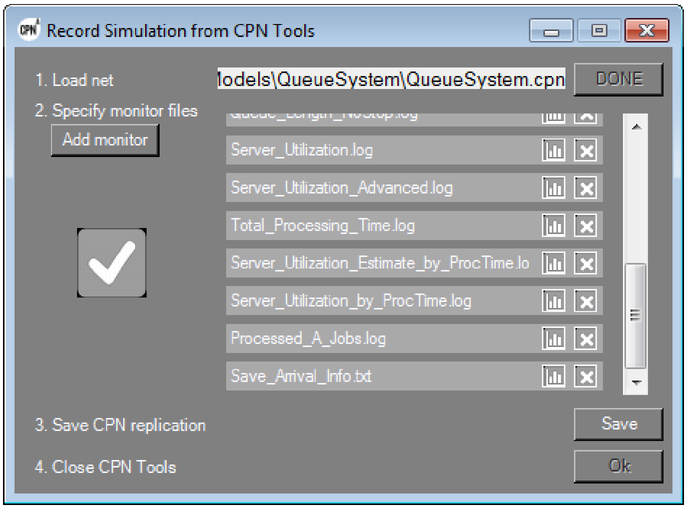
Fig. 1. CPN Assistant II main window

In Fig. 1 we can see a main window of CPN Assistant II running in the master mode. Its appearance in other modes is essentially the same. The window is divided into three parts – a toolbar, a queue of waiting simulation jobs and a list of running (active) and already finished simulations. The buttons on the toolbar allow to (from left to right) record a CPN, define and start a simulation, show available simulation nodes, show debug window, change application settings, show a user manual and display an about box. The first button is used to capture a net from CPN Tools and save it to a replication file. Before doing this, however, we have to be sure that CPN Tools on the master computer is set for a remote processing with port number 2099 (Fig. 2). After clicking the button a record simulation window is shown (Fig. 3) where we specify a net to load. Then CPN



```
▼Options
  Commands between autosave : 50
  ▶Simulation Report
  ▼Simulator Process
    ✓Remote
      Host : 127.0.0.1
      Port : 2099
      Runtime : ./
      Image path : ./cpn.ML
```

Fig. 2. CPN Tools settings for CPN capture by the master



Record Simulation from CPN Tools

1. Load net lodels\QueueSystem\QueueSystem.cpn DONE
2. Specify monitor files Add monitor
- Queue_Length_NoStop.log
- Server_Utilization.log
- Server_Utilization_Advanced.log
- Total_Processing_Time.log
- Server_Utilization_Estimate_by_ProcTime.lo
- Server_Utilization_by_ProcTime.log
- Processed_A_Jobs.log
- Save_Arrival_Info.txt

3. Save CPN replication Save
4. Close CPN Tools Ok

Fig. 3. CPN capture window

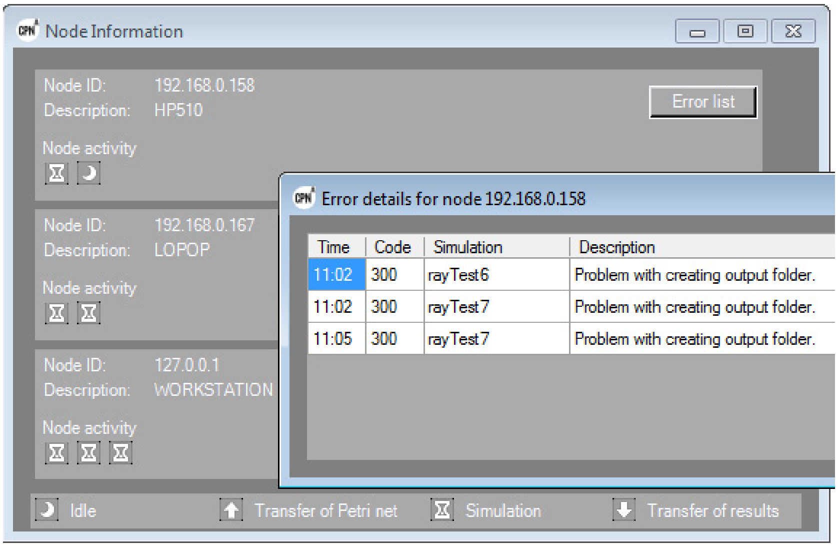
Assistant II invokes a CPN Tools instance and captures the net description from it. In addition, a list of monitor files is automatically populated by "write in file" and "data collector" monitors of the net while the net is being recorded. In the list the monitors are represented by names of files they create and fill during a simulation. When the recording process is finished a tick icon is shown on the left side and the net can be saved to a replication file. We can also change the monitor files list before saving - we can reduce the list or assign post-processing plug-ins to some monitors. Finally, we close the window and CPN Tools. The replication file can be used for multiple simulations. Re-capture of the net is required only if it is changed in CPN Tools. The monitors list can be also edited later – when creating a simulation job or while the job is waiting for execution.

When all required CPNs are saved as replication files we can prepare and run simulation jobs. This is done by clicking the second button on the toolbar and specifying simulation name, replication file, number of simulation runs and transition firings per run. Optionally, we can also select a specific node for the job execution. The simulation created is automatically added to the waiting simulations queue (Fig. 1). The simulation starts immediately after some (or the selected) simulation node become available. In the CPN Assistant GUI it is moved from the queue to the running (active) simulations list. Name, node IP address, replication file and progress are shown for each simulation. By clicking the button with magnifier icon we can see simulation details and hitting the cross button cancels the simulation.

Simulations that encounter problems during execution and crash are moved back to the queue of waiting simulations on the master. Their state is changed to paused and they are highlighted with a red background (rayTest6 in Fig. 1). Unlike "normal" simulations the crashed ones do not start automatically. They can be resumed only manually from the simulation details window where a user can also review the problems by checking an error list. Resumed simulations are highlighted with an orange background (rayTest7 in Fig. 1). It should be also noted that data collected before the crash can be recovered: When an error on a simulation node occurs and the simulation is cancelled the node creates a copy of simulation output folder with timestamp and job name as the folder name within its temporary folder. Error lists can be also viewed from a node perspective by clicking corresponding buttons in node information window (Fig. 4). The window is available via the third button on the toolbar.

The fourth button on the toolbar opens a debug window where various system messages are shown. The fifth one accesses an options window (Fig. 5), essential before the first use of the tool. The settings available include paths to CPN Tools GUI and simulator, folders for outputs of the monitors, the tool mode and network settings.

Three settings are dedicated to data collection and post-processing. The output folder is a folder on the master computer where data, collected by monitors during simulations, are stored. Each simulation has a separate subfolder with the same name as the simulation. The data (text files) are copied here from the corresponding slave temporary folder after the simulation is finished. The temporary folder is used during the simulation. Here the data obtained in an actual simulation run are appended to the data from previous runs of the same simulation job. The data are stored in text files (one file per monitor) with the same names and internal structure as the original files the monitors create and fill during the simulation. The post-processing plug-ins folder contains plug-ins for an additional processing of the text files. These plug-ins can be associated



Node ID: 192.168.0.158
Description: HP510

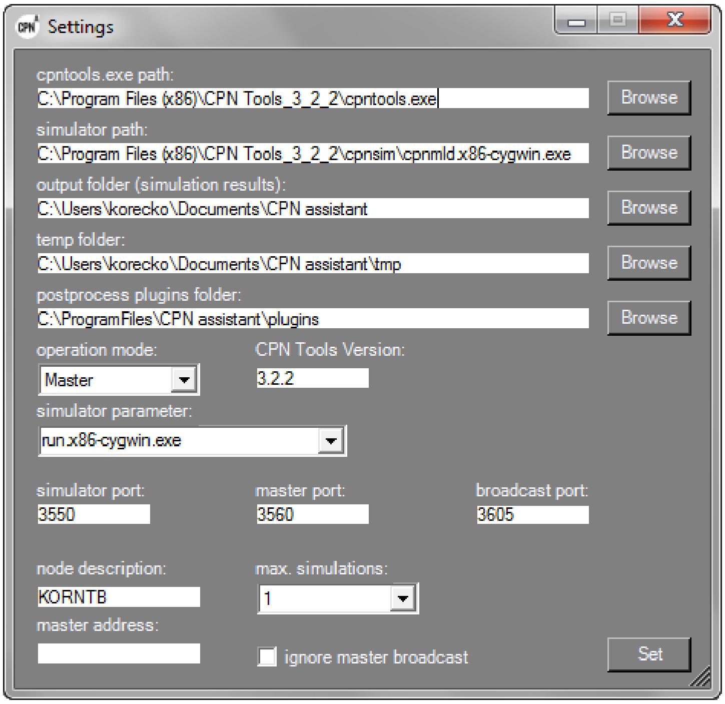
Node ID: 192.168.0.167
Description: LOPOP

Node ID: 127.0.0.1
Description: WORKSTATION

Error details for node 192.168.0.158

Time	Code	Simulation	Description
11:02	300	rayTest6	Problem with creating output folder.
11:02	300	rayTest7	Problem with creating output folder.
11:05	300	rayTest7	Problem with creating output folder.

Fig. 4. Node information window with error list



cpntools.exe path: C:\Program Files (x86)\CPN Tools_3_2_2\cpntools.exe
simulator path: C:\Program Files (x86)\CPN Tools_3_2_2\cpnsim\cpnmld.x86-cygwin.exe
output folder (simulation results): C:\Users\korecko\Documents\CPN assistant
temp folder: C:\Users\korecko\Documents\CPN assistant\tmp
postprocess plugins folder: C:\ProgramFiles\CPN assistant\plugins
operation mode: Master
CPN Tools Version: 3.2.2
simulator parameter: run.x86-cygwin.exe
simulator port: 3550
master port: 3560
broadcast port: 3605
node description: KORNTB
max. simulations: 1
master address:
☐ ignore master broadcast

Fig. 5. Options window with CPN Assistant II setup

with individual monitors of simulation jobs. This can be done everywhere where the monitor files list can be edited and also during simulation execution. Plug-in sample with source code is included with the tool.

4 Tool Architecture and Communication

The CPN Assistant II is written in the C# language and requires .NET framework 4 installed to work. TCP connections carrying custom remote procedure call protocol are used for communication between simulation nodes and the master.

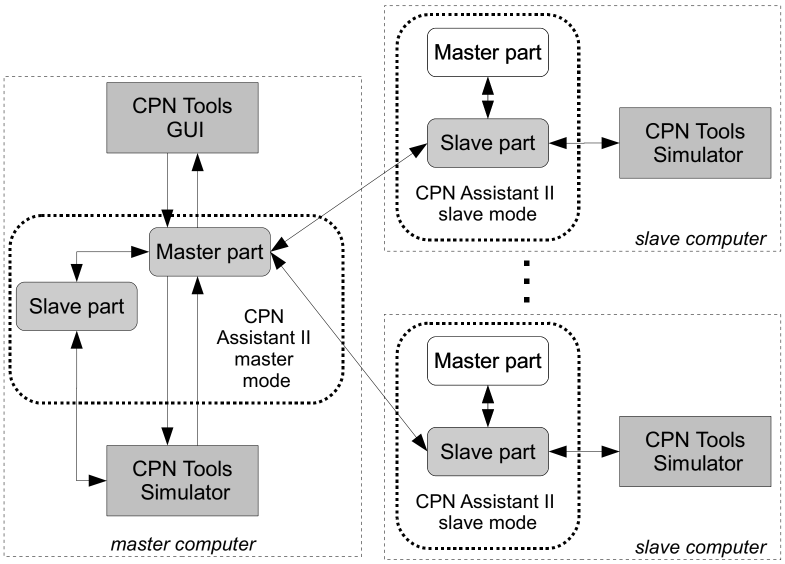


Fig. 6. CPN Assistant II instances in a master-slave configuration. Arrows are TCP connections.

The tool is internally divided into a master part and a slave part. Communication between the parts is implemented in the same way as between the tool in master and slave mode – by a TCP connection (Fig. 6).

When running, the slave part is always connected to the CPN Tools simulator, disconnecting only when an error occurs. The master part is connected to both the CPN Tools GUI and simulator, albeit to the GUI only during the net recording.

Lost connection is discovered by sending periodic keepalive packets from both sides. The packets sent by a simulation node carry state information for every simulation slot of the node. Number of simulation slots ("max. simulations" in Fig. 5) determines how many simulations can be executed at once on the node.

This number can be selected from range 1 to 16 in the slave mode or from 0 to 16 in the master mode.

The master's presence broadcasts are UDP datagrams sent to the entire subnet in fixed periods of 15 seconds. A number of port on which the master listens for nodes is a part of the broadcasted message, together with a 12 bytes long key to distinguish between the master's presence broadcasts and other services that may be sending broadcasts on the same UDP port. This means that nodes that do not ignore the master's broadcasts need only to know a broadcast port to be able to connect to the master. Connection port is supplied in the broadcasted messages.

When a simulation node receives a data about a new simulation job it examines a recorded CPN that is a part of the data and changes the output folder setting to the temporary folder of the node. Only after this the data are re-sent to the CPN Tools simulator for simulation run execution. During the simulation the node generates control packets for the simulator to set a number of transition firings, rewind a simulation run or start a run. After each simulation run monitor files specified to be collected are appended to text files, which are sent back to the master after the last run.

Both CPN Tools GUI and simulator are executed from CPN Assistant II. The simulator is invoked during start-up of the assistant and closed when the assistant is closed. CPN Tools GUI is invoked when recording a CPN. During the recording process a communication between CPN Tools GUI and simulator passes through CPN Assistant II, which acts as a proxy. The traffic from GUI is recorded before re-sending to the simulator. Simulator feedback traffic is directly forwarded to GUI without recording or inspection. For this mechanism to work, the CPN Tools GUI must be set to use the remote simulator process as shown in Fig. 2.

The post-processing plug-ins are dynamic libraries that are used to further process the text files containing data collected during simulations. Such a processing can be a conversion of the data to another format, for example a format of some spreadsheet processor where the data will be analysed. The plug-ins can be developed independently and have to implement a simple interface. The interface consists of a method `Process(string monitorPath)` and a single read-only property `Name` with the name of the plug-in. The `Process` method is invoked when the processing is needed and its argument (`monitorPath`) is a path to the text file to be processed. An exception catching is utilized in CPN Assistant II while executing the plug-ins to prevent an application crash caused by a bad plug-in implementation. To cope with a heavy load the plug-ins are executed in parallel in individual threads within a threadpool.

5 Conclusions

In this paper we introduced CPN Assistant II a simulation management tool for the CPN Tools software that allows to prepare, run and manage multiple simulation jobs in a networked environment. The tool is implemented in C#

language and requires .NET framework 4 or higher to run. It can be downloaded from [8]. To run the tool it is only required to unzip the downloaded archive.

As a future development of the tool, we plan several extensions. An important one could be a load balancing and priority based scheduler, where each node will be marked by a performance index, entered manually or based on some microbenchmark, and simulation jobs will have a priority value assigned. This should allow an out of order execution of simulations and optimization of node selection for specific simulations with very long simulation time or for simulations that have higher priority in obtaining results. Another modification could be a better analysis of captured CPN to allow further modifications during replaying (simulation) on a node. We also intend to add another method to the plug-ins interface, a method for a settings dialogue of a plug-in.

Acknowledgements. This work is the result of the project implementation: Development of the Center of Information and Communication Technologies for Knowledge Systems (ITMS project code: 26220120030) supported by the Research & Development Operational Program funded by the ERDF.

References

1. Korečko, Š., Sobota, B., Szabó, C.: Performance Analysis of Processes by Automated Simulation of Coloured Petri Nets. In: 10th International Conference on Intelligent Systems Design and Applications, ISDA 2010, Cairo, Egypt, pp. 176–181. IEEE (2010)
2. Sobota, B., Straka, M.: Interactive Rendering on Clusters in Virtual Reality System Proland. In: 6th International Scientific Conference Electronic Computer and Informatics, ECI 2004, pp. 457–462. VIENALA Press (2004)
3. Sobota, B., Perháč, J., Szabó, C., Schrötter, Š.: High-resolution visualisation in cluster environment. In: Grid Computing for Complex Problem, GCCP 2008, pp. 62–69. Institute of Informatics of Slovak Academy of Sciences (2008)
4. Westergaard, M.: The BRITNeY Suite: A Platform for Experiments, In: Seventh Workshop on Practical Use of Coloured Petri Nets and the CPN Tools, Arhus, Denmark (2006)
5. Westergaard, M., Kristensen, L.M.: The Access/CPN Framework: A Tool for Interacting with the CPN Tools Simulator. In: Franceschinis, G., Wolf, K. (eds.) PETRI NETS 2009. LNCS, vol. 5606, pp. 313–322. Springer, Heidelberg (2009)
6. 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)
7. CPN Tools homepage, http://cpntools.org/
8. CPN Assistant download page, https://sites.google.com/site/cpnassistant/

Author Index

Badouel, Eric 128
Bera, Debjyoti 328

Carmona, Josep 51
Cristini, Frédéric 348

Dong, Jin Song 92

Evangelista, Sami 248

Fahland, Dirk 168
Finkel, Alain 3

Gil-Costa, Veronica 288
Goubault-Larrecq, Jean 3

Halager, Anders E. 32
Hansen, Henri 208
Heiner, Monika 398
Heitmann, Frank 368
Herajy, Mostafa 398
Hoare, Tony 1

Inostrosa-Psijas, Alonso 288

Köhler-Bußmeier, Michael 368
Korečko, Štefan 408
Kounev, Samuel 388
Kristensen, Lars Michael 248

Lehmann, Andreas 228
Liu, Fei 398
Liu, Guan Jun 92
Liu, Yang 92
Lobos, Jair 288
Lohmann, Niels 228

Mailund, Thomas 32
Marcinčin, Ján 408
Marin, Mauricio 288
Martos-Salgado, María 108
Meier, Philipp 388

Nakatumba, Joyce 308

Prüfer, Robert 168

Rakow, Astrid 268
Rohr, Christian 398
Rosa-Velardo, Fernando 108

Schwarick, Martin 398
Slodičák, Viliam 408
Solé, Marc 51
Spinner, Simon 388
Sun, Jun 92

Tessier, Catherine 348

Valmari, Antti 208
van der Aalst, Wil M.P. 72, 308
van der Werf, Jan Martijn 148, 328
van Hee, Kees M. 328

Wang, Xu 188
Weidlich, Matthias 148
Westergaard, Michael 32, 308
Wolf, Karsten 228