Lars Michael Kristensen · Wojciech Penczek
Guest Editors

Transactions on **Petri Nets and Other Models of Concurrency XIII**

Maciej Koutny
Editor-in-Chief

Journal Subline

LNCS 11090

Springer

Maciej Koutny · Lars Michael Kristensen
Wojciech Penczek (Eds.)

Transactions on Petri Nets and Other Models of Concurrency XIII

Springer

Editor-in-Chief
Maciej Koutny
Newcastle University
Newcastle upon Tyne, UK

Guest Editors
Lars Michael Kristensen
Western Norway University of Applied Sciences
Bergen, Norway

Wojciech Penczek
Institute of Computer Science
Polish Academy of Sciences
Warsaw, Poland

ISSN 0302-9743 ISSN 1611-3349 (electronic)
Lecture Notes in Computer Science
ISSN 1867-7193 ISSN 1867-7746 (electronic)
Transactions on Petri Nets and Other Models of Concurrency
ISBN 978-3-662-58380-7 ISBN 978-3-662-58381-4 (eBook)
https://doi.org/10.1007/978-3-662-58381-4

Library of Congress Control Number: 2018961172

Preface by Editor-in-Chief

The 13th issue of LNCS *Transactions on Petri Nets and Other Models of Concurrency* (ToPNoC) contains revised and extended versions of a selection of the best papers from the workshops held at the 38th International Conference on Application and Theory of Petri Nets and Concurrency (Petri Nets 2017, Zaragoza, Spain, June 25–30, 2017), and the 17th International Conference on Application of Concurrency to System Design (ACSD 2017, Zaragoza, Spain, June 25–30, 2017).

I would like to thank the two guest editors of this special issue: Lars Michael Kristensen and Wojciech Penczek. Moreover, I would like to thank all authors, reviewers, and organizers of the Petri Nets 2017 and ACSD 2017 satellite workshops, without whom this issue of ToPNoC would not have been possible.

September 2018 Maciej Koutny

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

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

The annual International Conference on Application and Theory of Petri Nets and Concurrency started in 1980. For more information on the international Petri net community, see: http://www.informatik.uni-hamburg.de/TGI/PetriNets/.

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

ToPNoC contains:

– Revised versions of a selection of the best papers from workshops and tutorials concerned with Petri nets and concurrency
– Special issues related to particular subareas (similar to those published in the *Advances in Petri Nets* series)
– Other papers invited for publication in ToPNoC
– Papers submitted directly to ToPNoC by their authors

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

Topics

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

Also included are applications of nets to: biological systems; security systems; e-commerce and trading; embedded systems; environmental systems; flexible manufacturing systems; hardware structures; health and medical systems; office automation;

operations research; performance evaluation; programming languages; protocols and networks; railway networks; real-time systems; supervisory control; telecommunications; cyber physical systems; and workflow.

For more information about ToPNoC see: http://www.springer.com/gp/computer-science/lncs/lncs-transactions/petri-nets-and-other-models-of-concurrency-topnoc-/731240.

Submission of Manuscripts

Manuscripts should follow LNCS formatting guidelines, and should be submitted as PDF or zipped PostScript files to ToPNoC@ncl.ac.uk. All queries should be sent to the same e-mail address.

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

Preface by Guest Editors

This volume of ToPNoC contains revised versions of a selection of the best workshop papers presented at satellite events of the 38th International Conference on Application and Theory of Petri Nets and Other Models of Concurrency (Petri Nets 2017) and the 17th International Conference on Application of Concurrency to System Design (ACSD 2017).

As guest editors, we are indebted to the Program Committees of the workshops and in particular to their chairs. Without their enthusiastic support and assistance, this volume would not have been possible. The papers considered for this special issue have been selected in close cooperation with the workshop chairs. Members of the Program Committees participated in reviewing the new versions of the papers eventually submitted.

We received suggestions for papers for this special issue from:

- ATAED 2017: Workshop on Algorithms and Theories for the Analysis of Event Data (Chairs: Wil van der Aalst, Robin Bergenthum, Josep Carmona)
- PNSE 2017: International Workshop on Petri Nets and Software Engineering (Chairs: Lawrence Cabac, Daniel Moldt, Heiko Rölke).

The authors of the suggested papers were invited to improve and extend their results where possible, based on the comments received before and during the workshops. Each resulting revised submission was reviewed by at least two referees. We followed the principle of asking for fresh reviews of the revised papers, also from referees not involved initially in the reviewing of the original workshop contributions. All papers went through the standard two-stage journal reviewing process, and eventually eight were accepted after rigorous reviewing and revising. In addition, we invited the organizers of the eighth edition of the model checking contest to coordinate a paper reporting on the recent results and findings from the tool competition. The paper from the model checking contest was also subject to a two-stage review process and was selected for inclusion in this special volume.

The paper "Computing Alignments of Event Data and Process Models" by Sebastiaan van Zelst, Alfredo Bolt, and Boudewijn van Dongen considers the fundamental problem in process mining of checking conformance between a process model and an event log recorded from a system. The paper reports on large-scale experiments with process models aimed at investigating the impact that parameters of conformance checking algorithms have on the efficiency of computing optimal alignments. The paper concludes that the specific parameter configurations have a significant effect on computation efficiency.

Local process models describe structured fragments of process behavior that occurs in the context of business processes. The paper "Heuristic Mining Approaches for High-Utility Local Process Models" by Benjamin Dalmas, Niek Tax, and Sylvie Norre studies how a collection of process models that provide business insight can be generated. The paper proposes heuristics to prune the search space in high-utility local process model mining without significant loss of precision. The relation between event log properties and the effect of using the proposed heuristics in terms of search space and precision is also investigated.

The paper "On Stability of Regional Orthomodular Posets" by Luca Bernardinello, Carlo Ferigato, Lucia Pomello, and Adrian Puerto Aubel presents a fundamental study of so-called regional logics corresponding to the set of regions of a transition system ordered by set inclusion. The paper presents initial results related to stability of regional logics representing some first steps toward a full characterization of stability.

The variable ordering used in model checking based on binary decision diagrams is known to have a significant impact on verification performance. The paper "Decision Diagrams for Petri Nets: A Comparison of Variable Ordering Algorithms" by Elvio Gilberto Amparore, Susanna Donatelli, Marco Beccuti, Giulio Garbi, and Andrew Miner presents an extensive experimental comparison of static variable orderings in the context of Petri nets. The paper has led to new insight into fundamental properties exploited by existing variable orderings proposed in the literature.

Modeling complex systems requires a combination of techniques to facilitate multiple perspectives and adequate modeling. The paper "Model Synchronization and Concurrent Simulation of Multiple Formalisms Based on Reference Nets" by Pascale Möller, Michael Haustermann, David Mosteller, and Dennis Schmitz shows how multiple formalisms can be used together in their original representation without the transformation to a single formalism. The authors present an approach to transform modeling languages into Reference Nets, which can be executed with the simulation environment Renew. A finite automata modeling and simulation tool is given to showcase the application of the concept.

Web service composition represents a fundamental problem and its complexity depends on the restrictions assumed. The paper "Complexity Aspects of Web Services Composition" by Karima Ennaoui, Lhouari Nourine, and Farouk Toumani studies the impact of several parameters on the complexity of this problem. The authors show that the problem is EXPTIME-complete if there is a bound on either: the number of instances of services that can be used in a composition, or the number of instances of services that can be used in parallel, or the number of the hybrid states in the finite state machines representing the business protocols of existing services.

The paper "GPU Computations and Memory Access Model Based on Petri net" by Anna Gogolińska, Łukasz Mikulski, and Marcin Piątkowski presents a general and uniform GPU computation and memory access model based on bounded inhibitor Petri nets. The effectiveness of the model is demonstrated by comparing its throughput with practical computational experiments performed on the Nvidia GPU with the CUDA architecture. The accuracy of the model was tested with different kernels.

Testing of fault-tolerant distributed software systems is a challenging task. The paper "Model-Based Testing of the Gorums Framework for Fault-Tolerant Distributed Systems" by Rui Wang, Lars Michael Kristensen, Hein Meling, and Volker Stolz shows how colored Petri net models can be used for model-based test case generation. The authors concentrate on so-called quorum-based distributed systems, and experimentally demonstrate that test cases automatically obtained from colored Petri net models may lead to a high statement coverage.

The Model Checking Contest (MCC) is an annual competition aimed at providing a fair evaluation of software tools that verify concurrent systems using state-space exploration techniques and model checking. The paper “MCC 2017 – The Seventh Model Checking Contest” by Fabrice Kordon, Hubert Garavel, Lom Messan Hillah, Emmanuel Paviot-Adet, Loïg Jezequel, Francis Hulin-Hubard, Elvio Amparore, Marco Beccuti, Bernard Berthomieu, Hugues Evrard, Peter G. Jensen, Didier Le Botlan, Torsten Liebke, Jeroen Meijer, Jiří Srba, Yann Thierry-Mieg, Jaco van de Pol, and Karsten Wolf presents the principles and the results of the 2017 edition of the MCC, which took place along with the Petri Net and ACSD joint conferences.

As guest editors, we would like to thank all authors and referees who contributed to this issue. The quality of this volume is the result of the high scientific standard of their work. Moreover, we would like to acknowledge the excellent cooperation throughout the whole process that has made our work a pleasant task. We are also grateful to the Springer/ToPNoC team for the final production of this issue.

September 2018

Lars Michael Kristensen
Wojciech Penczek

Organization

Guest Editors

Lars Michael Kristensen	Bergen University College, Norway
Wojciech Penczek	Polish Academy of Sciences, Poland

Workshop Co-chairs

Wil van der Aalst	RWTH Aachen University, Germany
Robin Bergenthum	FernUniversität in Hagen, Germany
Josep Carmona	Universitat Politecnica de Catalunya, Spain
Lars Michael Kristensen	Bergen University College, Norway
Daniel Moldt	University of Hamburg, Germany
Heiko Rölke	DIPF, Germany

Reviewers

Étienne André
Seppe van den Broucke
Robin Bergenthum
Didier Buchs
Jörg Desel
Susanna Donatelli
Giuliana Franceschinis
Monica Heiner
Thomas Hildebrandt
Ekkart Kindler
Jetty Kleijn
Maciej Koutny
Michael Köhler-Bußmeier
Robert Lorenz
Giancarlo Mauri
Lucia Pomello
Hernan Ponce de Leon
Marcos Sepúlveda
Yann Thierry-Mieg
Karsten Wolf

Contents

Computing Alignments of Event Data and Process Models

Sebastiaan J. van Zelst(✉), Alfredo Bolt, and Boudewijn F. van Dongen

Department of Mathematics and Computer Science,
Eindhoven University of Technology,
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
{s.j.v.zelst,a.bolt,b.f.v.dongen}@tue.nl

Abstract. The aim of conformance checking is to assess whether a process model and event data, recorded in an event log, conform to each other. In recent years, alignments have proven extremely useful for calculating conformance statistics. Computing optimal alignments is equivalent to solving a shortest path problem on the state space of the synchronous product net of a process model and event data. State-of-the-art alignment based conformance checking implementations exploit the A^*-algorithm, a heuristic search method for shortest path problems, and include a wide range of parameters that likely influence their performance. In previous work, we presented a preliminary and exploratory analysis of the effect of these parameters. This paper extends the aforementioned work by means of large-scale statistically-sound experiments that describe the effects and trends of these parameters for different populations of process models. Our results show that, indeed, there exist parameter configurations that have a significant positive impact on alignment computation efficiency.

Keywords: Process mining · Conformance checking · Alignments

1 Introduction

Most organizations, in a variety of fields such as banking, insurance and healthcare, execute several different (business) processes. Modern information systems allow us to track, store and retrieve data related to the execution of such processes, in the form of *event logs*. Often, an organization has a global idea, or even a formal specification, of how the process is supposed to be executed. In other cases, laws and legislations dictate explicitly in what way a process is ought to be executed. Hence, it is in the company's interest to assess to what degree the execution of their processes is in line with the corresponding specification.

Conformance checking techniques, originating from the field of *process mining* [3], aim at solving the aforementioned problem. Conformance checking techniques allow us to quantify to what degree the actual execution of a process, as recorded in an event log, conforms to a corresponding process specification.

Recently, *alignments* were introduced [4,6], which rapidly developed into the de-facto standard in conformance checking. The major advantage of alignments w.r.t. alternative conformance checking techniques, is the fact that deviations and/or mismatches are quantified in an exact, unambiguous manner.

When computing alignments, we convert a given process model, together with the behaviour recorded in an event log, into a *synchronous product net* and subsequently solve a shortest path problem on its state space. Typically, the well known A^* algorithm [8] is used as an underlying solution to the shortest path problem. However, several (in some cases alignment-specific) parametrization options are defined and applied on top of the basic A^* solution.

In previous work [19] we presented a preliminary and exploratory analysis of the effect of several parameters on the conventional alignment algorithm. In this paper, we extend the aforementioned work further, by assessing a significantly larger population of process models. We specifically focus on those parametrizations of the basic approach that, in our previous work, have shown to have a positive impact on the algorithm's overall performance. Moreover, we present a concise algorithmic description of alignment calculation which explicitly includes these parameters. Our experiments confirm that, indeed, the parameters studied enable us to increase the overall efficiency of computing alignments.

The remainder of this paper is organized as follows. In Sect. 2, we present preliminaries. In Sect. 3, we present the basic A^*-based alignment algorithm. In Sect. 4, we evaluate the proposed parametrization. In Sect. 5, we discuss related work. Section 6 concludes the paper.

2 Preliminaries

In this section, we present preliminary concepts needed for a basic understanding of the paper. We assume the reader to be reasonably familiar with concepts such as functions, sets, bags, sequences and Petri nets.

2.1 Sets, Tuples, Sequences and Matrices

We denote the set of all possible multisets over set X as $\mathcal{B}(X)$. We denote the set of all possible sequences over set X as X^*. The empty sequence is denoted $\langle\rangle$. Concatenation of sequences σ_1 and σ_2 is denoted as $\sigma_1 \cdot \sigma_2$. Given tuple $\overline{x} = (x_1, x_2, \ldots, x_n)$ of Cartesian product $X_1 \times X_2 \times \ldots \times X_n$, we define $\pi_i(\overline{x}) = x_i$ for all $i \in \{1, 2, \ldots, n\}$. In case we have a tuple $\overline{t} \in X \times Y \times Z$, we have $\pi_1(\overline{t}) \in X$, $\pi_2(\overline{t}) \in Y$ and $\pi_3(\overline{t}) \in Z$. We overload notation and extend projection to sequences, i.e. given sequence $\overline{\sigma} \in (X_1 \times X_2 \times \ldots \times X_n)^*$ of length k where $\overline{\sigma} = \langle (x_1^1, x_2^1, \ldots, x_n^1), (x_1^2, x_2^2, \ldots, x_n^2), \ldots, (x_1^k, x_2^k, \ldots, x_n^k)\rangle$, we have $\pi_i(\overline{\sigma}) = \langle x_i^1, x_i^2, \ldots, x_i^k\rangle \in X_i^*$, for all $i \in \{1, 2, \ldots, n\}$. Given a sequence $\sigma = \langle x_1, x_2, \ldots, x_k\rangle \in X^*$ and a function $f : X \rightarrow Y$, we define $\pi^f : X^* \rightarrow Y^*$ with $\pi^f(\sigma) = \langle f(x_1), f(x_2), \ldots, f(x_k)\rangle$. Given $Y \subseteq X$ we define $\downarrow_Y : X^* \rightarrow Y^*$ recursively with $\downarrow_Y (\langle\rangle) = \langle\rangle$ and $\downarrow_Y (\langle x\rangle \cdot \sigma) = \langle x\rangle \cdot \downarrow_Y (\sigma)$ if $x \in Y$ and $\downarrow_Y (\langle x\rangle \cdot \sigma) = \downarrow_Y (\sigma)$ if $x \notin Y$. We write $\sigma_{\downarrow_Y}$ for $\downarrow_Y (\sigma)$. Given an $m \times n$ matrix

$\mathbf{A}$, i.e. $\mathbf{A}$ has m rows and n columns, $\mathbf{A}_{i,j}$ represents the element on row i and column j ($1 \leq i \leq m, 1 \leq j \leq n$). $\mathbf{A}^{\intercal}$ represents the transpose of $\mathbf{A}$. $\vec{x} \in \mathbb{R}^n$ denotes a column vector of length n, whereas $\vec{x}^{\intercal}$ represents a row vector.

2.2 Event Logs and Petri Nets

The execution of business processes within a company generates traces of event data in its supporting information systems. We are able to extract such data from these information systems, describing, for specific instances of the process, e.g. an insurance claim, what sequence of activities has been performed over time. We refer to a collection of such event data as an *event log*. A sequence of executed process activities, related to a process instance, is referred to as a *trace*. Consider Table 1, which depicts a simplified view of an event log.

The event log describes the execution of activities related to a phone repair process. For example, consider all events related to case 1216, i.e. a new *defect is registered* by *Abdul*, *Maggy* subsequently *repairs* the phone, *Harry informs* the corresponding *client*, etc. When we consider all events executed for case *1216*, ordered by time-stamp, we observe that it generates the sequence of activities $\langle a, b, d, e, g \rangle$ (note that we use short-hand activity names for simplicity). Such projection, i.e. merely focussing on the sequential ordering of activities, is also referred to as the *control-flow perspective*. In the remainder of this paper we assume this perspective, which we formalize in Definition 1.

Definition 1 (Event Log). *Let $\mathcal{A}$ denote the universe of activities. An event log L is a multiset of sequences over activities in $\mathcal{A}$, i.e. $L \in \mathcal{B}(\mathcal{A}^*)$.*

Each sequence $\sigma \in L$ describes a *trace*, which potentially occurs multiple times in an event log.

Table 1. Example event log fragment.

Event-id	Case-id	Activity	Resource	Time-stamp
...	...	...	...	
12474	*1215*	*test (e)*	*John*	*2017-11-14 14:45*
12475	*1216*	*register defect (a)*	*Abdul*	*2017-11-14 15:12*
12476	*1215*	*order replacement (h)*	*Maggy*	*2017-11-14 15:14*
12477	*1216*	*repair (b)*	*Maggy*	*2017-11-14 15:31*
12478	*1216*	*inform client (d)*	*Harry*	*2017-11-14 15:40*
12479	*1216*	*test (e)*	*Maggy*	*2017-11-14 14:49*
12480	*1216*	*return to client (g)*	*Maggy*	*2017-11-14 16:01*
12481	*1217*	*register defect (a)*	*John*	*2017-11-14 16:03*
...	...	...	...	...

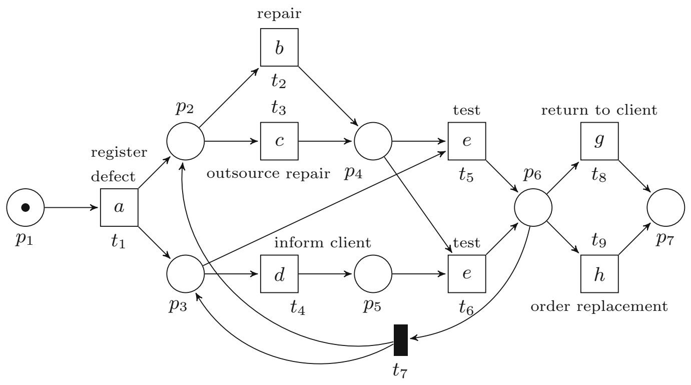


Fig. 1. Example labelled Petri net $N_1 = (P_1, T_1, F_1, \lambda_1)$ describing a (simplified) phone-repair process.

Event logs describe the actual execution of business processes, i.e. as recorded within a company's information system. Process models on the other hand allow us to describe the intended behaviour of a process. In this paper we use Petri nets [15] as a process modelling notation. An example Petri net is depicted in Fig. 1. The Petri net, like the event log, describes a process related to phone repair. It dictates that first a *register defect* activity needs to be performed. After this, a *repair* needs to be performed. Such repair is alternatively *outsourced*. In parallel with the repair, the client is optionally *informed* about the status of the repair. In any case, after the repair is completed, the repaired phone is *tested*. If the test succeeds the phone is *returned to the client*. If the test fails, either a new repair is performed, or, a *replacement* is ordered.

A Petri net is simply a bipartite graph with a set of vertices called *places* and a set of vertices called *transitions*. Places, depicted as circles, represent the state of the process. Transitions, depicted as boxes, represent executable actions, i.e. activities. A place can be *marked* by one ore more *tokens* which are graphically represented by black dots, depicted inside of the place, e.g. place p_1 is marked with one token in Fig. 1. If all places connected to a transition, by means of an *ingoing arc into the transition*, contain a token, we are able to fire the transition, i.e. equivalent with executing the activity represented by the transition. In such case, the transition consumes a token for each incoming arc, and produces a token for each of its outgoing arcs, e.g. transition t_1 is enabled in Fig. 1. After firing transition t_1, the token in p_1 is removed and both place p_2 and place p_3 contain a token. In this paper we assume that each transition has an associated (possibly unobservable) label which represents the corresponding activity, e.g. the label of transition t_1 is *register defect* (or simply a in short-hand notation) whereas transition t_7 is unobservable.

Definition 2 (Labelled Petri net). *Let P denote a set of places, let T denote a set of transitions and let $F \subseteq (P \times T) \cup (T \times P)$ denote the flow relation. Let*

Σ denote the universe of labels, let $\tau \notin \Sigma$ denote the unobservable label and let $\lambda \colon T \to \Sigma \cup \{\tau\}$. A labelled Petri net is a quadruple $N = (P, T, F, \lambda)$.

Observe that N_1 in Fig. 1, in terms of Definition 2, is described as $N_1 = (P_1 = \{p_1, \ldots, p_7\}, T_1 = \{t_1, t_2, \ldots, t_9\}, F_1 = \{(p_1, t_1), \ldots, (t_9, p_7)\}, \lambda_1 = \{\lambda_1(t_1) = a, \lambda_1(t_2) = b, \ldots, \lambda_1(t_9) = h\})$. Additionally observe that $\lambda_1(t_7) = \tau$, which is graphically visualized by the black solid fill of transition t_7.

Given an element $x \in P \cup T$, we define $\bullet x = \{y \in P \cup T \mid (y, x) \in F\}$ and $x\bullet = \{y \in P \cup T \mid (x, y) \in F\}$. A marking m of Petri net $N = (P, T, F, \lambda)$ is a multiset of P, i.e. $m \in \mathcal{B}(P)$. Given such marking and a Petri net, we write (N, m), which we refer to as a *marked net*. The initial marking of N is denoted as m_i, and thus, (N, m_i) represents the *initially marked net*. When a transition t is enabled in a marking m, i.e. $\forall p \in \bullet t(m(p) > 0)$, we write $(N, m)[t\rangle$, e.g. $(N_1, [p_1])[t_1\rangle$. Firing an enabled transition t in marking m, yielding $m' = (m - \bullet t) \uplus t\bullet$, is written as $(N, m) \xrightarrow{t} (N, m')$. If firing a sequence of transitions $\sigma = \langle t_1, t_2, \ldots, t_n\rangle \in T^*$, starts in marking m and yields marking m', i.e. $(N, m) \xrightarrow{t_1} (N, m_1) \xrightarrow{t_2} \ldots (N, m_{n-1}) \xrightarrow{t_n} (N, m')$, we write $(N, m) \xrightarrow{\sigma} (N, m')$. The set of all reachable markings from marking m is denoted $\mathcal{R}(N, m) = \{m' \subseteq \mathcal{B}(P) \mid \exists \sigma \in T^*((N, m) \xrightarrow{\sigma} (N, m'))\}$. Given a designated marking m and target marking m', we let $\mathcal{L}(N, m, m') = \{\sigma \in T^* \mid (N, m) \xrightarrow{\sigma} (N, m')\}$. For example, $\langle t_1, t_2, t_5, t_8\rangle \in \mathcal{L}(N_1, [p_1], [p_7])$. In case we are interested in the sequence of activities described by a firing sequence σ we apply $(\pi^{\lambda}(\sigma))_{\downarrow_\Sigma}$, e.g. $(\pi^{\lambda_1}(\langle t_1, t_2, t_5, t_7, t_3, t_4, t_6, t_8\rangle))_{\downarrow_\Sigma} = \langle a, b, e, c, d, e, g\rangle$.

In the remainder, we let $\mathbf{A}$ denote the incidence matrix of a (labelled) Petri net $N = (P, T, F, \lambda)$. $\mathbf{A}$ is an $|T| \times |P|$ matrix where $\mathbf{A}_{i,j} = 1$ if $p_j \in t_i \bullet \setminus \bullet\, t_i$, $\mathbf{A}_{i,j} = -1$ if $p_j \in \bullet t_i \setminus t_i\bullet$ and $\mathbf{A}_{i,j} = 0$ otherwise. Further more, given some marking $m \in \mathcal{B}(P)$, we write $\vec{m}$ to denote a $|P|$-sized column vector with $\vec{m}(i) = m(p_i)$ for $1 \leq i \leq |P|$.

2.3 Alignments

The example event log and Petri net, presented in Table 1 and Fig. 1 respectively, are both related to a simplified phone repair process. In case of our example trace related to case *1216*, i.e. $\langle a, b, d, e, g\rangle$, it is easy to see that there exists a $\sigma \in T_1^*$ s.t. $(\pi^{\lambda_1}(\sigma))_{\downarrow_\Sigma} = \langle a, b, d, e, g\rangle$, i.e. $\sigma = \langle t_1, t_2, t_4, t_6, t_8\rangle$. In practice however, such direct mapping between observed activities and transition firings in a Petri net is often not possible. In some cases, activities are not executed whereas the model specifies they are supposed to. Similarly, in some cases we observe activities that according to the model are not possible, at least at that specific point within the trace. Such mismatches are for example caused by employees deviating from the process as specified, e.g. activities are executed twice or mandatory activities are skipped. Moreover, in many cases the process specification is not exactly in line with the actual execution of the process, i.e. some aspects of the process are overlooked when designing the process specification.

Alignments allow us to compare the behaviour recorded in an event log with the behaviour as described by a Petri net. Conceptually, an alignment represents

$$\gamma_1 : \begin{array}{||c|c|c|c|c|c|} a & d & d & \gg & e & g \\ \hline t_1 & t_4 & \gg & t_2 & t_6 & t_8 \end{array} \quad \gamma_2 : \begin{array}{||c|c|c|c|c|c|} a & \gg & d & d & e & g \\ \hline t_1 & t_3 & \gg & t_4 & t_6 & t_8 \end{array} \quad \gamma_3 : \begin{array}{||c|c|c|c|c|c|c|c|c|} a & \gg & d & \gg & \gg & \gg & d & e & g \\ \hline t_1 & t_2 & t_4 & t_6 & t_7 & t_3 & t_4 & t_6 & t_8 \end{array}$$

Fig. 2. Example alignments for $\langle a, d, d, e, g \rangle$ and N_1.

a mapping between the activities observed in a trace $\sigma \in L$ and the execution of transitions in the Petri net. As an example, consider trace $\langle a, d, d, e, g \rangle$ and reconsider Petri net N_1 in Fig. 1. The trace does not fit the behaviour described by N_1. In Fig. 2 we present three different alignments of $\langle a, d, d, e, g \rangle$ and N_1. The first alignment, i.e. γ_1, specifies that the execution of the second d-activity is abundant, and, that an activity described by transition t_2 is missing. Similarly, the second alignment, i.e. γ_2, specifies that an activity described by transition t_3 is missing and that the first execution of the d-activity is abundant. Alignment γ_3 specifies that we are able to map each activity observed in the trace to a transition in the model, however, in such case, we at least miss activities described by transitions t_2, t_3 and t_6. Note that we do not miss an activity related to the execution of transition t_7, as this is an invisible transition.

When ignoring the $\gg$-symbols, the top row of each alignment equals the given trace, i.e. $\langle a, d, d, e, g \rangle$. The bottom row of each alignment, again when ignoring the $\gg$-symbols, represents a firing sequence in the language of N_1, i.e. $\sigma \in \mathcal{L}(N_1, [p_1], [p_7])$. Each individual column is referred to as a *move*. A move of the form $|\frac{a}{\gg}|$ is called a *log move* and represents behaviour observed in the trace that is not mapped onto the model. A move of the form $|\frac{\gg}{t}|$ is called a *model move* and represents behaviour that according to the model should have happened, yet was not observed at that position in the trace. A move of the form $|\frac{a}{t}|$ is called a *synchronous move* and represents an observed activity that is also described by the model at that position in the trace.

Definition 3 (Alignment). *Let $\sigma \in \mathcal{A}^*$ be a trace. Let $N = (P, T, F, \lambda)$ be a labelled Petri net and let $m_i, m_f \in \mathcal{B}(P)$ denote N's initial and final marking. Let $\gg \notin \mathcal{A} \cup T$. A sequence $\gamma \in ((\mathcal{A} \cup \{\gg\}) \times (T \cup \{\gg\}))^*$ is an alignment if:*

1. *$(\pi_1(\gamma))_{\downarrow_\mathcal{A}} = \sigma$; event part equals σ.*
2. *$(N, m_i) \xrightarrow{(\pi_2(\gamma))_{\downarrow_T}} (N, m_f)$; transition part is in the Petri net's language.*
3. *$\forall (a, t) \in \gamma (a \neq \gg \vee t \neq \gg)$; $(\gg, \gg)$ is not valid in an alignment.*

We let $\Gamma(N, \sigma, m_i, m_f)$ denote the set of all possible alignments of Petri net N and trace σ given markings m_i and m_f.

As exemplified by the three alignments of $\langle a, d, d, e, g \rangle$ and N_1, a multitude of alignments exists for a given trace and model. Hence, we need a means to be able to rank and compare alignments in such way that we are able to express our preference of an alignment w.r.t. other alignments. For example, in Fig. 2, we prefer γ_1 and γ_2 over γ_3, as we need less $\gg$-symbols to explain the observed behaviour in terms of the model. Computing such preference is performed by means of minimizing a cost function defined over the possible moves of an alignment. We present a general definition of such cost function in Definition 4, after which we provide a commonly used corresponding instantiation.

Definition 4 (Alignment Cost). *Let $\sigma \in \mathcal{A}^*$, let $N = (P, T, F, \lambda)$ be a labelled Petri net with $m_i, m_f \in \mathcal{B}(P)$, let $\gg \notin \mathcal{A} \cup T$ and let $c\colon (\mathcal{A} \cup \{\gg\}) \times (T \cup \{\gg\}) \rightarrow \mathbb{R}_{\geq 0}$. Given alignment $\gamma \in \Gamma(N, \sigma, m_i, m_f)$, the costs κ^c of γ, given move cost function c, is defined as $\kappa^c(\gamma) = \sum_{i=1}^{|\gamma|} c(\gamma(i))$. Finally, we let $\gamma^*_c \in$* $\mathbf{arg\,min}_{\gamma \in \Gamma(N,\sigma,m_i,m_f)} \kappa^c(\gamma)$ *denote the optimal alignment.*

The cost of an alignment is defined as the sum of the cost of each move within the alignment, as specified by cost function c. If it is clear from context what cost function c is used, we omit it from the cost related notation, i.e. we write κ, γ^* etc. Note that γ^* is an alignment that has minimum costs amongst all alignments of a given model and trace, i.e. an *optimal alignment*. In general one can opt to use an arbitrary instantiation of c, however, a cost function that is used quite often is the following *unit-cost function*:

1. $c(a, t) = 0 \Leftrightarrow a \in \mathcal{A}, t \in T$ and $\lambda(t) = a$ or $a = \gg$ and $\lambda(t) = \tau$.[1]
2. $c(a, t) = \infty \Leftrightarrow a \in \mathcal{A}, t \in T$ and $\lambda(t) \neq a$
3. $c(a, t) = 1$ otherwise

Using the unit-cost function, γ_1 and γ_2 of Fig. 2 are both optimal for $\langle a, d, d, e, g \rangle$ and N_1, i.e. both alignments have cost 2. This shows that *optimality is not guaranteed to be unique for alignments.*

3 Computing Optimal Alignments

In this section we present the basic alignment computation algorithm. The algorithm, in essence, is a modification of the A^* algorithm [8], i.e. a general purpose shortest path algorithm. We do however incorporate alignment-specific optimizations within the algorithm that have shown to be beneficial for the overall performance of the approach, i.e. in terms of search efficiency and memory usage [19]. The algorithm applies a shortest path search on the state-space of the *synchronous product net* of the given trace and Petri net. As such, we first present how such synchronous product net is constructed, after which we present the alignment algorithm.

3.1 Constructing the Synchronous Product Net

To find an optimal alignment, i.e. an alignment that minimizes the cost function of choice, we solve a shortest path problem defined on the state space of the synchronous product net of the given trace and model. Such synchronous product

[1] In some cases, if the absence of *token-generators* is not guaranteed, we use $c(a, t) = \epsilon$, where ϵ is a positive real number smaller than 1 and close to 0.

net encodes the trace as a sequential Petri net and integrates it with the original model. As such, each transition in the synchronous product net represents a move within the resulting alignment. Executing such transition corresponds to putting the corresponding move in the alignment. Consider Fig. 3, which depicts the synchronous product net of trace $\langle a, d, d, e, g\rangle$ and example Petri net N_1.

The sequence of black transitions, depicted on the top of the synchronous product net represents the input trace, i.e. $\langle a, d, d, e, g\rangle$. The labels of these transitions represent log moves, e.g. transition $(t_1, \gg)$ has label $(a, \gg)$. Observe that, we are able to, from the initial marking $[p_1, p'_1]$, generate a firing sequence $\langle(a, \gg), (d, \gg), \ldots, (g, \gg)\rangle$ (projected onto labels) marking $[p'_1, p_6]$. Such firing sequence corresponds to a sequence of log moves which describe the given trace. The lower part of the synchronous product net represents Petri net N_1, however, the transition names represent model moves, e.g. transition $(\gg, t'_1)$ directly relates to a model move on t_1 in N_1. Observe that, using these transitions we are able to generate firing sequences of model moves that correspond to firing sequences that are in N_1's language. Finally, the middle (grey) transitions manipulate both the marking of the top part of the synchronous product net as the bottom part. Each of these transitions represents a synchronous move, e.g. consider transition (t_1, t'_1) representing a synchronous move of the first event of $\langle a, d, d, e, g\rangle$, i.e. representing activity a, and transition t'_1 (i.e. identified as t_1 in Fig. 1).

We formally define a synchronous product net as the product of a Petri net that represents the input trace (i.e., a trace net), together with the given

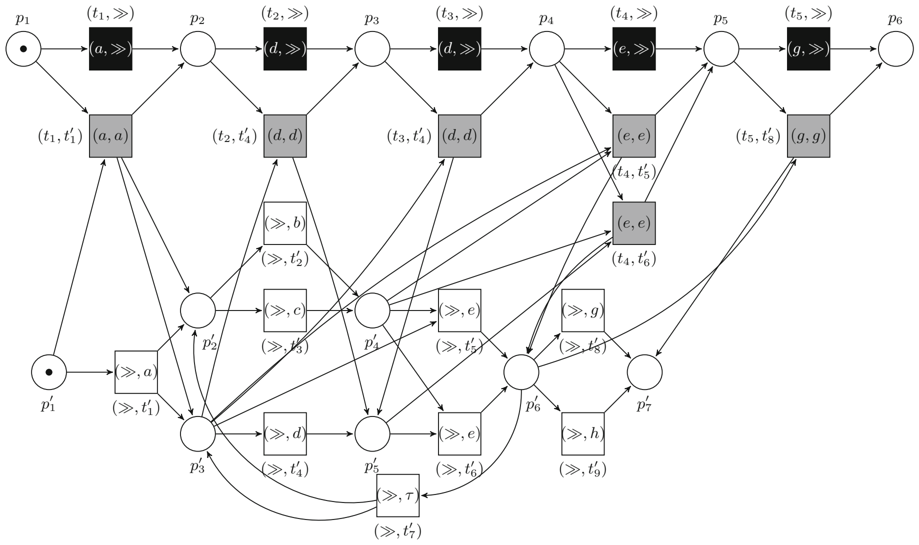


Fig. 3. Synchronous product net N_1^S of trace $\langle a, d, d, e, g\rangle$ and example Petri net N_1. Note that we have renamed elements of N_1 using a $'$-symbol, i.e. p'_1, t'_1 etc.

process model. As such, we first define a *trace net*, after which we provide a general definition of the product of two Petri nets.

Definition 5 (Trace net). *Let $\sigma \in \mathcal{A}^*$ be a trace. We define the trace net of σ as a labelled Petri net $N = (P, T, F, \lambda)$, where:*

- $P = \{p_i \mid 1 \leq i \leq |\sigma| + 1\}$.
- $T = \{t_i \mid 1 \leq i \leq |\sigma|\}$.
- $F = \{(p_i, t_i) \mid 1 \leq i \leq |\sigma| \land p_i \in P \land t_i \in T\} \cup \{(t_i, p_{i+1}) \mid 1 \leq i \leq |\sigma| \land p_{i+1} \in P \land t_i \in T\}$.
- $\lambda(t_i) = \sigma(i)$, *for* $1 \leq i \leq |\sigma|$.

Given a trace σ we write N^σ to refer to the trace net of σ. We subsequently define the product of two arbitrary labelled Petri nets.

Definition 6 (Petri net Product). *Let $N = (P, T, F, \lambda)$ and $N' = (P', T', F', \lambda')$ be two Petri nets (where $P \cap P' = \emptyset$ and $T \cap T' = \emptyset$). The product of N and N', i.e. Petri net $N \otimes N' = (P^\otimes, T^\otimes, F^\otimes, \lambda^\otimes)$ where:*

- $P^\otimes = P \cup P'$.
- $T^\otimes = (T \times \{\gg\}) \cup (\{\gg\} \times T') \cup \{(t, t') \in T \times T' \mid \lambda(t) = \lambda'(t')\}$
- $F^\otimes = \{(p, (t, t')) \in P^\otimes \times T^\otimes \mid (p, t) \in F \lor (p, t') \in F')\} \cup \{((t, t'), p) \in T^\otimes \times P^\otimes \mid (t, p) \in F \lor (t', p) \in F'\}$.
- $\lambda^\otimes : T^\otimes \to (\Sigma \cup \{\tau\} \cup \{\gg\}) \times (\Sigma \cup \{\tau\} \cup \{\gg\})$ *(assuming* $\gg \notin \Sigma \cup \{\tau\}$*) where:*
  $\lambda^\otimes(t, \gg) = (\lambda(t), \gg)$ *for* $t \in T$
  $\lambda^\otimes(\gg, t') = (\gg, \lambda'(t'))$ *for* $t' \in T'$
  $\lambda^\otimes(t, t') = (\lambda(t), \lambda'(t'))$ *for* $t \in T, t' \in T'$.

A *synchronous product net* is defined as the product of a trace net N^σ and an arbitrary Petri net N, i.e. $N^\sigma \otimes N$. Assume we construct such synchronous product net $N^S = (P^S, T^S, F^S, \lambda^S)$ based on a trace net $N^\sigma = (P^\sigma, T^\sigma, F^\sigma, \lambda^\sigma)$ of trace σ and Petri net $N = (P, T, F, \lambda)$. Moreover, let $p_i \in P^\sigma$ with $\bullet p_i = \emptyset$, $p_f \in P^\sigma$ with $p_f \bullet = \emptyset$, and, let m_i, m_f denote a designated initial and final marking of N. Furthermore, let $m_i^S = m_i \uplus [p_i]$ and $m_f^S = m_f \uplus [p_f]$. Any firing sequence $\sigma' \in (T^S)^*$, s.t. $m_i^S \xrightarrow{\sigma'} m_f^S$ corresponds to an alignment of σ and N [6]. To be able to compute an alignment based on a synchronous product net, such firing sequence needs to exist. The problem of determining whether such sequence exists is known as the *reachability problem*, which is shown to be decidable [10,13]. However, within conformance checking, we assume that a reference model of a process is designed by a human business process analyst/designer. We therefore assume that a process model has a certain level of quality, e.g. the Petri net is a *sound workflow net* [1, Definition 7]. In context of alignment computation we therefore simply assume that a Petri net N, given initial marking m_i and final marking m_f is *easy sound*, i.e. $\mathcal{L}(N, m_i, m_f) \neq \emptyset$. A synchronous product net of a trace net and an easy sound Petri net is, by construction, easy sound, and thus guarantees reachability of its final marking.

To derive the actual move related to each transition in some firing sequence $\sigma' \in \mathcal{L}(N^S, m_i^S, m_f^S)$, we utilize the label function of the synchronous product net. In case we observe a transition of the form $(\gg, t)$, i.e. with $t \in T^\sigma$, we know it relates to a log move, which is obtained by applying $\lambda^S((t, \gg))$, e.g. $\lambda((t_1, \gg)) = (a, \gg)$ in Fig. 3. In case we observe a transition of the form $(\gg, t)$ $t \in T$, we know it relates to a model move, which is reflected by the transition name, i.e. we do not need to fetch the transition's label, e.g. $(\gg, t_1')$ in Fig. 3. A transition of the form $(t, t') \in T^S$, i.e. with $t \neq \gg$ and $t' \neq \gg$ corresponds to a synchronous move, which we translate into such move by applying $(\lambda^\sigma(t), t')$, e.g. $(\lambda^\sigma(t_1), t_1') = (a, t_1')$ in Fig. 3. Since we are able to map each transition in the synchronous product net onto a corresponding move, we are also able to deduce the move costs corresponding to any such transition present in the synchronous product net. *Therefore, in the remainder, given a synchronous product net $N^S = (P^S, T^S, F^S, \lambda^S)$, we assume the existence of a corresponding transition-based move cost function $c^S \colon T^S \to \mathbb{R}_{\geq 0}$ that maps each transition in the synchronous product net to the costs of the underlying move it represents.*

3.2 Searching for Optimal Alignments

In this section, we present the state-of-the art algorithm for optimal alignment computation. We first present an informal overview of the A^*-algorithm. Subsequently we describe how to exploit the marking equation for the purpose of heuristic estimation, after which we show how to limit the number of states enqueued during the search. Finally, we present a concise corresponding algorithmic description.

Applying A^*. Each transition in the synchronous product net corresponds to a move in an alignment, and moreover, to an arc in the state space of the synchronous product. Since each move/transition has an associated cost, we are able to assign the weight of each arc in the net's state space with the cost of the associated move. For example, observe Fig. 4, in which we schematically depict a (small) part of the state-space of N_1^S (Fig. 3). The initial state of the state space, i.e. $[p_1, p_1']$, is depicted on the top-left. We are able to fire $(t_1, \gg)$, corresponding to a log-move $|\frac{a}{\gg}|$, yielding marking $[p_2, p_1']$. Similarly, in $[p_1, p_1']$, we are able to fire $(\gg, t_1')$, corresponding to model move $|\frac{\gg}{t_1}|$, yielding marking $[p_1, p_2', p_3']$. The order of firing these two transitions is irrelevant, i.e. $[p_1, p_1'] \xrightarrow{\langle (t_1, \gg), (\gg, t_1') \rangle} [p_2, p_2', p_3']$, and, $[p_1, p_1'] \xrightarrow{\langle (\gg, t_1'), (t_1, \gg) \rangle} [p_2, p_2', p_3']$.[2] Observe that we are also able to mark $[p_2, p_2', p_3']$ by firing (t_1, t_1') in marking $[p_1, p_1']$, corresponding to synchronous move $|\frac{a}{t_1'}|$. From $[p_2, p_2', p_3']$ we are able to fire $(t_2, \gg)$, (t_2, t_4'), $(\gg, t_2')$, $(\gg, t_3')$, and $(\gg, t_4')$ (not all of these transitions are explicitly visualized for the ease of readability/simplicity).

As indicated, each transition corresponds to a move, which, according to the corresponding cost function c^S has an associated cost. As such, the goal of

[2] The label $[p_2, p_2', p_3']$ is not shown in Fig. 4, it corresponds to the state on the second row and second column.

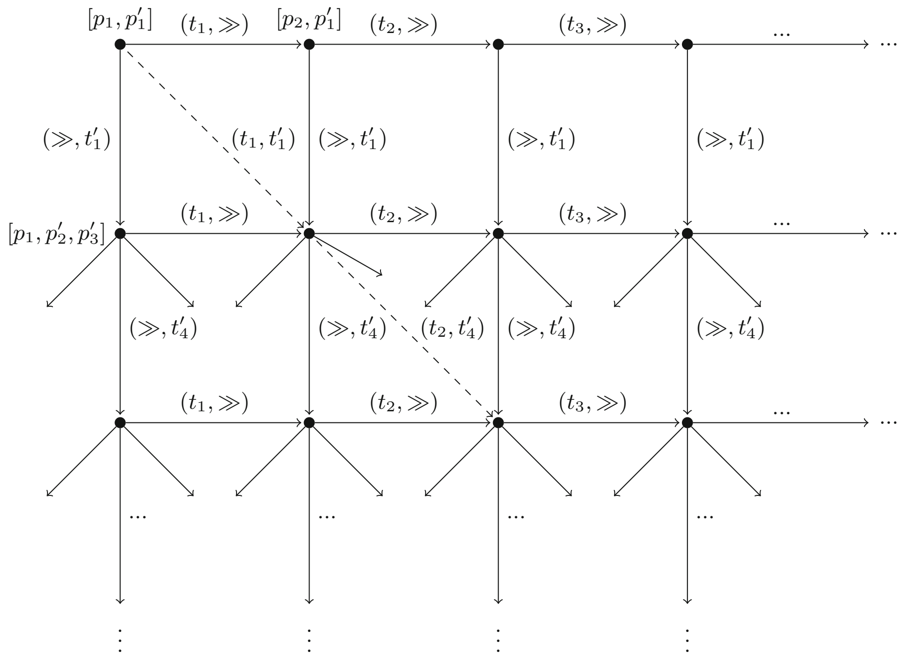


Fig. 4. Part of the state-space of the synchronous product net N_1^S shown in Fig. 3. Observe that in some markings, more transitions are enabled than we explicitly show here, e.g. in marking $[p_1, p'_2, p'_3]$, transitions $(\gg, t'_2)$ and $(\gg, t'_3)$are additionally enabled.

finding an optimal alignment is equivalent to solving a shortest path problem on the state space of the synchronous product net [6]. Within the given shortest path problem, the initial marking of the given Petri net combined with the first place of the trace net defines the initial state (i.e. m_i^S). Similarly the target state is a combination of the given final marking of the model combined with the last place of the trace net (i.e. m_f^S).

Many algorithms exist that solve a shortest path problem on a weighted graph with a unique start vertex and a set of end vertices. In this paper we predominantly focus on the A^* algorithm [8]. The A^* algorithm is an *informed search algorithm*, i.e. it tries to incorporate specific knowledge of the graph within the search. In particular, it uses a *heuristic function* that approximates, for each vertex in the given graph, the expected remaining distance to the closest end vertex. The A^* algorithm is *admissible*, i.e. it guarantees to find a shortest path, if the heuristic always underestimates the actual distance to the/any final state. In case of computing optimal alignments based on the state space of the synchronous product net, markings of the synchronous product net represent vertices. Hence, we formally define a heuristic function on the basis of arbitrary labelled Petri nets, after which we provide an instantiation tied to synchronous product nets.

Definition 7 (Petri net based heuristic function). *Let* $N = (P, T, F, \lambda)$ *be a Petri net. A heuristic function* h^N *is a function* $h^N : \mathcal{B}(P) \times \mathcal{B}(P) \rightarrow \mathbb{R}_{\geq 0}$.

Using the previously defined heuristic, we are able to, given an initial marking m_i^S and final marking m_f^S, of a synchronous product net $N^S = (P^S, T^S, F^S, \lambda^S)$, apply the default A^* approach, which roughly performs the following steps.

1. Inspect marking m that minimizes $f(m) = g(m) + h^{N^S}(m, m_f^S)$, where $g(m)$ is the *actual distance* from m_i^S to m (note that $g(m_i^S) = 0$).
2. For each adjacent marking m', i.e. $\exists t \in T^S(m \xrightarrow{t} m')$, compute $h^{N^S}(m', m_f^S)$. Furthermore, $\forall t \in T^S(m \xrightarrow{t} m')$, we apply $g(m') \leftarrow \mathbf{min}(g(m'), g(m) + c^S(t))$ (initially $g(m) = \infty, \forall m \in \mathcal{R}(N, m_i^S) \setminus m_i^S$).

Initially, marking m_i^S is the only known marking with a g-value unequal to ∞, i.e. $g(m_i^S) = 0$. Thus, starting with m_i^S, we repeat the two aforementioned steps until either we end up at m_f^S, or, no more markings are to be assessed. Due to the easy-soundness assumption we are guaranteed to always arrive, at some point, at m_f^S. Moreover, admissibility implies that the first time we assess marking m_f^S, $g(m_f^S)$ represents the shortest path from m_i^S to m_f^S in terms of move costs, and thus corresponding alignment costs. In general, it is possible to visit a marking m multiple times in *step 2*, potentially leading to a lower $g(m)$-value. However, if a heuristic function is *consistent*, i.e. for markings m, m', m'' and transition $t \in T^S$ s.t. $(N^S, m) \xrightarrow{t} (N^S, m')$, we have $h^{N^S}(m, m'') \leq h^{N^S}(m', m'') + c^S(t)$, we are guaranteed that once we reach a vertex during the A^* search, we are not able to reach it using an alternative path with lower costs than the current path. Hence, in case the heuristic function used is consistent, we know that once we inspect a marking m in *step 1*, $g(m)$ is minimal. As a consequence, whenever we reach it again in *step 2*, we are allowed to ignore it.

Exploiting the State Equation. In this paper we provide an instantiation of the heuristic function, i.e. h^{N^S}, that exploits the *state equation* of Petri nets, i.e. an algebraic expression of marking changes in a Petri net. Let $\vec{x}$ denote at $|T|$-sized column vector of integers, let m and m' denote two markings and let $\sigma \in T^*$ s.t. $(N, m) \xrightarrow{\sigma} (N, m')$ and let $\vec{m}$ and $\vec{m}'$ denote the corresponding $|P|$-sized marking column vectors. The state equation states that when we instantiate $\vec{x}$ as the *Parikh vector* of σ, i.e. if transition t_i occurs k times in σ then $\vec{x}(i) = k$, then $\vec{x}$ is a solution to $\vec{m}' = \vec{m} + \mathbf{A}^\intercal \vec{x}$. The reverse does however not hold, i.e. if we find a solution to $\vec{m}' = \vec{m} + \mathbf{A}^\intercal \vec{x}$, such solution $\vec{x}$ is not necessarily a Parikh representation of a $\sigma' \in T^*$ s.t. $(N, m) \xrightarrow{\sigma'} (N, m')$.

Nonetheless, we utilize the state equation for the purpose of calculating a Petri net based heuristic function. Given a marking m and target marking m' within the synchronous product net, we try to find a solution to $\vec{m}' = \vec{m} + \mathbf{A}^\intercal \vec{x}$, where $\mathbf{A}$ and $\vec{x}$ are defined in terms of the synchronous product net. Moreover, such solution needs to minimize the corresponding alignment cost, i.e. recall that for $1 \leq i \leq |T^S|$, $\vec{x}(i)$ refers to a transition $t_i \in T^S$ which has an associated cost as defined by the transition-based move cost function $c^S(t_i)$.

Definition 8 (State equation based heuristic). *Let $\sigma \in \mathcal{A}^*$ be a trace and let $N = (P, T, F, \lambda)$ be a Petri net. Let $N^S = N^\sigma \otimes N = (P^S, T^S, F^S, \lambda^S)$ be the synchronous product net of σ and N. Let $\mathbf{A}$ denote the incidence matrix of N^S, let $m, m' \in \mathcal{B}(P^S)$, and let $\vec{m}, \vec{m}'$ be the corresponding $|P^S|$-sized vectors. Let $c^S : T^S \to \mathbb{R}_{\geq 0}$ be the transition-based move cost function and let $\vec{c}$ denote a corresponding $|T^S|$-sized vector with $\vec{c}(i) = c^S(t_i)$ (for $t_i \in T^S$). Let $\vec{x} \in \mathbb{R}_{\geq 0}^{|T^S|}$ be a $|T^S|$-sized vector. We instantiate h^{N^S} (Definition 7) with $h^{N^S}(m, m') = \infty$ if no solution exists to $\vec{m}' = \vec{m} + \mathbf{A}^\intercal \vec{x}$, and otherwise:*

$$\boldsymbol{min}(\vec{c}^\intercal \vec{x} \mid \vec{m}' = \vec{m} + \mathbf{A}^\intercal \vec{x})$$

Observe that we define $\vec{x}$ as a vector containing non-negative real valued numbers ($\mathbb{R}_{\geq 0}$) rather than naturals ($\mathbb{N}$). Note that this potentially leads to fractional values in $\vec{x}$. However, since we aim at underestimating the true distance to the final marking this is acceptable, i.e. the vector does not need to correspond to an actual firing sequence. We are thus able to compute the state equation based heuristic by formulating and subsequently solving it as either a Linear Programming- (LP) or an Integer Linear Programming (ILP) problem [17].[3] Observe that, in case no solution to the (I)LP exists, we simply assign a value of ∞ to the heuristic. Since an (I)LP solution is always smaller or equal to the true costs of reaching target marking m' from marking m, the state equation based heuristic is admissible. As shown in [6], the heuristic is also *consistent*.

In step 2 of the A^* approach, we compute the heuristic $h^{N^S}(m', m_f^S)$ as defined in Definition 8 by solving an (Integer) Linear Programming problem. Observe that, as exemplified earlier, there are often multiple ways to arrive at a certain marking within the state space of the synchronous product net. To avoid solving the same (I)LP multiple times, once we have computed $h^{N^S}(m', m_f^S)$ we are able to store the solution value for m' in a temporary cache, and, remove it when we fetch m' in step 1. However, specifically in case of solving an Integer Linear Programming problem, computing the $h^{N^S}(m', m_f^S)$ is potentially time consuming. As it turns out, in some cases, the solution vector $\vec{x}$ of the (I)LP solved for marking m allows us to derive, for an adjacent marking m', i.e. $\exists t \in T^{N^S}(m \xrightarrow{t} m')$, an exact value for $h^{N^S}(m', m_f^S)$.

Conceptually, given that we assess some marking m, this works as follows. When we compute $h^{N^S}(m, m_f^S)$, given that it is not equal to ∞, we obtain an associated solution vector $\vec{x}$. Such vector essentially describes the number of times a transition is ought to be fired to reach m_f from m, even though there does not necessarily exists a corresponding firing sequence containing the exact number of transition firings as described by $\vec{x}$. Assume that from m we are able to traverse an edge related to firing a transition t_i, for which $\vec{x}(i) \geq 1$, yielding marking m'. In such case, we are guaranteed, as we show in Proposition 1, that the solution value for $h^{N^S}(m', m_f^S)$ equals $h^{N^S}(m, m_f^S) - c^S(t_i)$, i.e. we are able to subtract the cost of the move represented by t_i from $h^{N^S}(m, m_f^S)$. Even in

[3] In case we solve an ILP, we enforce $\vec{x} \in \mathbb{N}^{|T^S|}$.

the case that $\vec{x}(i) < 1$, we are able to devise a lower bound for the value of $h^{N^S}(m', m_f^S)$, as we show in Proposition 2.

Proposition 1 (State based heuristic provides exact solution). *Let* $\mathbf{A}$ *denote the incidence matrix of a synchronous product net* $N^S = (P^S, T^S, F^S, \lambda^S)$ *and let* $m, m_f \in \mathcal{B}(P^S)$. *Let* $c^S \colon T^S \to \mathbb{R}_{\geq 0}$ *be the transition-based move cost function and let* $\vec{c} \in \mathbb{R}_{\geq 0}^{|T^S|}$ *with* $\vec{c}(i) = c^S(t_i)$ *for* $t_i \in T^S$. *Let* $\vec{x}^* \in \mathbf{arg\,min}_{\vec{x} \in \mathbb{R}_{\geq 0}^{|T^S|}}(\vec{c}^\intercal \vec{x} \mid \vec{m_f} = \vec{m} + \mathbf{A}^\intercal \vec{x})$. *Let* $m' \in \mathcal{B}(P^S)$ *and let* $t_i \in T^S$ *s.t.* $(N^S, m) \xrightarrow{t_i} (N^S, m')$. *If* $\vec{x}^*(i) \geq 1$, *then* $\vec{c}^\intercal(\vec{x}^* - \vec{1}_{t_i}) = \boldsymbol{min}_{\vec{x} \in \mathbb{R}_{\geq 0}^{|T^S|}}(\vec{c}^\intercal \vec{x} \mid \vec{m_f} = \vec{m'} + \mathbf{A}^\intercal \vec{x})$.

Proof. Observe that, according to the state equation, $\vec{m'} = \vec{m} + \mathbf{A}^\intercal \vec{1}_{t_i}$, *and thus,* $\vec{m} = \vec{m'} - \mathbf{A}^\intercal \vec{1}_{t_i}$. *From this, we deduce* $\vec{m_f} = \vec{m} + \mathbf{A}^\intercal \vec{x}^* = \vec{m'} - \mathbf{A}^\intercal \vec{1}_{t_i} + \mathbf{A}^\intercal \vec{x}^* = \vec{m'} + \mathbf{A}^\intercal(\vec{x}^* - \vec{1}_{t_i})$, *i.e.* $\vec{x}^* - \vec{1}_{t_i}$ *is a solution to* $\vec{m_f} = \vec{m'} + \mathbf{A}^\intercal \vec{x}$.

Assume $\vec{c}^\intercal(\vec{x}^* - \vec{1}_{t_i}) > \boldsymbol{min}_{\vec{x} \in \mathbb{R}_{\geq 0}^{|T^S|}}(\vec{c}^\intercal \vec{x} \mid \vec{m_f} = \vec{m'} + \mathbf{A}^\intercal \vec{x})$, *which implies that there exists an alternative minimal solution for* $\vec{m_f} = \vec{m'} + \mathbf{A}^\intercal \vec{x}$, *i.e.* $\exists \vec{y} \in \mathbf{arg\,min}_{\vec{x} \in \mathbb{R}_{\geq 0}^{|T^S|}}(\vec{c}^\intercal \vec{x} \mid \vec{m_f} = \vec{m'} + \mathbf{A}^\intercal \vec{x})$ *with* $\vec{c}^\intercal \vec{y} < \vec{c}^\intercal(\vec{x}^* - \vec{1}_{t_i})$.

Again by using the fact that $\vec{m'} = \vec{m} + \mathbf{A}^\intercal \vec{1}_{t_i}$, *we observe that since* $\vec{y}$ *is a solution to* $\vec{m_f} = \vec{m'} + \mathbf{A}^\intercal \vec{x}$, *also* $(\vec{y} + \vec{1}_{t_i})$ *is a solution to* $\vec{m_f} = \vec{m} + \mathbf{A}^\intercal \vec{x}$. *This however contradicts minimality of* $\vec{x}^*$ *since* $\vec{c}^\intercal \vec{y} < \vec{c}^\intercal(\vec{x}^* - \vec{1}_{t_i}) \implies \vec{c}^\intercal(\vec{y} + \vec{1}_{t_i}) < \vec{c}^\intercal \vec{x}^*$. $\square$

Proposition 1 shows that if we compute a heuristic value for $h^{N^S}(m, m_f^S)$ formed by underlying variable assignment $\vec{x}^*$, then in case there exists some $t_i \in T^S$ with $\vec{x}^*(i) \geq 1$ and $m \xrightarrow{t_i} m'$, we are guaranteed that $h^{N^S}(m', m_f^S) = h^{N^S}(m, m_f^S) - \vec{c}(i) = h^{N^S}(m, m_f^S) - c^S(t_i)$. This effectively allows us to reduce the number of (I)LP's we need to solve. It is however also possible that there is some $t_j \in T^S$ with $\vec{x}^*(j) < 1$. In such case $\vec{x}^* - \vec{1}_{t_j}$ is not a solution to $\vec{m_f} = \vec{m'} + \mathbf{A}^\intercal \vec{x}$, it does however provide a lower bound on the actual heuristic value of m'.

Proposition 2 (State based heuristic provides an upper bound). *Let* $\mathbf{A}$ *denote the incidence matrix of a synchronous product net* $N^S = (P^S, T^S, F^S, \lambda^S)$ *and let* $m, m_f \in \mathcal{B}(P^S)$. *Let* $c^S \colon T^S \to \mathbb{R}_{\geq 0}$ *be the transition-based move cost function and let* $\vec{c} \in \mathbb{R}_{\geq 0}^{|T^S|}$ *with* $\vec{c}(i) = c^S(t_i)$ *for* $t_i \in T^S$. *Let* $\vec{x}^* \in \mathbf{arg\,min}_{\vec{x} \in \mathbb{R}_{\geq 0}^{|T^S|}}(\vec{c}^\intercal \vec{x} \mid \vec{m_f} = \vec{m} + \mathbf{A}^\intercal \vec{x})$. *Let* $m' \in \mathcal{B}(P^S)$ *and let* $t_i \in T^S$ *s.t.* $(N^S, m) \xrightarrow{t_i} (N^S, m')$. *If* $\vec{x}^*(i) < 1$, *then* $\vec{c}^\intercal(\vec{x}^* - \vec{1}_{t_i}) \leq \boldsymbol{min}_{\vec{x} \in \mathbb{R}_{\geq 0}^{|T^S|}}(\vec{c}^\intercal \vec{x} \mid \vec{m_f} = \vec{m'} + \mathbf{A}^\intercal \vec{x})$.

Proof. Assume there is a minimal solution $\vec{y}$ *to* $\vec{m_f} = \vec{m'} + \mathbf{A}^\intercal \vec{x}$, *i.e.* $\exists \vec{y} \in \mathbf{arg\,min}_{\vec{x} \in \mathbb{R}_{\geq 0}^{|T^S|}}(\vec{c}^\intercal \vec{x} \mid \vec{m_f} = \vec{m'} + \mathbf{A}^\intercal \vec{x})$ *s.t.* $\vec{c}^\intercal \vec{y} < \vec{c}^\intercal(\vec{x}^* - \vec{1}_{t_i})$. *Since* $(\vec{y} +$

$\vec{1}_{t_i}$) *is a solution to* $\vec{m_f} = \vec{m} + \mathbf{A}^\intercal \vec{x}$ *(cf. Proposition* 1*), this contradicts* $\vec{x}^* \in \mathbf{arg\ min}_{\vec{x} \in \mathbb{R}_{\geq 0}^{|T^S|}} (\vec{c}^\intercal \vec{x} \mid \vec{m_f} = \vec{m} + \mathbf{A}^\intercal \vec{x})$ *since* $\vec{c}^\intercal(\vec{y} + \vec{1}_{t_i}) < \vec{c}^\intercal \vec{x}^*$. □

If Proposition 2 applies, we know that $h^{N^S}(m', m_f^S) \geq \vec{c}^\intercal(\vec{x}^* - \vec{1}_{t_i})$. Thus, $\vec{c}^\intercal(\vec{x}^* - \vec{1}_{t_i})$ underestimates the true value of $h^{N^S}(m', m_f^S)$ and we write $\hat{h}^{N^S}(m', m_f^S) = \vec{c}^\intercal(\vec{x}^* - \vec{1}_{t_i})$. Whenever we derive $\hat{h}^{N^S}(m', m_f^S)$, we know $g(m')$, i.e. we compute $\hat{h}^{N^S}(m', m_f^S)$ in step 2 of the basic A^* approach. Thus, we are also able to derive an underestimating $f(m')$ value, i.e. $\hat{f}(m') = g(m') + \hat{h}^{N^S}(m', m_f^S)$. In practice this implies that instead of solving an (I)LP when we investigate a new marking, we just deduce the f-value, which is potentially an underestimate. In case it is an underestimate, we keep track of this, and whenever we, in step 1, inspect an element with a minimal underestimated f-value, we try to find an exact solution by solving an (I)LP. In such case it is possible that $f(m) > \hat{f}(m)$, in which we need to select a new marking in step 1 that minimizes the f-value.

Limiting Transition Ordering. Reconsider the synchronous product net shown in Fig. 3 with initial marking $[p_1, p'_1]$. Recall that there are three firing sequences in the net to achieve marking $[p_2, p'_2, p'_3]$, i.e. $\langle(\gg, t'_1), (t_1, \gg)\rangle$, $\langle(t_1, \gg), (\gg, t'_1)\rangle$ and $\langle(t_1, t'_1)\rangle$. Under the unit-cost function, the cost associated with $\langle(t_1, t'_1)\rangle$ is 0 whereas the cost for $\langle(\gg, t'_1), (t_1, \gg)\rangle$ and $\langle(t_1, \gg), (\gg, t'_1)\rangle$ is 2. We observe that both possible permutations of the sequence containing $(t_1, \gg)$ and $(\gg, t'_1)$ have the same cost and are both part of an (sub-optimal) alignment.

In general, assume we have an alignment $\gamma \cdot \langle x, y\rangle \cdot \gamma' \in \Gamma(N, \sigma, m_i, m_f)$ s.t. x is a log move and y is a model move. Moreover, let t_x denote the transition in the underlying synchronous product related to x, and let t_y denote the transition related to move y. Since, by construction, $\bullet t_x \cap \bullet t_y = \emptyset$, $\bullet t_x \cap t_y \bullet = \emptyset$, $t_x \bullet \cap \bullet t_y = \emptyset$ and $t_x \bullet \cap t_y \bullet = \emptyset$, we trivially deduce that also $\gamma \cdot \langle y, x\rangle \cdot \gamma' \in \Gamma(N, \sigma, m_i, m_f)$. Additionally, we have $\kappa(\gamma \cdot \langle x, y\rangle \cdot \gamma') = \kappa(\gamma \cdot \langle x, y\rangle \cdot \gamma')$, i.e. alignment costs are order independent.

Hence, to find an optimal alignment we only need to traverse/inspect one specific permutation of such log/model move combinations, rather than all possible permutations. In step 2 of the basic A^* scheme, each enabled transition in marking m is investigated. However, we are able to limit this number of transitions by exploiting the previously mentioned property, i.e.:

- *Log move restriction;* If the transition leading to the current marking relates to a *model move* we only consider those transitions t that relate to a *model* or *synchronous move.*
- *Model move restriction;* If the transition leading to the current marking relates to a *log move* we only consider those transitions t that relate to a *log* or *synchronous move.*

In the first option we are not able to schedule a log move after a model move. The other way around is however possible, i.e. we are allowed to schedule a model move after a log move. The second option behaves exactly opposite, i.e. we are not allowed to schedule a model move after a log move. Note that during the search we either only apply log move restriction, or, model move restriction, i.e. these techniques cannot be mixed.

Algorithmic Description. In Algorithm 1, we present the basic algorithm for optimal alignment computation using A^*, which additionally incorporates Propositions 1 and 2 together with *model move restriction*. The algorithm takes a trace net and a sequence as an input, and for both nets, expects an initial- and final marking. In line 1 and line 2 we construct the synchronous product net and corresponding initial- and final marking. Subsequently, in lines 3–5, we initialize the closed set C, open set X, and estimated heuristic set Y. Since the heuristic is consistent, whenever we investigate a marking, we know that the f-value for such marking no longer changes. Hence, the closed set C contains all markings that we have already visited, i.e. for which we have a corresponding final f-value. Within X we maintain all markings inspected at some point, i.e. their g-value is known, and their h-value is either exact or estimated. In Y we keep track of all markings with an underestimating heuristic. In line 6 we initialize pointer function p, which allows us to reconstruct the actual alignment once we reach m_f^S. As long as X contains markings, we select one of the markings having a minimal f-value (line 11). In case the new marking equals m_f^S we construct, using pointer-structure $p(m_f^S)$, the alignment. If the marking is not equal to m_f^S, we, in line 14, check if the corresponding f-value is exact or not. In case it is not, and, the exact $h^{N^S}(m, m_f^S)$ value is exceeding the estimate, we recalculate the marking's f-value and go back to line 11. In any other case, we proceed by storing marking m in the closed set C and by removing it from X. In lines 21–23 we apply model move restriction. Note that it is trivial to alter the code in these lines in order to apply log move restriction. Finally, we fire each transition $t \in T'$ s.t. $(N^S, m) \xrightarrow{t} (N^S, m')$. If the newly reached marking m' is not in the closed set C, we add it to X and check whether we found a shorter path to reach it. If so we update its g-value and derive its h-value.[4] If we actually compute an underestimate, i.e. an $\hat{h}$-value, we register this by adding m' to Y. Finally, we update the pointer-structure p for m'.

It is important to note that set X of Algorithm 1, is typically implemented as a queue. In its basic form, fetching the top element of the queue, i.e. as represented by $m \leftarrow \mathbf{arg\ min}_{m \in X} f(m)$ in line 11, yields any marking that minimizes the (potentially estimated) f-value. In general, a multitude of such markings exists. As observed in [19], minimizing the individual h^{N^S}-value (or $\hat{h}^{N^S}$) as a *second-order criterion*, enhances alignment computation efficiency significantly. We call such second-order criterion `DFS`, as it effectively reduces

[4] In practice, we cache h-values, thus we only derive a new h-value if we did not compute an exact h-value in an earlier stage.

Algorithm 1. A* (Alignments)

```
input  : $N^\sigma = (P^\sigma, T^\sigma, F^\sigma, \lambda^\sigma), m_i^\sigma, m_f^\sigma \in \mathcal{B}(P^\sigma), N = (P, T, F, \lambda), m_i, m_f \in \mathcal{B}(P)$
output: optimal alignment $\gamma^* \in \Gamma(N, \sigma, m'_i, m'_f)$
begin
    $N^S = (P^S, T^S, F^S, \lambda^S) = N^\sigma \otimes N$;          // create synchronous product
    $m_i^S \leftarrow m_i^\sigma \uplus m_i^S$; $m_f^S \leftarrow m_f^\sigma \uplus m'_f$;          // create initial/final marking
    $C \leftarrow \emptyset$;          // initialize closed set
    $X \leftarrow \{m_i^S\}$;          // initialize open set
    $Y \leftarrow \emptyset$;          // initialize estimated heuristics
    $p(m_i^S) = (\varnothing, \varnothing)$;          // initialize predecessor function
    $\forall m \in \mathcal{R}(N^S, m_i^S)\ g(m) \leftarrow \infty$;          // initialize cost so far function $g$
    $g(m_i^S) \leftarrow 0$;          // initialize distance for initial marking
    $f(m_i^S) \leftarrow h^{N^S}(m_i^S, m_f^S)$;          // compute estimate for initial marking
    while $|X| > 0$ do
        $m \leftarrow$ arg min$_{m \in X} f(m)$;
        if $m = m_f^S$ then
            return alignment derived from $\langle t_1, \ldots, t_n \rangle$ where $t_n = \pi_1(p(m_f^S))$,
            $t_{n-1} = \pi_1(p(\pi_2(p(m_f^S))))$, etc. until the initial marking is reached recursively;
        if $m \in Y$ then
            $Y \leftarrow Y \setminus \{m\}$ ;          // remove estimated heuristic
            if $h^{N^S}(m, m_f^S) > \hat{h}^{N^S}(m, m_f^S)$ then
                $f(m) \leftarrow g(m) + h^{N^S}(m, m_f^S)$;
                continue while;          // $m$ is not necessarily minimizing $f$ any more
        $C \leftarrow C \cup \{m\}$;          // add $m$ to the closed set
        $X \leftarrow X \setminus \{m\}$;          // remove $m$ from the open set
        $T' \leftarrow T^S$;
        if $\pi_1(p(m)) = (t, \gg)$, where $t \in T^\sigma$ then
            $T' \leftarrow T' \setminus (\{\gg\} \times T)$;          // model moves not allowed after log moves
        forall the $t \in T'$ s.t. $(N^S, m) \xrightarrow{t} (N^S, m')$ do
            if $m' \notin C$ then
                if $g(m) + c^S(t) < g(m')$ then
                    $g(m') \leftarrow g(m) + c^S(t)$;          // update cost so far function
                    $\hat{h}^{N^S}(m', m_f^S) \leftarrow h^{N^S}(m, m_f^S) - c^S(t)$;          // estimate heuristic
                    $f(m') \leftarrow g(m') + \hat{h}^{N^S}(m', m_f^S)$;
                    if $\hat{h}^{N^S}(m', m_f^S)$ is not exact then
                        $Y \leftarrow Y \cup \{m'\}$;          // add $m'$ to the estimated heuristics set
                    $X \leftarrow X \cup \{m'\}$;          // add $m'$ to the open set
                    $p(m') \leftarrow (t, m)$;          // update predecessor function
    return failure;
```

the estimated distance to a final marking. Within this paper we assume DFS is always applied as a second-order sorting criterion.

4 Evaluation

In this section, we evaluate the effect of different parameters in the search for an optimal alignment for different populations of Petri nets, measured in terms of *search efficiency* and *memory usage*. We measure the efficiency of the search using two metrics: the number of *visited states* and the number of *traversed arcs*. We measure the memory usage of the search using a single metric: the maximum number of *queued states*. Due to the scale of the experiments we have used multiple machines which does not allow us to compare computation time/memory usage directly in all cases. We do however provide such results in case they are comparable. In the remainder of this section we describe the experimental set-up and present a discussion of the obtained results.

4.1 Experimental Setup

The global workflow used in the experiment is illustrated in Fig. 5. For each combination of parameters, the following analysis steps are executed:

1. Generate a sample of (block-structured) Petri nets from a given population (defined in the "Petri net Population Parameters" object).
2. For each Petri net, generate a sample of traces that fit it (defined in the "Log Parameters" object).
3. For each generated trace, add an amount of noise (defined in the "Noise Parameters" object).
4. For each trace with added noise, check the conformance of the trace with respect to the Petri net (using the parameters defined in the "Conformance Checking Parameters" object) and store the results.

Note that the *blocks*, i.e. analysis steps, included in this high-level workflow are not necessarily bound to a concrete implementation. For example, one can use the approach presented in [9] to generate *process trees* that are translated into block-structured Petri nets, and also to generate event logs from them. Alternatively, any other approach that can generate Petri nets from defined populations can be used instead.

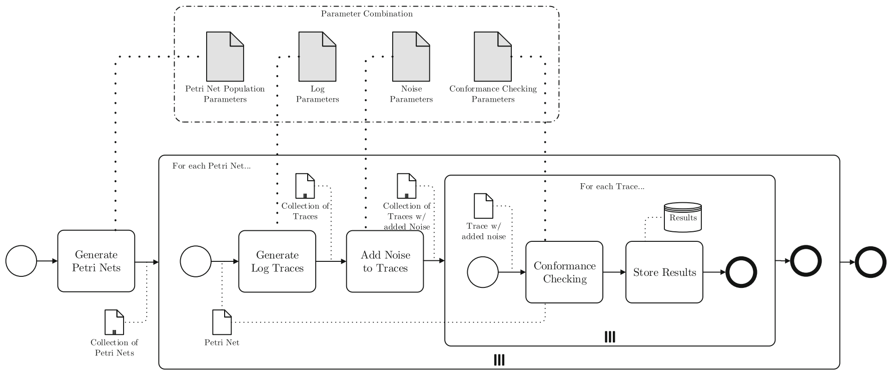


Fig. 5. Schematic overview of the experiment design.

For the experiment, this high-level workflow was implemented as a scientific process mining workflow [7] in `RapidMiner` using building-blocks from the

Table 2. Alignment parameters used in the experiment.

Parameter	Type	Values
Heuristic (h)	Categorical	`LP without lower-bound estimation`
		`LP with lower-bound estimation`
Second-order Queueing Criterion	Categorical	`DFS` (sort on minimized `h`-value)
		`DFS with certainty priority` (sort on minimized `h`-value)
Transition Restriction	Categorical	`MODEL`
		`LOG`

process mining extension `RapidProM` [5]. The workflow enables many types of analysis that allow us to answer a wide variety of research questions related to the efficiency and memory usage of alignments. The experiment performed in this paper focuses on alignment parameters through the following research questions:

Q1. What is the global effect of approximation of the heuristic on the search efficiency and memory usage of alignments?
Q2. What is the effect of incorporating exact derived heuristics within the second-order queuing criterion on the search efficiency and memory usage of alignments?
Q3. What is the effect of transition restriction on the search efficiency and memory usage of alignments?

In order to be able to generalize the results, these effects are studied for different types of Petri nets with different levels of noise added to traces. Within the experiment, we considered 768 value combinations of seven parameters. The alignment related parameters and their values are described in Table 2, whereas the model related parameters are described in Table 3. For each parameter value combination, a collection of 64 Petri nets is generated, and 10 traces are generated from each Petri net. Then, after adding noise, each trace is aligned with the Petri net. In total, this resulted in computing roughly 500, 000 alignments within the experiment.

It is important to note that the different parameter combinations (e.g., heuristics, second-order queueing criterion) were not tested using the same set of Petri nets and traces. To the contrary, they were tested using independent samples of Petri nets and traces randomly obtained from the same populations of processes. In this way we avoid selection bias. Therefore, we consider the absolute differences and trends described in Sect. 4.2 as mere indications. For a proper analysis, in Sect. 4.3 we evaluated the differences in terms of statistical tests and not in terms of absolute differences.

4.2 Results

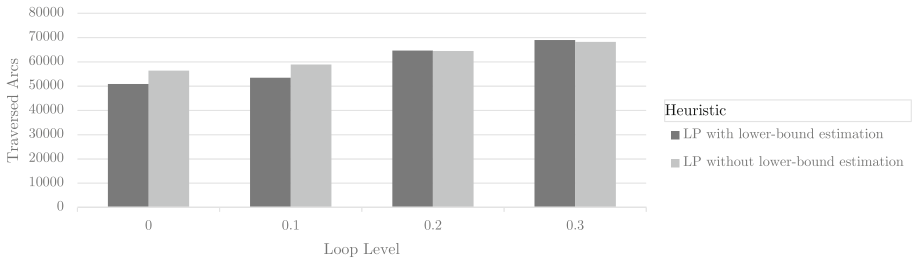
The results of the experiment are scoped in order to provide a straight-forward answer to the research questions proposed earlier. Figure 6 shows the results that

Table 3. Petri net (Pn) and log generation parameters used in the experiment.

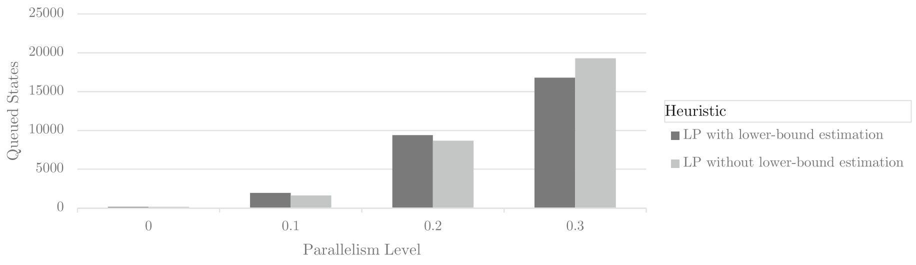
Parameter	Type	Values
Number of activities (Pn)	Numerical	25 50 75
Control-flow Charactersitic (Pn)	Categorical	`Parallelism` `Loops`
C-f Characteristic Level (Pn)	Numerical	0% 10% 20% 30%
Added Noise (Log)	Numerical	0% 20% 40% 60%

relate to **Q1** (i.e., the effect of the *Heuristic* parameter). Figure 6a shows the average number of traversed arcs (related to search efficiency) over increasing levels of loops for two different values of the *Heuristic* parameter: `LP with lower-bound estimation` and `LP without lower-bound estimation`. Here, using LP with lower-bound estimation refers to always deriving (a potentially approximate) heuristic based on a previously computed heuristic, i.e. applying both Propositions 1 and 2. Using LP without estimation refers to only deriving a heuristic when we are guaranteed that it is exact, i.e. only when Proposition 1 holds. When no exact heuristic can be derived an LP is solved immediately. We observe that for increasing levels of loops, the number of traversed arcs is relatively equal. This is as expected as the lower-bound does not affect the search efficiency directly, i.e. it merely allows us to potentially postpone or even prohibit needless solve calls to the underlying LP-solver. Figure 6b shows the average number of queued states (related to memory usage) over increasing levels of parallelism for the same two values of the *Heuristic* parameter: `LP with lower-bound estimation` and `LP without lower-bound estimation`. Again we observe that there is no clear setting that outperforms the other. Also in this case this is expected as using lower-bound estimation does not affect the number of states put in the queue. In Fig. 6c we present computation time.[5] For these results, we expect using lower-bound estimation is beneficial in terms of computation time, i.e. we potentially solve less LP's. We do however not observe this. This is most likely explained by the fact that within the search, markings with an estimated heuristic end up in the top of the queue, and, LP's have to be solved anyway. Moreover, in such case, a marking is potentially reinserted on a lower position in the priority

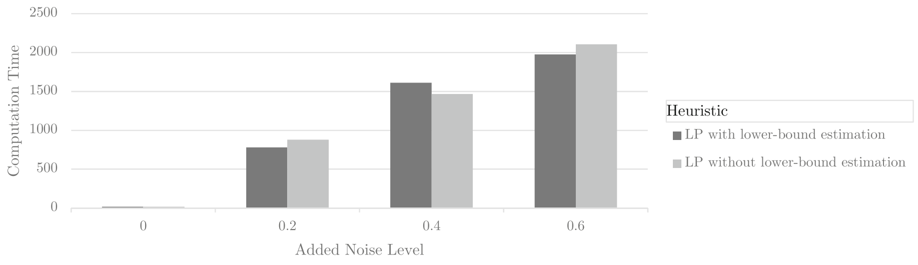
[5] Both experiments ran on the same machine in this instance.



(a) Effect on Traversed Arcs over increasing levels of loops.



(b) Effect on Queued States over increasing levels of parallelism.



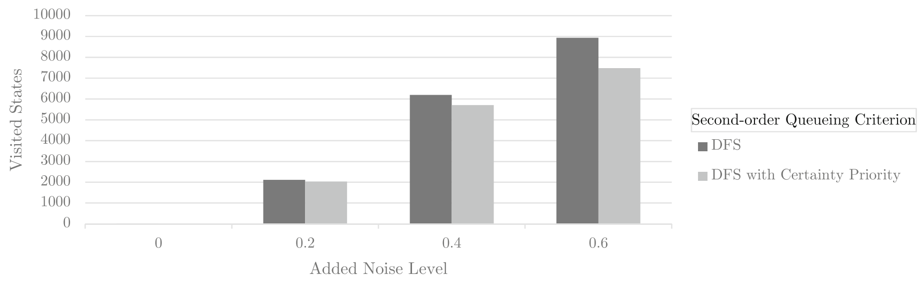
(c) Effect on computation time over increasing levels of noise.

Fig. 6. The effects of the *Heuristic* parameter

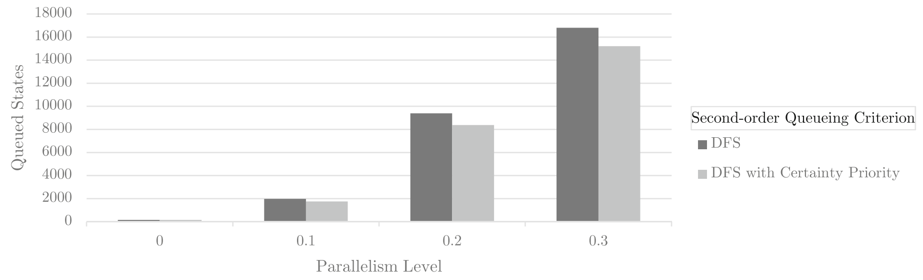
queue. Hence, we do not observe a clear impact on the global efficiency of the search by applying heuristic estimation.

The previous results seem to indicate that the effect of heuristic approximation is negligible. Observe however, that apart from using a DFS-based second order criterion, we arbitrarily select any marking on top of queue X. Figure 7 shows the results that relate to **Q2** (i.e., the effect of heuristic approximation on the *Second-order Queueing Criterion* parameter). In particular, we compare arbitrary top-of-queue selection versus prioritizing markings with an exact heuristic w.r.t. estimated heuristics. Thus, if two markings m and m' have the same f and h value, yet the h value for m is exact whereas that of m' is not, we prioritize m over m'.

Figure 7a shows the average number of visited states (related to search efficiency) over increasing levels of added noise for two different values of the *Second-*



(a) Effect on Visited States over increasing levels of added noise.

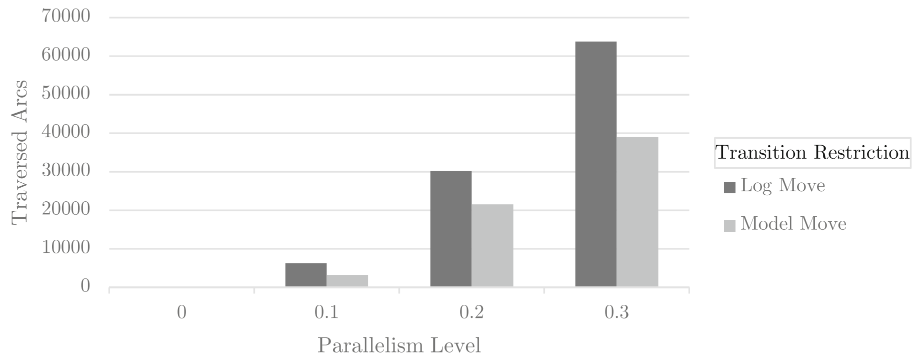


(b) Effect on Queued States over increasing levels of parallelism.

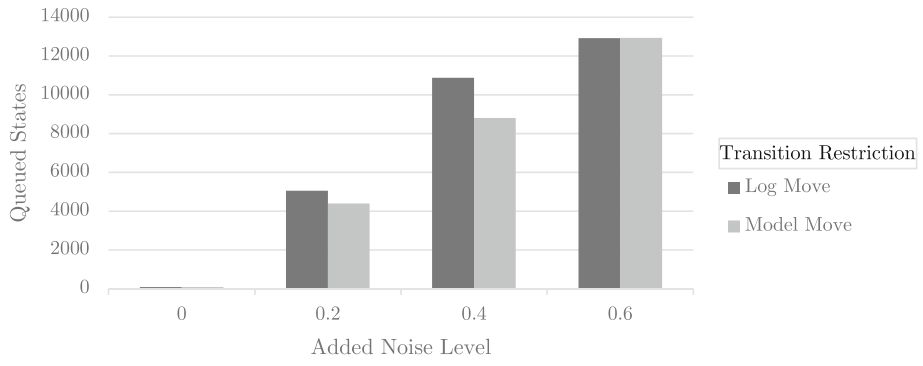
Fig. 7. The effects of the *Second-order Queueing Criterion* parameter

order Queueing Criterion parameter: `DFS` and `DFS with certainty priority`. We observe that `DFS with certainty priority` outperforms default `DFS`. This is most likely explained by the fact that in case a solution to the state equation, at some point, actually corresponds to a firing sequence, the search progresses extremely efficiently. In contrast, in such case, using default `DFS` leads to unnecessary exploration of markings that do not lead to a final state. Figure 6b shows the average number of queued states (related to memory usage) over increasing levels of parallelism for the same two values of the *Second-order Queueing Criterion* parameter: `DFS` and `DFS with certainty priority`. As expected we observe similar results to Fig. 7a.

Finally, Fig. 8 shows the results that relate to **Q3** (i.e., the effect of transition restriction on the search efficiency). In Fig. 8a we present the effect of the different types of transition restriction w.r.t. the traversed arcs, for increasing levels of parallelism. Similarly, in Fig. 8b we present the effects on the number of queued states, for increasing levels of noise. Interestingly, except for noise level of 0.6, using model move restriction outperforms log move restriction. This is as expected since the model part of the synchronous product entails the most variety in terms of behaviour. Hence, limiting model move scheduling is expected to have a positive impact on the search performance.



(a) Effect on traversed arcs over increasing levels of parallelism.



(b) Effect on Queued States over increasing levels of noise.

Fig. 8. The effects of the *Transition Restriction* parameter

4.3 Statistical Analysis

In this section we analyse the statistical significance of the differences in terms of search efficiency and memory usage of several values of the three alignment parameters. We do not assume normal distributions of values, hence, the Kruskal-Wallis non-parametric test [11] is used. This a one-way significance rank-based test for multiple samples, designed to determine whether samples originate from the same population by observing their average ranks. This test does not assume that the samples are normally distributed.

Regarding research question 1 (**Q1**) we performed a Kruskal-Wallis test to assess the significance of the effect of the `Heuristic` parameter in the number of traversed arcs, visited states and queued states with an alpha of 0.05. The test indicated that the effect of the `Heuristic` parameter is not statistically significant in any of the search efficiency or memory usage measurements (*p-values* ≈ 0.5). The same applies for computation time. Regarding research question 2 (**Q2**) we performed a Kruskal-Wallis test to assess the significance of the effect of the `Second-order Queueing Criterion` parameter on the number of traversed arcs, visited states and queued states with an alpha of 0.05. The test indicated that the effect of the parameter on the tree measurements is statistically significant ($p\text{-}value < 0.001$). Regarding research question 3 (**Q3**) we performed a Kruskal-Wallis test to assess the significance of the effect of the `Transition Restriction` parameter in the number of traversed arcs, visited states and queued states with an alpha of 0.05. The test indicated that the effect of the parameter on the number of traversed arcs and queued states is statistically significant ($p\text{-}value < 0.001$ and $p\text{-}value = 0.008$ respectively) but the effect on the number of visited states is not statistically significant ($p\text{-}value = 0.1139$).

5 Related Work

A complete overview of process mining is outside the scope of this paper, hence we refer to [3]. Here, we primarily focus on related work in conformance checking.

Early work in conformance checking uses token-based replay [16]. The techniques try to replay a given trace in a model and add missing tokens if a transition is not able to fire. After replaying the full trace, remaining tokens are counted and a conformance statistic is computed based on missing and remaining tokens.

Alignments are introduced in [6]. The work proposes to transform a given Petri net and a trace from an event log into a synchronous product net, and, subsequently solve the shortest path problem on the corresponding state space. Its implementation in ProM may be regarded as the state-of-the-art technique in alignment computation and serves as a basis for this paper.

In [2,14] decomposition techniques are proposed together with computing alignments. The input model is split into smaller, transition-bordered, submodels for which local alignments are computed. Using decomposition techniques greatly enhances computation time. The downside of the techniques is the fact that they are capable to decide whether a trace fits the model or not, rather than quantifying to what degree a trace fits.

Recently, approximation schemes for alignments, i.e. computation of near-optimal alignments, have been proposed in [18]. The techniques use a recursive partitioning scheme, based on the input traces, and solve multiple Integer Linear Programming problems. The techniques identify deviations between sets of transitions, rather than deviations between singletons (which is the case in [6]). Finally, alignments have also been defined as a planning problem [12] and have been recently studied in online settings [20].

6 Conclusion

In this paper we have presented and formalized an adapted version of the A^* search algorithm used in alignment computation. Within the algorithm we have integrated a number of parameters that, in previous work [19], have shown to be most promising in terms of algorithm efficiency. Based on large-scale experiments, we have assessed the impact of these parameters w.r.t. the algorithm's efficiency. Our results show that restricting the scheduling of model-move based transitions of the synchronous product net most prominently affects search efficiency. Moreover, the explicit prioritization of exactly derived heuristics seems to have a positive, yet less prominent, effect as well.

Future Work. Within this work we have assessed, using large-scale experiments, the impact of several parameters on the efficiency of computing optimal alignments. However, several approximation schemes exist for A^*, e.g. using a scaling function within the heuristic. We plan to assess the impact of these approximation schemes on alignment computation as well. We also plan to examine the use of alternative informed search methods, e.g. *Iterative Deepening* A^*.

References

1. van der Aalst, W.M.P.: The application of Petri nets to workflow management. J. Circ. Syst. Comput. **8**(1), 21–66 (1998)
2. van der Aalst, W.M.P.: Decomposing Petri nets for process mining: a generic approach. Distrib. Parallel Databases **31**(4), 471–507 (2013)
3. van der Aalst, W.M.P.: Process Mining - Data Science in Action, 2nd edn. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49851-4
4. van der Aalst, W.M.P., Adriansyah, A., van Dongen, B.F.: Replaying history on process models for conformance checking and performance analysis. Wiley Interdisc. Rev.: Data Min. Knowl. Disc. **2**(2), 182–192 (2012)
5. van der Aalst, W.M.P., Bolt, A., van Zelst, S.J.: RapidProM: mine your processes and not just your data. CoRR abs/1703.03740 (2017)
6. Adriansyah, A.: Aligning observed and modeled behavior. Ph.D. thesis, Department of Mathematics and Computer Science, Eindhoven University of Technology, July 2014
7. Bolt, A., de Leoni, M., van der Aalst, W.M.P.: Scientific workflows for process mining: building blocks, scenarios, and implementation. STTT **18**(6), 607–628 (2016)
8. Hart, P.E., Nilsson, N.J., Raphael, B.: A formal basis for the heuristic determination of minimum cost paths. IEEE SSC **4**(2), 100–107 (1968)
9. Jouck, T., Depaire, B.: PTandLogGenerator: a generator for artificial event data. In: BPM Demos, vol. 1789, pp. 23–27. CEUR-WS.org (2016)
10. Kosaraju, S.R.: Decidability of reachability in vector addition systems (preliminary version). In: ACM Theory of Computing, pp. 267–281. ACM (1982)
11. Kruskal, W.H., Wallis, W.A.: Use of ranks in one-criterion variance analysis. J. Am. Stat. Assoc. **47**(260), 583–621 (1952)
12. de Leoni, M., Marrella, A.: Aligning real process executions and prescriptive process models through automated planning. Expert Syst. Appl. **82**, 162–183 (2017)
13. Mayr, E.W.: An algorithm for the general Petri net reachability problem. SIAM J. Comput. **13**(3), 441–460 (1984)
14. Munoz-Gama, J., Carmona, J., van der Aalst, W.M.P.: Single-entry single-exit decomposed conformance checking. Inf. Syst. **46**, 102–122 (2014)
15. Murata, T.: Petri nets: properties, analysis and applications. Proc. IEEE **77**(4), 541–580 (1989)
16. Rozinat, A., van der Aalst, W.M.P.: Conformance checking of processes based on monitoring real behavior. Inf. Syst. **33**(1), 64–95 (2008)

17. Schrijver, A.: Theory of Linear and Integer Programming. Wiley-Interscience Series in Discrete Mathematics and Optimization. Wiley, Hoboken (1999)
18. Taymouri, F., Carmona, J.: A recursive paradigm for aligning observed behavior of large structured process models. In: La Rosa, M., Loos, P., Pastor, O. (eds.) BPM 2016. LNCS, vol. 9850, pp. 197–214. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-45348-4_12
19. van Zelst, S.J., Bolt, A., van Dongen, B.F.: Tuning alignment computation: an experimental evaluation. In: Proceedings of ATAED 2017, pp. 1–15 (2017)
20. van Zelst, S.J., Bolt, A., Hassani, M., van Dongen, B.F., van der Aalst, W.M.P.: Online conformance checking: relating event streams to process models using prefix-alignments. IJDSA (2017). https://link.springer.com/article/10.1007/s41060-017-0078-6

Heuristic Mining Approaches for High-Utility Local Process Models

Benjamin Dalmas[1](✉), Niek Tax[2], and Sylvie Norre[1]

[1] Clermont-Auvergne University, LIMOS CNRS UMR 6158, Aubière, France
{benjamin.dalmas,sylvie.norre}@isima.fr

[2] Eindhoven University of Technology, Department of Mathematics and Computer Science, P.O. Box 513, 5600 MB Eindhoven, The Netherlands
n.tax@tue.nl

Abstract. Local Process Models (LPMs) describe structured fragments of process behavior that occur in the context of business processes. Traditional support-based LPM discovery aims to generate a collection of process models that describe highly frequent behavior, in contrast, in High-Utility Local Process Model (HU-LPM) mining the aim is to generate a collection of process models that provide useful business insights according to a specified *utility function*. Mining LPMs is computationally expensive as the search space depends combinatorially on the number of activities in the business process. In support-based LPM mining, the search space is constrained by leveraging the anti-monotonic property of support (i.e., the apriori principle). We show that there is no property of monotonicity or anti-monotonicity in HU-LPM mining that allows for lossless pruning of the search space. We propose four heuristic methods to explore the search space only partially. We show on a collection of 57 event logs that these heuristics techniques can reduce the size of the search space of HU-LPM mining without much loss in the mined set of HU-LPMs. Furthermore, we analyze the effect of several properties of the event log on the performance of the heuristics through statistical analysis. Additionally, we use predictive modeling with regression trees to explore the relation between *combinations of log properties* and the effect of the heuristics on the size of the search space and on the quality of the HU-LPMs, where the statistical analysis focuses on the effect of log properties *in isolation*.

Keywords: Process discovery · Pattern mining
Approximate methods

1 Introduction

Process Mining [1] has emerged as a new discipline that aims at supporting business process improvement through the analysis of event data recorded by information systems. An event log contains recorded *events* related to a process execution. Events consist of a case identifier (grouping together events that belong to the same process instance), and information on what was performed, when, by whom, etc. Process discovery techniques aim to discover an interpretable model that accurately describes the process from such an event log. The process models obtained with process discovery give insight into what is happening in the process and can be used as a starting point for different types of further analysis, e.g., bottleneck analysis [23], and comparison of the same process between organizations [6]. Many algorithms have been proposed for process discovery, e.g., [3,4,19,21,22].

A recent technique is Local Process Model (LPM) discovery [27,30], which aims at the discovery of a ranking of process models (i.e., LPMs), where each LPM describes only a subset of the process activities. LPMs aim to describe frequent local pieces of behavior, therefore, LPM mining can be seen as a special form of *frequent pattern mining* [13], where each pattern is a process model. However, in contrast to other pattern mining approaches that operate on sequence data, such as episode mining [18] and sequential pattern mining [26], LPMs are not limited to subsequences or partial orders and can additionally consist of more complex structures, such as loops and choices.

A recent trend in the frequent pattern mining field is to focus on the mining of *useful patterns* as opposed to *frequent patterns*, i.e., patterns that address business concerns such as high financial costs. In previous work, we introduced high-utility local process models (HU-LPMs) [29] to bridge the concept of utility based discovery into the process mining field and adapted it to the logging concepts typically seen in process mining event logs, such as event attributes, trace attributes, etc. In the pattern mining field, *utility* often follows a narrower definition that is solely based on the set of activities that is described by a pattern.

To deal with the computational complexity of searching patterns with such a rich set of constructs that are supported by LPMs (i.e., sequential orderings, parallel blocks, loops, choices), a support-based pruning strategy [30] as well as a set of heuristic approaches [27] have been introduced for the discovery of LPMs. However, support-based pruning and the existing set of heuristic approaches for LPM discovery cannot be used for the discovery of high-utility LPMs, as LPMs with high utility can be infrequent. Furthermore, the utility of an LPM is not necessarily monotonic, i.e., an LPM that does not meet a utility threshold can still be expanded into an LPM that does meet this threshold.

This paper extends the work started in [8] to propose four different heuristic approaches to prune the search space of the HU-LPM discovery task. We perform experiments on three real-life logs and 54 synthetic logs and show that the heuristics reduce size of the HU-LPM mining search space while still being able to discover useful HU-LPMs. Furthermore, we analyze the relation between the performance of the heuristic HU-LPM techniques and properties of the event log. The techniques described in this paper have been implemented in the ProM process mining framework [10] as part of the *LocalProcessModelDiscovery*[1] package.

[1] https://svn.win.tue.nl/repos/prom/Packages/LocalProcessModelDiscovery/.

This paper is organized as follows. Section 2 describes related work. Section 3 introduces the basic concepts used in this paper. In Sect. 4, we introduce the four heuristic approaches for HU-LPM mining. In Sect. 5 we evaluate the heuristics on a collection of even logs and present two experiments to analyze the relation between characteristics of the event log and the performance of the heuristics: a *statistical analysis* into the effect of properties of the log *in isolation*, and a *regression tree analysis* to explore *joint effects of combinations of log properties*. Finally, we conclude and discuss future areas of research in Sect. 6.

2 Related Work

In the pattern mining discipline, the limitations of support-based mining have become apparent in recent years, and as a result, the interest has grown in high-utility patterns; i.e., patterns providing useful business insight. This has led to an increasing number of methods and techniques that address the high-utility mining (HUM) problem [9,32–34]. USpan uses a *lexicographic quantitative sequence tree (LSQ-tree)* to extract the complete set of high utility sequences [32]. An LQS-tree is a tree structure where each node stores a sequence of activities and its utility. The sequence stored by a node being a super-sequence of the sequence stored by the node's parent, this type of structure allows for fast access and updates when mining high-utility patterns. A similar tree structure, the *HUSP-Tree* is used by the HUSP-Stream algorithm to enable fast updates when mining high-utility patterns from sequential data streams [34]. The problem of mining incremental sequential datasets is also addressed in [9], using an efficient indexing strategy. In [33], the HUSRM algorithm efficiently mines sequential rules using a *utility table*, a novel data structure to support recursive rule expansions.

The utility in sequential patterns is regarded to be the sum of the utility of the activities that fit the sequential pattern. The majority of pruning strategies that are used in HUM algorithms are based on *Transaction-Weighted Utility* (TWU). The TWU of a pattern X is the sum of utilities of the sequences containing X, resulting in an upper bound for the utility of pattern X that can be computed efficiently. In case the TWU of a pattern X does not meet a predefined minimum threshold, X can be safely pruned since its actual utility can only be lower than or equal to TWU. In traditional HUM algorithms, the utility function is defined on the activity level; i.e., each activity in the dataset is given a utility, and the utility of a pattern is the sum of all activity utilities. Therefore, TWU and other activity-based pruning strategies can be used for efficient pruning for HU-LPM mining when the utility is defined on the activity level. However, utility functions of HU-LPMs are defined in a more general way, allowing utility additionally to depend on event and/or trace attributes instead of solely on the activity. Therefore, TWU cannot be used to prune the search space of HU-LPMs. With sequence-based pruning strategies being inapplicable in HU-LPM mining, we investigate in this paper utility-based heuristics.

3 Preliminaries

In this section, we introduce notations related to event logs, Local Process Models (LPMs) and High-Utility Local Process Models (HU-LPMs) which are used in later sections of this paper.

3.1 Events, Traces, and Event Logs

X^* denotes the set of all sequences over a set X and $\sigma = \langle a_1, a_2, \ldots, a_n \rangle$ a sequence of length n, with $\sigma(i) = a_i$ and $|\sigma| = n$. $\langle\rangle$ is the empty sequence and $\sigma_1\sigma_2$ is the concatenation of sequences σ_1 and σ_2. We denote with $\sigma\restriction_X$ the projection of sequence σ on set X, e.g., for $\sigma = \langle a, b, c \rangle$, and $X = \{a, c\}$, $\sigma\restriction_X = \langle a, c \rangle$.

In the context of process logs, we assume the set of all *process activities* Σ to be given. An *event* e in an event log is the occurrence of an activity $e \in \Sigma$. We call a sequence of events $\sigma \in \Sigma^*$ a *trace*. An *event log* $L \in \mathbb{N}^{\Sigma^*}$ is a finite multiset of traces. For example, the event log $L = [\langle a, b, c \rangle^2, \langle b, a, c \rangle^3]$ consists of 2 occurrences of trace $\langle a, b, c \rangle$ and three occurrences of trace $\langle b, a, c \rangle$. We lift projection of sequences to multisets of sequences, e.g., $L\restriction_{\{a,c\}} = [\langle a, c \rangle^5]$.

3.2 Local Process Models

LPMs [30] are process models that describe frequent but partial behavior; i.e., they model a subset of the activities of the process, seen in the event log. An iterative expansion procedure is used in [30] to generate a ranked collection of LPMs. LPMs are limited to 5 activities as the expansion procedure is a combinatorial problem of which the size depends on the number of activities in the event log as well as the maximum number of activities in the LPMs that are mined. Though LPMs can be represented in any process modeling notation, such as BPMN [24], UML [14], or EPC [16], here we use *process trees* [5] to represent LPMs.

A process tree is a tree structure where leaf nodes represent activities. The non-leaf nodes represent *operators*, which specify the allowed behavior over the activity nodes. Allowed operator nodes are the *sequence* operator ($\rightarrow$) that indicates that the first child is executed before the second, the *exclusive choice* operator ($\times$) that indicates that exactly one of the children can be executed, the *concurrency* operator ($\wedge$) that indicates that every child will be executed but allows for any ordering, and the *loop* operator ($\circlearrowright$), which has one child node and allows for repeated execution of this node. $\mathfrak{L}(LPM)$ represents the language of process tree LPM, i.e., the set of sequences allowed by the model. Figure 1d shows an example process tree M_4, with $\mathfrak{L}(M_4) = \{\langle A, B, C \rangle, \langle A, C, B \rangle, \langle D, B, C \rangle, \langle D, C, B \rangle\}$. Informally, it indicates that either activity A or D is executed first, followed by the execution of activities B and C in any order. M_4 can also be written shorthand as $\rightarrow(\times(A, D), \wedge(B, C))$. Note that our definition of $\circlearrowright$ deviates from its traditional definition [5] where it is defined as having three children: a *do*, *redo*, and *exit* child, where execution of the redo subtree enables another execution of the do subtree, and finally the exit subtree is executed. It is easy to see that this traditional definition leads to redundancy in the search space for the case of LPM mining: the exit child can be mimicked by combining the $\rightarrow$ and unary $\circlearrowright$ operator nodes. Furthermore, an activity node a as do-child and b as redo-child is identical to the following model with a unary loop: $\rightarrow(a, \circlearrowright(\rightarrow(b, a)))$.

A technique to generate a ranked collection of LPMs through iterative expansion of candidate process trees is proposed in [30]. The expansion procedure consists in the replacement of one of the leaf activity node a of the process tree by either operator node $\rightarrow, \times$, or $\wedge$, where one of the child nodes is the replaced activity node a and the other is a new activity node b. Alternatively, a leaf node of an LPM can be expanded by replacing it with operator node $\circlearrowright$ with the replaced node as single child. $\mathcal{M}$ is the LPM universe; i.e., the set of all possible LPMs. An LPM $M \in \mathcal{M}$ can be expanded in many ways, as it can be extended by replacing any one of its activity nodes, expanding it with any of the operator nodes, and with a new activity node that represents any of the activities present in the event log. $Exp(M)$ denotes the set of expanded LPMs that can be created by expanding M, and exp_max the maximum number of expansions allowed from an *initial LPM*; i.e., an LPM containing only one activity.

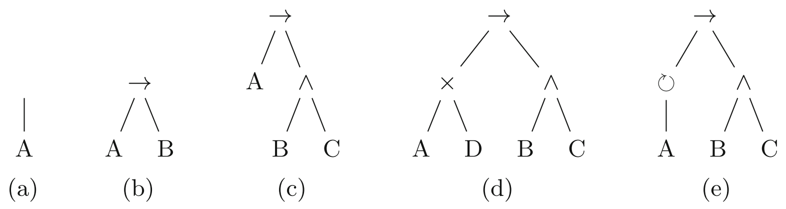


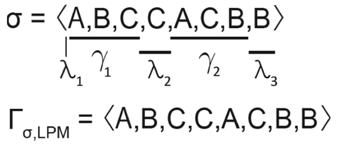
Fig. 1. *(a)* An initial LPM M_1, *(b-d)* M_2, M_3, and M_4, three consecutive expansions of M_1, and *(e)* M_5, an alternative expansion of M_3.

Figure 1 illustrates the expansion procedure, starting from the initial LPM M_1 of Fig. 1a. The LPM of Fig. 1a is first expanded into a larger LPM by replacing A by operator node $\rightarrow$, with activity A as its left child node and B its right child node, resulting in the LPM of Fig. 1b. Note that M_1 can also be expanded in other ways, and LPM discovery recursively explores all possible process trees that meet a support threshold by iterative expansion. In a second expansion step, activity node B of the LPM of Fig. 1b is replaced by operator node $\wedge$, with activity B as its left child and C its right child, resulting in Fig. 1c. The activity node A of the LPM of Fig. 1c is replaced by operator node $\times$ with as left child activity A and as right child activity D, forming the LPM of Fig. 1d. Another expansion of the Fig. 1c LPM is shown in Fig. 1e, replacing activity node A by the loop operator ($\circlearrowright$) with activity A as child.

To evaluate a given LPM on a given event log L, its traces $\sigma \in L$ are first projected on the set of activities X in the LPM, i.e., $\sigma' = \sigma\restriction_X$. The projected trace

event id	activity	time	cost
1	A	26-3-2017 13:00	€100
2	D	26-3-2017 13:27	€200
3	B	26-3-2017 13:25	€300
4	C	26-3-2017 13:35	€100
5	D	26-3-2017 13:42	€400
6	C	26-3-2017 15:47	€200
7	A	26-3-2017 16:10	€100
8	C	26-3-2017 16:34	€400
9	B	26-3-2017 16:52	€300
10	D	26-3-2017 16:59	€200
11	A	26-3-2017 17:13	€1000
12	B	26-3-2017 17:15	€1000
13	A	26-3-2017 17:16	€150

(a)



(b)

Fig. 2. *(a)* A trace σ of an event log L. *(b)* The segmentation of σ on M_3.

σ' is then segmented into γ-segments that fit the behavior of the LPM and λ-segments that do not fit the behavior of the LPM, i.e., $\sigma'{=}\lambda_1\gamma_1\lambda_2\gamma_2\cdots\lambda_n\gamma_n\lambda_{n+1}$ such that $\gamma_i{\in}\mathfrak{L}(LPM)$ and $\lambda_i{\notin}\mathfrak{L}(LPM)$. We define $\Gamma_{\sigma,LPM}$ to be a function that projects trace σ on the LPM activities and obtains its subsequences that fit the LPM, i.e., $\Gamma_{\sigma,LPM} = \gamma_1\gamma_2\ldots\gamma_n$.

Let our LPM M_3 under evaluation be the process tree of Fig. 1c and let σ be the example trace shown in Fig. 2a. Function $Act(LPM)$ obtains the set of process activities in the LPM, e.g. $Act(M_3) = \{A, B, C\}$. Projection on the activities of the LPM gives $\sigma{\restriction}_{Act(M_3)} = \langle A, B, C, C, A, C, B, A, B, A\rangle$. Figure 2b shows the segmentation of the projected trace on the LPM, leading to $\Gamma_{\sigma,LPM} = \langle A, B, C, A, C, B\rangle$. The segmentation starts with an empty non-fitting segment λ_1, followed by a fitting segment $\gamma_1{=}\langle A, B, C\rangle$, which completes one run through the process tree. The second event C in σ cannot be replayed on LPM, since it only allows for one C and γ_1 already contains a C. This results in a non-fitting segment $\lambda_2{=}\langle C\rangle$. $\gamma_2{=}\langle A, C, B\rangle$ again represents a run through process tree, the segmentation ends with non-fitting segment $\lambda_3{=}\langle A, B, A\rangle$. We lift segmentation function Γ to event logs, $\Gamma_{L,LPM}{=}\{\Gamma_{\sigma,LPM}|\sigma{\in}L\}$. An alignment-based [2] implementation of Γ, as well as a method to rank and select LPMs based on their support, i.e., the number of events in $\Gamma_{L,LPM}$, is described in [30]. Note that there can exist multiple optimal alignments for a trace on a model. The default ProM implementation of alignments therefore returns a arbitrary optimal alignment from the set of optimal alignments. However, other implementations exist that obtain *all optimal alignments* instead of a single arbitrary one. Since searching all optimal alignments and LPM mining are both computationally complex tasks, in practice the support of an LPM is calculated by obtaining an arbitrary optimal alignment.

The expansion procedure in traditional LPM discovery stops when the number of instances (called *support*) of the behavior that is described by it does not exceed some given threshold. This support is defined as the number of γ-segments in $\Gamma_{L,LPM}$. Several metrics are taken into account to assess the quality of an LPM, but all of them are support-based. We refer to [30] for more detailed and formal definitions of LPMs, extensions of LPMs, and evaluating LPMs on logs.

Note the definition of *Exp* provided in [30] allows for the generation of identical LPMs by expanding different LPMs, i.e., there can exist LPMs M, M', and M'' such that $M{\in}Exp(M')$ and $M{\in}Exp(M'')$ but $M'{\neq}M''$. The ProM implementation of LPM mining uses a hashset to check in constant time for each LPM M that is created by expanding another LPM M' whether M was already created by expanding some other LPM M'' and only calculates $\Gamma_{L,M}$ and expands M further when this is not the case. Note that, as an effect, the pruning depends on the *order in which* LPMs are expanded, and therefore we assume the ordering in which LPMs are expanded to be arbitrary but consistent. Note that this does not have any effect for the correctness of the pruning: if $\Gamma_{L,M}$ yields a support below the threshold, then some expansion leading to M' that is guaranteed to reduce support can be safely pruned away, even if M' might also be reachable by extending some other LPM M'' that does meet the support threshold. Worst case, this might lead to suboptimal pruning, in the case M'' is scheduled to be expanded to M' before M is expanded to M'.

Definition 1 ***Local Process Model Mining Problem:*** *Given an event log L, the LPM mining problem is defined as the task of discovering a set of frequent LPMs, where the total number of replayable fragments exceeds a defined threshold.*

3.3 High-Utility Local Process Models

A High-Utility Local Process Model (HU-LPM) is an LPM where (i) its importance is related to the *utility* of the fragments it can replay instead of the number of fragments it can replay and (ii) where this utility is above a predefined threshold. Note that HU-LPMs are a generalization of LPMs, as we have shown in [29] that the quality measures that are used in support-based LPM mining can be expressed as utility functions for HU-LPM mining. Several scopes on which utility functions can be defined are described in [29]:

Trace the most general class of utility functions. The trace-level utility functions allow the utility of fitting trace fragments to depend on the events in these specific fragments, their attributes, and properties of the case itself. An example of a trace-level utility function is mining for LPMs that explain a high share of the total running time of a case.

Event this class of utility functions can be used when the interest is focused on some event properties, but it does not concern the trace-context of those events. An example of an event-level utility function is the search for LPMs describing process fragments with high financial cost.

Activity defines the utility of an LPM based on the frequency of occurrences of each activity. It can be used when the analyst is more interested in some activities with high impact (e.g., lawsuits, security breaches, etc.). This scope is the one to which traditional pattern mining algorithms are mostly limited, which allows algorithms from this field to define upper bounds to efficiently prune the search space without loss.

Model this class of utility functions is not log-dependent, but it allows the analyst to specify preferences for specific structural properties of the LPM.

Functions on the different scopes can be combined to form composite functions, consisting of component functions on one of the scopes above. The utility of an LPM M over an event log L is denoted $u(L, M)$, and we define as *HU-list* a collection of HU-LPMs sorted in descending order according to their utility, with $|\mathcal{S}|$ the number of HU-LPMs in HU-list $\mathcal{S}$. For a more thorough introduction of HU-LPMs and related concepts, we refer the reader to [29].

4 Pruning Strategies

In contrast to regular Local Process Model (LPM) mining, High Utility LPM (HU-LPM) mining cannot be performed with techniques that prune the search space based on frequency, leading to a large search space. Therefore, there is a need for an alternative pruning strategy for HU-LPM mining that makes mining possible on larger logs, however, the utility metric as defined in [29] is not necessarily anti-monotonic, preventing any lossless reduction of the search space. When setting a stopping criterion c on LPMs such that we expand an LPM M only when c holds for M, we say that c is anti-monotonic when M violating c implies that all $M' \in \mathit{Exp}(M)$ violate c.

Property 1. *$\exists_{M \in \mathcal{M}}(\exists_{M' \in Exp(M)} u(L, M') < u(L, M) \;\wedge\; \exists_{M' \in Exp(M)} u(L, M') \geq u(L, M))$.*

We show that anti-monotonicity does through the following counter-example. Let M_1, M_2, M_3, M_4 shown in Figs. 1a-d, with $M_2 \in \mathit{Exp}(M_1)$, $M_3 \in \mathit{Exp}(M_2)$, $M_4 \in \mathit{Exp}(M_3)$. Let event log L consist of the single trace shown in Fig. 2a, and let the utility be the sum of the cost attributes of the events that belong to replayable fragments. This results in utilities $u(L, M_1) = 1350$, $u(L, M_2) = 2800$, $u(L, M_3) = 1300$, $u(L, M_4) = 1400$. It is easy to see that utility is not anti-monotonic, as $u(L, M_3) < u(L, M_2)$, but $u(L, M_4) > u(L, M_3)$. This leads to non-optimal HU-LPMs when we prune using a minimum utility threshold, e.g., stopping criterion $c : u(L, M) \geq 1350$ leads to M_3 not being expanded because its utility is below the threshold, while the utility of M_4 would have again been above the threshold. This is mainly explained by the fact that the utility added by the new activity does not compensate for the utility lost because of the fragments that do not fit the new LPM but that did fit the previous LPM. Note that anti-monotonicity property does in fact hold in the special case of a utility function that is defined on the set of γ-segments that fit the LPM and is independent of the events or activities in those γ-segments. This special class of utility functions is close to traditional LPM mining, where each γ-segment is assumed to have equal utility.

Definition 2 ***High-Utility Local Process Model Mining Problem:*** *Given an event log L, the HU-LPM mining problem is defined as the task of discovering a set of LPMs with utility above a predefined threshold* u_{min}*, i.e.,* $u(L, LPM) \geq u_{min}$.

In High-Utility Local Process Model (HU-LPM) discovery, the size of the search space grows combinatorially with the number of activities. Reducing the search space is an inevitable step to ensure efficiency or even to enable the algorithm to run in acceptable time. We have shown that the utility metric is not anti-monotonic and that we, therefore, cannot reduce the search space without loss. However, heuristics can be used to reduce execution time, without formal guarantee of finding an optimal solution; i.e., the discovered set of LPMs fulfilling the utility threshold might be incomplete.

The remainder of this section is as follows. We define new concepts related to HU-LPMs in Sect. 4.1, we introduce two memoryless heuristics in Sect. 4.2, and introduce two memory-based heuristics in Sect. 4.3.

4.1 Basic Concepts

Let $M \in \mathcal{M}$ be a HU-LPM, then $Par(M)$ denotes the parent of m; $Par(M) = M' \in \mathcal{M}$ such that $M \in Exp(M')$. For example, for the process trees of Fig. 1, $Par(M_3) = M_2$. We generalize the concept of parent in Eq. 1, and define $Par^i(M)$ as the i^{th} parent of M, with $Par^0(M) = M$, $Par^1(M) = Par(M)$, $Par^2(M) = Par(Par(M))$ and so forth; In general, we define $Par^i(M)$ as:

$$Par^i(M) = \begin{cases} Par^{i-1}(Par(M)) & \text{if } i > 1, \\ M & \text{if } i = 0. \end{cases} \tag{1}$$

For example, for the process trees of Fig. 1, $Par^3(M_4) = M_1$. Note that $M \notin dom(Par)$ for LPMs $M \in \mathcal{M}$ that are initial LPMs, as initial LPMs have no parent LPM defined. Note that when a LPM can be reached by expansion of more than one LPM, Par yields the one from which the LPM actually was extended in practice. Furthermore, we define $it\_nb(M)$ as the number of expansions to reach HU-LPM M from an initial HU-LPM; e.g., $it\_nb(M_3) = 2$. Formally:

$$it\_nb(M) = \begin{cases} 0, & \text{if } M \notin dom(Par), \\ it\_nb(Par(M)) + 1, & \text{if } M \in dom(Par). \end{cases} \tag{2}$$

Using Par and it_nb we define $Anc(M)$ as the set of ancestors of the LPM M; $Anc(M) = \bigcup_{i=1}^{it\_nb(M)} Par^i(M)$, e.g., for Fig. 1, $Anc(M_4) = \{M_1, M_2, M_3\}$.

Based on these definitions, we now introduce four heuristics to reduce the execution time of HU-LPM mining.

4.2 Memoryless Heuristics

This first type of heuristics focuses on local comparisons, i.e., an LPM is only compared with its parent, the previous expansions are not considered. The heuristics work as follows: for a defined number of successive extensions (noted k, such that $0 < k < exp\_max$), the new LPM is allowed to have a utility lower than or equal to the utility of its parent.

For each heuristic, we define a *continuation criterium function*, $ctn(L, k, M)$, which results to 1 if the k most recent expansion steps leading to LPM M meet the requirements of the heuristic, indicating that M should be expanded further. Otherwise, function *ctn* results to 0 and M will not be expanded further, therefore reducing the size of the search space. Let $\beta(bexp)$ be the function that returns 1 if boolean expression *bexp* is True and returns 0 otherwise. We introduce heuristic h_1, which formalizes the function $ctn(L, k, M)$ as defined above:

Heuristic 1 (h_1)**:** The expansion of LPM $M \in \mathcal{M}$ is stopped if all LPMs from the $k-1^{th}$ parent of M to M itself have a utility lower or equal to the utility of its parent.

$$ctn(L,k,M) = \begin{cases} 1 - \prod\limits_{i=0}^{min(it\_nb(M),k)-1} \beta(u(L, Par^i(M)) \leq u(L, Par^{i+1}(M))), & \text{if } it\_nb(M) \geq 1 \\ 1, & \text{otherwise.} \end{cases} \tag{3}$$

As we want initials LPMs always to be expanded independently of any heuristic, $ctn(L, k, M) = 1$ when $it\_nb(M) = 0$, and the function defined by each heuristic otherwise. We additionally propose heuristic h_2, a relaxed version of h_1, where the expanded LPM is always allowed to have the same utility as its parent:

Heuristic 2 (h_2)**:** The expansion of LPM $m \in \mathcal{M}$ is stopped if all LPMs from the $k-1^{th}$ parent of M to M itself have a utility strictly lower than the utility of their parents.

$$ctn(L,k,M) = \begin{cases} 1 - \prod\limits_{i=0}^{min(it\_nb(M),k)-1} \beta(u(L, Par^i(M)) < u(L, Par^{i+1}(M))), & \text{if } it\_nb(M) \geq 1 \\ 1, & \text{otherwise.} \end{cases} \tag{4}$$

4.3 Memory-Based Heuristics

The second type of heuristics keeps in memory the set of LPMs produced by the successive expansions. Instead of comparing two successive expansions, it compares an extension with its best ancestor. For an LPM M, we define $B(L, M)$ as the highest utility among the ancestors of M in event log L; $B(L, M) = u(L, M')$ of LPM $M' \in Anc(M)$ such that $\nexists_{M'' \in Anc(M)} : u(L, M'') > u(L, M')$.

The heuristics work as follows: for a defined number of successive expansions (noted k, such that $0 < k < exp\_max$), the expanded LPM is allowed to have a utility lower than or equal to the utility of its best ancestor. We introduce heuristic h_3, which formalizes the function $ctn(L, k, M)$ as defined above:

Heuristic 3 (h_3)**:** The expansion of LPM $M \in \mathcal{M}$ is stopped if all LPMs from the $k-1^{th}$ parent of M to M itself have a utility lower or equal to the highest utility among their ancestors.

$$ctn(L,k,M) = \begin{cases} 1 - \displaystyle\prod_{i=0}^{min(it\_nb(M),k)-1} \beta(u(L, Par^i(M)) \leq B(L, Par^i(M))), & \text{if } it\_nb(M) \geq 1 \\ 1, & \text{otherwise.} \end{cases} \quad (5)$$

We also propose heuristic h_4, a relaxed version of h_3, where the expanded LPM is always allowed to have the same utility as its best ancestor:

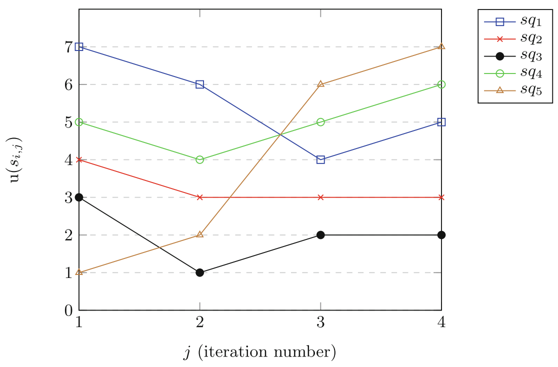
Heuristic 4 (h_4)**:** The expansion of LPM $M \in \mathcal{M}$ is stopped if all LPMs from the $k-1^{th}$ parent of M to M itself have a utility strictly lower than the highest utility among their ancestors.

$$ctn(L,k,M) = \begin{cases} 1 - \displaystyle\prod_{i=0}^{min(it\_nb(M),k)-1} \beta(u(L, Par^i(M)) < B(L, Par^i(M))), & \text{if } it\_nb(M) \geq 1 \\ 1, & \text{otherwise.} \end{cases} \quad (6)$$

To illustrate these four heuristics, let the plot in Fig. 3a be our running example. Let sq_i represent a sequence of expansions from an initial LPM, and $sq_{i,j}$ be the j^{th} LPM of that sequence of expansions sq_i. For each heuristic and $k = 2$, Fig. 3b presents the sequences that would be expanded until the fourth step (marked with ✓) and those that would be stopped before (marked with ✗).

Here, sq_1 and sq_5 are two extremes. While sq_1 contains two successive utility decreases ($sq_{1,2}$ and $sq_{1,3}$), sq_5 contains only LPMs having a higher utility at each expansion. In consequence, every heuristic would have stopped sq_1 after $sq_{1,3}$; removing further expansions from the search space, and would have expanded sq_5 until $sq_{5,4}$. Sq_2 is only expanded until the fourth step by h_2 because this relaxed version allows for the utility to stagnate during two steps after a first decrease. On the contrary, the expansion of sq_4 is only stopped by h_3 because the utility obtained in the first step remains higher than the ones obtained at each further step. Finally, sq_3 is expanded until the fourth step by the memoryless heuristics but stopped by the memory-based heuristics because the increase at $sq_{3,3}$ is enough for the utility of the expansion to be higher than the utility of its parent, but not enough to be at least equal to the utility of its best ancestor.

Heuristic h_2 is the most permissive as an LPM is only compared with its parent and can have the same utility. On the contrary, Heuristic h_3 is the most



(a)

heuristic / exp. seq.	h_1	h_2	h_3	h_4
sq_1: □	✗	✗	✗	✗
sq_2: ×	✗	✓	✗	✗
sq_3: ●	✓	✓	✗	✗
sq_4: ○	✓	✓	✗	✓
sq_5: △	✓	✓	✓	✓

(b)

Fig. 3. *(a)* Four successive expansions on five initial LPMs, *(b)* the pruning scope of the different heuristics.

restrictive as an LPM is compared with its best ancestor and must have a higher utility. As heuristics are approximate methods, some LPMs will be wrongly pruned. In the example in Fig. 3, the expansion line sq_4 would have been pruned by heuristic h_3 after $sq_{4,3}$. However, we notice that the LPM produced in the next expansion has a utility higher than the best ancestor. This is an example of expansion line that shouldn't have been stopped. Therefore, a good strategy will be a compromise between the number of good HU-LPMs we allow to lose and how small the search space has become thanks to the pruning methods.

5 Experiments

In this section, we present experiments to evaluate the HU-LPM mining approaches presented in this paper. First, we explore whether the different *heuristic configurations* achieve the goal of reducing the search space size, and assess the quality of the obtained HU-LPMs. Here, a *heuristic configuration* is considered to be the combination of a heuristic and the value of k. Then, we explore the relation between (1) the performance of the heuristic configurations in terms of search space reduction and the quality of the mined HU-LPMs, and (2) properties of the event log. We detail the experimental setup in Sect. 5.1, discuss the results in Sect. 5.2. We then continue by exploring how the performance of the heuristics are impacted by characteristics of the event logs. In Sect. 5.3 we analyze the effect of log properties *in isolation* on the performance of the heuristics using statistical testing. In Sect. 5.4 we investigate more complex effects on the performance of the heuristics *that involve multiple log properties together.*

5.1 Methodology

To evaluate the performance of the different heuristic configurations we apply the mining configurations to a collection of event logs consisting of 3 existing event logs and a collection of artificially-generated event logs. The existing logs are the BPI'13 closed problems log, consisting of 1487 traces and 6660 events, the BPI'13 open problems log, consisting of 819 traces and 2351 events and an artificial log used in the Process Mining book [1] (Chap. 2), consisting of 6 traces and 42 events. Furthermore, considering our aim of exploring the relationship between properties of event logs and the performance of the heuristic configurations, we generate a collection of artificial logs where we aim to generate logs with diverse log properties. To generate the event logs we use the *PTandLogGenerator* tool [15], which allows for the generation of random process trees, requiring the user to set as parameters the percentage of operator nodes in the tree that are of each operator node type (i.e., sequence, choice, parallel, loop). We generate 27 unique process trees by setting the operator node probabilities according to the configuration shown in Table 1. Since the computation time of LPM mining is highly dependent on the number of activities in the log, it is important to also make the set of artificial logs diverse in their number of activities. Therefore, we randomly chose the number of activities to be used in each process trees from a triangular probability distribution with a minimum of 10 activities, a maximum of 30 activities, and a mode of 20 activities. We simulate 100 traces from each of the 27 process trees to generate 27 artificial event logs, providing us with a collection of event logs that is diverse in control-flow characteristics.

Table 1. The process tree properties of the artificially generated event logs.

Log	Act	Sequence	Loop	Parallel	Choice	Log	Act	Sequence	Loop	Parallel	Choice
1	21	0.4	0.0	0.0	0.6	15	15	0.5	0.1	0.2	0.2
2	15	0.4	0.0	0.1	0.5	16	18	0.5	0.2	0.0	0.3
3	12	0.4	0.0	0.2	0.4	17	23	0.5	0.2	0.1	0.2
4	18	0.4	0.1	0.0	0.5	18	13	0.5	0.2	0.2	0.1
5	20	0.4	0.1	0.1	0.4	19	17	0.5	0.0	0.0	0.5
6	22	0.4	0.1	0.2	0.3	20	14	0.5	0.0	0.1	0.4
7	13	0.4	0.2	0.0	0.4	21	17	0.5	0.0	0.2	0.3
8	19	0.4	0.2	0.1	0.3	22	13	0.6	0.1	0.0	0.3
9	14	0.4	0.2	0.2	0.2	23	14	0.6	0.1	0.1	0.2
10	21	0.5	0.0	0.0	0.5	24	17	0.6	0.1	0.2	0.1
11	16	0.5	0.0	0.1	0.4	25	14	0.6	0.2	0.0	0.2
12	14	0.5	0.0	0.2	0.3	26	13	0.6	0.2	0.1	0.1
13	22	0.5	0.1	0.0	0.4	27	18	0.6	0.2	0.2	0
14	15	0.5	0.1	0.1	0.3						

These 27 artificially generated event logs do not have a *cost* attribute that can be used for high-utility LPM mining. Therefore, we artificially add a cost attribute to each event in the log in two different ways. First, we add cost to the log in a way that the costs of events in the log are relatively homogeneous, i.e., the differences between how much utility the events can contribute to an LPM are fairly small. To add this type of costs to the log, we sample the cost of each event from a Normal distribution with a mean of 10 and a standard deviation of 1. Secondly, we explore the effect of a heavily skewed distribution of cost over the events, i.e., only a small number of events take up a large share of the total cost of the log. To generate this highly skewed costs, we sample the cost of each event from a Pareto distribution with a scale parameter of 1 and a shape parameter of 1. For each of the 27 event logs, we generate both a version with normally distributed costs and with Pareto distributed costs, leading to a total of 54 artificial logs.

For each event log, we first apply the HU-LPM discovery algorithm without any pruning, generating the desired list of HU-LPMs in terms of quality and providing us with the size of the full search space when all LPMs are explored. As a next step, we discover HU-LPMs with each of the four different heuristic strategies, and $\{1, 2, 3\}$ the values of parameter k. We limit the LPM expansion procedure to four successive expansions, i.e., $exp\_max = 4$, to be able to perform the experiments within a reasonable time. We compare the number of explored LPMs obtained with each of the heuristics with the number of LPMs explored when no pruning was applied.

As shown in Sect. 4, the heuristics might prevent the discovery of HU-LPMs with high utility, leading to HU-lists of lower quality. Let $\mathcal{S}_a = \langle M_1, M_2, \ldots, M_n \rangle$ be the HU-list obtained using heuristic a. We define $\mathcal{S}_{id}$ as the ideal HU-list; i.e., the HU-list extracted without any pruning. To assess the efficiency of the four heuristics, we compare the HU-list extracted with the heuristics with the ideal HU-list obtained with the existing HU-LPM mining technique [29] which does not use any pruning. The quality of the extracted HU-list depends on the utility of the HU-LPMs in the HU-list compared to the utility of the HU-LPMs in the ideal HU-list. We compare the HU-list and the ideal HU-list using the normalized Discounted Cumulative Gain (nDCG) [7], which is an evaluation metric for rankings that is one of the most commonly used metrics in the Information Retrieval field [28]. Discounted Cumulative Gain (DCG) measures the quality of a ranking based on the relevance of the elements in the ranking in such a way that it gives higher importance to the top positions in the ranking. We denote the relevance of the element of the ranking at position i with rel_i. For the evaluation of a HU-list we regard rel_i to be the utility of the LPM at position i. Equation 7 formally defines DCG over the first p elements of a ranking.

$$DCG_p = \sum_{i=1}^{p} \frac{2^{rel_i} - 1}{log_2(i + 1)} \tag{7}$$

IDCG is defined as the DCG obtain the optimal ranking, which is the ideal HU-list in our example. Equation 8 defines nDCG based on the DCG of a ranking and the IDCG of the respective ideal ranking.

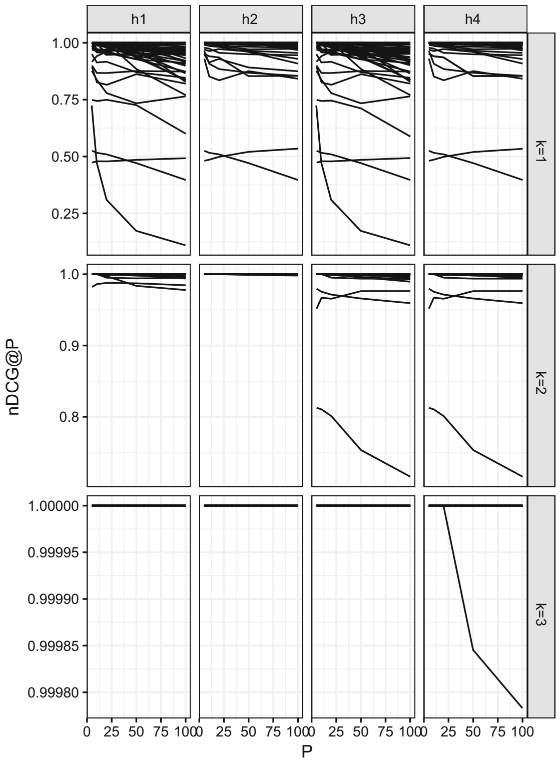
$$nDCG_p = \frac{DCG_p}{IDCG_p} \tag{8}$$

We limit the nDCG calculation to the p first HU-LPMs in the ranking, with $0 < p \leq |\mathcal{S}_a|$ for $\mathcal{S}_a$ being the HU-list obtained with pruning.

5.2 Aggregate Results

Figure 4a shows the nDCG_p results for different values of p, for each of the four heuristics and the three values of k. The x-axis represents the value of p used to compute nDCG_p, while the nDCG_p is shown on the y-axis. Each line in this plot represents one of the 54 artificially generated event logs or one of the three existing event logs. The figure shows that all four heuristics achieve nDCG scores close to 1 on most event logs. Heuristic $h1$ and $h3$ result in the lowest nDCG values, especially when used with $k = 1$. Figure 4b shows the median ratio of the search space (i.e., compared to exploring the full search space without heuristic) of the four heuristics and the values of k, over all 57 event logs. It shows that heuristics $h1$ and $h3$ lead to the largest reduction of the search space, where at median only 16% of the full search space without using any pruning is explored. When using $k = 3$, heuristics $h1$, $h2$ and $h3$ result in doing a full exploration of the search space, resulting in an nDCG value of 1.0 on all event logs and a median search space ratio of 1.0. With $h4$, there is one single event log for which $k = 3$ does not result in a full exploration of the search space, as indicated by the single line with an nDCG lower than 1.0. The general trend of the nDCG values on the majority of logs is that the nDCG_p decreases for higher numbers of p. This indicates that the majority of the LPMs that were missed in the mining procedure as a result of using a heuristic were positioned at lower positions of the ranking in the ground truth ranking, indicating that the most of the top LPMs still get found when mining with heuristics. Figure 4b also shows the median absolute deviation of the search space ratio over the 57 event logs. The low median absolute deviation values show that the search space reduction is consistent over all event logs. In total, $h1$ and $h3$ with $k = 1$ enable substantial pruning of the search space with some loss in nDCG, while with $k = 2$ the search space is still reduced, with almost no loss in nDCG. We highlight that all of the low nDCG value (<0.6) results, over all configurations, originate from the same log: log #1 with a Pareto distribution of the cost, and log #3 with both the normal and the Pareto distribution of the cost.

Note that we do not show the speedup of HU-LPM mining in terms of CPU time, but instead focus on the reduction of the search space. The reason that we do not present CPU time is that exploration of the full search space without using any pruning on a few of the logs results a memory usage of the miner that require the usage of virtual memory in the form of swap files on hard disk. In these circumstances, the CPU time is influenced heavily by characteristics of memory management by the operating system, while we are interested in measuring the effect of pruning on the efficiency of HU-LPM mining. However, the full mining on the different logs takes at median less that 3 min (177 s). An exception is log #23, where exploration of the full search space takes 1.5 h due to memory management by the operating system.



(a)

		Search Space Ratio	
H	**k**	**median**	**med. abs. dev.**
h_1	**1**	**16%**	10%
	2	84%	8%
	3	100%	0%
h_2	**1**	32%	8%
	2	88%	7%
	3	100%	0%
h_3	**1**	**16%**	10%
	2	75%	16%
	3	100%	0%
h_4	**1**	32%	8%
	2	75%	16%
	3	99%	2%

(b)

Fig. 4. *(a)* The nDCG values for each configuration on the different logs and *(b)* the reduction in the number of LPMs that have to be explored as part of the mining procedure for each configuration on the different logs.

5.3 The Influence of Log Properties on Heuristic Performance

We now list the event logs properties that we use to our analysis:

Number of events & activities. The number of events and activities in the log. The number of activities is expected to negatively impact the size of the search space, as the number of expansion operations of an LPM depends on the number of activities.

Number of trace variants. The number of unique traces in the log. This property is expected to influence the size of the search space as this determines the number of times per LPM that the alignment algorithm [2] needs to be executed.

Average number of event & activities per trace. The average trace length, and the average number of activities per trace.

Maximum trace length. The length of the longest trace in the log. This property is expected to influence the size of the search space, as the computation time of the alignment algorithm [2] is dependent on the trace length.

Furthermore, to capture the control-flow properties of the event log, we extract several properties from the *directly-follows graph* (dfg) that we extract from the event logs. A dfg of an event log L is a weighted directed graph over the alphabet of activities Σ where two activities $a, b \in \Sigma$ are connected by an edge if a is directly followed by b somewhere in a trace of L. The *weight* of the edge is proportional to the *frequency* of how often this happens. The network science research field [20] is concerned with the analysis of graphs and the extraction of descriptive properties from a graph. We extract the following properties from the dfg, using well-known techniques from the network science field:

Graph diameter. let the *geodesic distance* between two vertices a, b of a graph G be the number of edges in the shortest path from a to b, where the shortest path is based on the *frequency* of the edges in the dfg. The *eccentricity* of a vertex a of a graph is defined as the maximum geodesic distance of a with any other vertex b of G. The *Graph diameter* is the highest eccentricity of all vertices in a graph. Graph diameter can be seen as a proxy variable to measure the degree of connectedness of the activities in the log.

Graph betweenness centrality. [11] If X is the set of vertices of a graph, the *betweenness centrality* of a vertex $x_1 \in X$ is the ratio of shortest paths between any two vertices $x_2, x_3 \in X$ that go through x_1. We lift betweenness centrality from vertices to the graph level by averaging the betweenness centrality of the vertices. A short distance between two nodes a and b on the dfg means that both are quite unrelated (they have weak links in terms of succession frequency), a vertex that has a high betweenness centrality is in fact quite on the outskirts of the graph (not strongly linked with many vertices). The betweenness centrality of a dfg can be regarded as a measure of *sequentialness* of an event log: the betweenness centrality will be high if the event log is sequential in nature to a high degree, while it will be low if the process is highly parallel or random.

Graph density. The number of edges in the graph divided by the number of possible edges in the graph, i.e., $|\Sigma|^2$ for a dfg.

PageRank-based graph properties. PageRank [25] is one of the most well-known algorithms to measure the relative importance of vertices in a graph. PageRank is an iterative algorithm that at each iteration updates the importance of a vertex taking into account the importance of the vertices that link to it, eventually ended in a steady-state. Xing et al. [31] later extended PageRank to graphs with weighted edges. We apply the weighted PageRank extension to the dfg, which results in frequent activities that are also frequently followed and preceded by other frequent activities getting high PageRank scores, while infrequent activities, or activities that are mainly followed and preceded by activities getting low PageRank scores. Many high-PageRank nodes in the graph can be seen as an indication that there are many frequent LPMs in the log. From the vertex PageRank scores, we extract the **maximum value** and the **variance** as event log properties. Logs with a high PageRank variance would indicate they have both very strong and very weak activities. Therefore, we expect to be few LPMs, thus resulting in a lot of pruning and good nDCG scores.

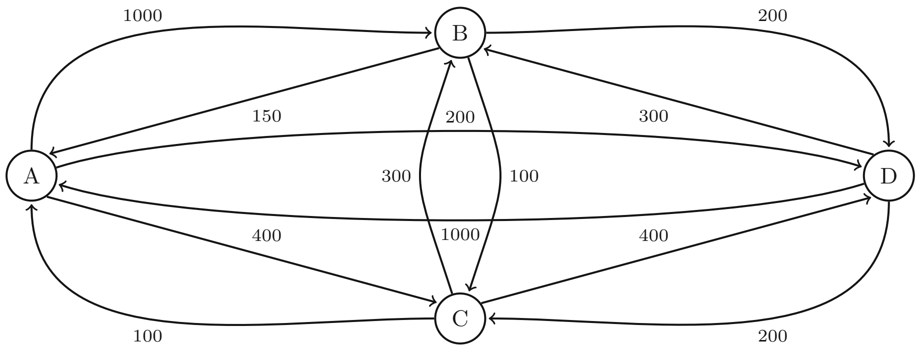


Fig. 5. Example of utility directly-follows graph for the log of Fig. 2a.

The dfg based features capture different elements of the control-flow of the event log. In this work, however, we are especially interested in the utility of the events in the process model, which is not captured in the dfg. The rationale behind this is that for logs where the costs are equally divided over the events, there will be a larger set of strong HU-LPM in the log, therefore leading to a larger search space size. We calculate the **Gini index** [12], one of the most commonly used inequality measures, to measure the degree of inequality between the cost values of the events in the log. Additionally, we define a special version of the dfg that defines that arc weights between nodes based on utility, which we call the *utility dfg*, and we extract each graph-based event log property for the utility dfg that we have above defined for the dfg.

Figure 5 shows an example of a utility dfg that is based on the log shown in Fig. 2a. Each activity is represented by a node, and the edges between two nodes a and b are weighted by the total sum of the *cost* attribute of events of activity b that follow an event of activity a. In the example, the edge linking from activity B to D has weight 200 because the log contains a single D event that directly follows a B, and this event has a cost of 200.

We analyze the relation between the listed event log properties and the search space ratio as well as the relation between those properties and the quality of the obtained results in terms of nDCG@100. For each heuristic and for each value of k we calculate the Kendall τ rank correlation [17] between value of the log property and the search space ratio, testing whether there is a relation between the value of the property and the search space ratio using as sample the collection of logs. We repeat the same analysis for nDCG@100. The correlation analysis yields a value in $[-1, 1]$ interval, where -1 indicates a strong negative correlation, 0 indicates no correlation, and 1 indicates a strong positive correlation. Kendall's τ rank correlation test is a non-parametric test, i.e., it does not make any assumptions on how the data is distributed, in contrast to for example a t-test. In addition to reporting the correlation value we test whether the relation is statistically significant, using a τ test with significance level α of 0.05.

Table 2 shows the correlation values between the event log properties and the search space ratio, indicating the statistically significant relations in bold. Looking at the number of statistically significant correlations, it seems that the number of events, the number of trace variants, the number of activities per trace, the Gini index of the total utility of the traces, and the maximum pagerank of the dfg are most strongly related to the search space ratio, aggregated over all heuristics.

Table 2. Kendall τ correlation coefficients between log properties and search space ratio. Statistically significant ($\alpha = 0.05$) effects are indicated in bold.

Property	h1		h2		h3		h4	
	k = 1	k = 2	k = 1	k = 2	k = 1	k = 2	k = 1	k = 2
Num_events	**0.51**	−0.04	0.04	−0.16	**0.51**	**−0.31**	0.05	**−0.32**
Num_activities	**−0.21**	−0.19	−0.17	−0.15	**−0.21**	0.02	−0.18	0.01
Num_trace_variants	**0.44**	−0.07	−0.01	−0.17	**0.44**	**−0.33**	0.00	**−0.33**
Avg_num_events_per_trace	**0.52**	−0.06	0.13	−0.18	**0.52**	**−0.20**	0.14	**−0.20**
Avg_num_activities_per_trace	**0.46**	−0.10	0.13	**−0.18**	0.46	−0.17	0.14	−0.17
Max_trace_length	**0.56**	0.06	0.15	−0.08	**0.56**	−0.17	0.14	−0.17
Gini_index_event_utility	0.09	−0.07	−0.09	−0.12	0.10	−0.19	−0.09	**−0.19**
Gini_index_trace_utility	−0.07	−0.16	**−0.21**	−0.16	−0.06	**−0.21**	**−0.21**	**−0.22**
Dfg_graph_diameter	**0.24**	−0.04	0.10	−0.10	**0.25**	−0.08	0.11	−0.10
Dfg_betweenness_centrality	0.00	0.02	0.10	−0.10	**0.25**	−0.08	0.11	−0.10
Dfg_graph_density	0.11	−0.05	−0.08	−0.06	0.12	−0.15	−0.07	−0.14
Dfg_pagerank_max	**0.23**	−-0.06	−0.11	−0.13	**0.23**	**−0.24**	−0.10	**−0.24**
Dfg_pagerank_variance	**0.26**	**0.11**	0.09	0.07	**0.25**	−0.06	0.10	−0.06
Udfg_graph_diameter	**0.27**	0.02	0.17	−0.05	**0.28**	−0.03	0.18	−0.03
Udfg_betweenness_centrality	−0.05	−0.12	−0.05	−0.11	−0.05	0.00	−0.06	−0.02
Udfg_pagerank_max	0.18	−0.07	−0.08	−0.12	0.19	**−0.22**	−0.07	**−0.22**
Udfg_pagerank_variance	**0.26**	0.11	0.09	0.07	**0.25**	−0.06	0.10	−0.06

Table 3 shows the obtained correlation values between the event log properties and the nDCG@100, indicating the statistically significant relations in bold. The results show that the maximum pagerank of the dfg graph as well as of the utility dfg graph that are extracted from the log are significantly correlated with the nDCG@100 of the obtained LPMs, for almost all of the heuristics and values of k. Note that this is a negative correlated, indicating that all heuristics lead to suboptimal results when these graphs are dominated by one strong central node. Furthermore, it is noteworthy that the length of the longest trace has a negative influence on the quality of the results.

Table 3. Kendall τ correlation coefficients between log properties and nDCG@100. Statistically significant ($\alpha = 0.05$) effects are indicated in bold.

Property	h1		h2		h3		h4	
	k = 1	k = 2	k = 1	k = 2	k = 1	k = 2	k = 1	k = 2
Num_events	0.12	**−0.27**	−0.19	−0.17	0.12	**−0.24**	−0.18	**−0.29**
Num_activities	0.03	0.12	0.19	0.14	0.03	0.19	**0.21**	0.14
Num_trace_variants	0.11	**−0.24**	−0.19	−0.07	0.10	−0.20	−0.19	**−0.25**
Avg_num_events_per_trace	**0.21**	−0.19	−0.12	−0.18	**0.20**	−0.13	−0.11	−0.12
Avg_num_activities_per_trace	**0.27**	−0.12	−0.03	−0.14	**0.26**	−0.02	−0.03	−0.01
Max_trace_length	0.13	−0.21	**−0.22**	−0.12	0.12	**−0.25**	**−0.20**	**−0.26**
Gini_index_event_utility	−0.02	0.01	−0.08	−0.02	−0.02	−0.05	−0.09	−0.06
Gini_index_trace_utility	−0.07	0.01	−0.07	0.03	−0.07	−0.01	−0.07	−0.05
Dfg_graph_diameter	**0.20**	−0.20	0.02	−0.22	**0.20**	−0.07	0.04	−0.11
Dfg_betweenness_centrality	0.11	−0.04	0.17	−0.06	0.11	0.04	0.19	−0.01
Dfg_graph_density	−0.05	0.00	−0.18	0.10	−0.05	−0.07	−0.20	−0.04
Dfg_pagerank_max	**−0.22**	**−0.28**	**−0.47**	−0.18	**−0.24**	**−0.36**	**−0.49**	**−0.32**
Dfg_pagerank_variance	−0.03	−0.08	**−0.21**	0.01	−0.03	−0.17	**0.23**	−0.12
Udfg_graph_diameter	**0.22**	−0.16	0.01	−0.18	**0.21**	−0.11	0.02	−0.08
Udfg_betweenness_centrality	0.14	0.04	0.19	0.02	0.13	0.13	**0.20**	0.09
Udfg_pagerank_max	−0.19	**−0.26**	**−0.43**	−0.14	**−0.20**	**−0.34**	**−0.44**	**−0.30**
Udfg_pagerank_variance	−0.03	−0.08	**−0.21**	0.01	−0.03	−0.17	−0.23	−0.12

5.4 The Influence of Combinations of Multiple Log Properties on Heuristic Performance

We leverage regression techniques to analyze interplay between properties of the event logs on the one hand and computation time, search space size, and quality of the LPMs in terms of nDCG on the other hand. We chose the *regression trees* and *linear regression* regression techniques, since both techniques produce interpretable models.

The goal is to predict a defined response variable, i.e., the search space size or the nDCG result. To estimate the accuracy of the models we use the *leave-one-group-out cross-validation* model evaluation setup, where we train the model on the observations from all but one event log in the learning phase and then predict and evaluate the prediction performance on the one event log left out. This process is then repeated 57 times (for every log) to predict and evaluate for every event log. The accuracy of the model is the average accuracy over all logs. We evaluate the predicted value with respect to the ground truth value using MAPE (Mean Absolute Percentage Error), which is defined as:

$$MAPE = \frac{100}{n} \sum_{i=1}^{n} \left| \frac{A_i - P_i}{A_i} \right| \tag{9}$$

with n the number of predicted values, A_i the actual ground truth values and P_i the predicted value by the model. This measure indicates, in percentage, how far the prediction is from the actual value, without making the distinction between a prediction above or below the actual value.

Figure 6 shows the discovered regression tree for the relation between search space ratio and event log properties. Each node of the tree represents a binary split on one of the properties. Furthermore, each node in the tree is annotated with the number of observations (samples) in the training set taking this path in the tree, and the predicted target value for that node in terms of search space ratio. We left out $k = 3$ values from the analysis, since the heuristics have shown to result in a search space ratio of 1.0 for all logs in Sect. 5.2. The first split splits the tree into a $k = 1$ and $k = 2$ path, where it is noticeable that the predicted search space ratios are much higher for $k = 2$. However, for the $k = 1$ path we still see that the search space ratio depends on some properties of the log: especially the variance of the PageRank of the dfg turns out to be of influence: both the lowest and the highest PageRank variance values correspond to higher search space ratio, while the search space ratio turns out to be lower for middle-range values. For the $k = 2$ path we see $h1$ and $h2$ result in higher search space size compared to $h4$, which is in agreement with what we have found before. However, the tree also shows that higher numbers of activities in the log ($\geq$16.5) result in higher search space size.

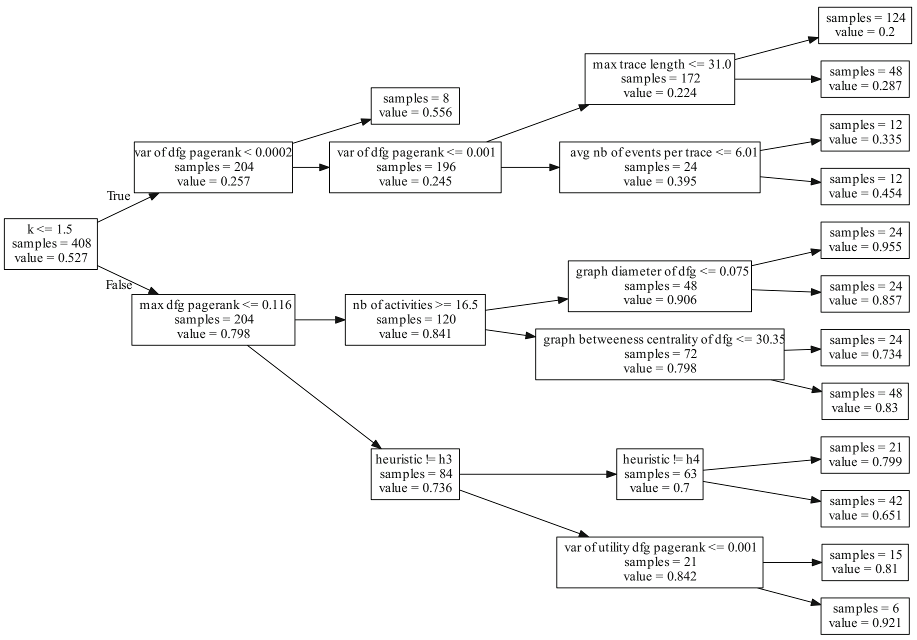


Fig. 6. Regression tree for the relation between log properties and search space ratio.

Figure 7 shows a regression tree built with nDCG_{100} as the variable to predict. Most of the features used in this regression tree are event log properties; e.g., the total number of activities, the number of events or the average number of

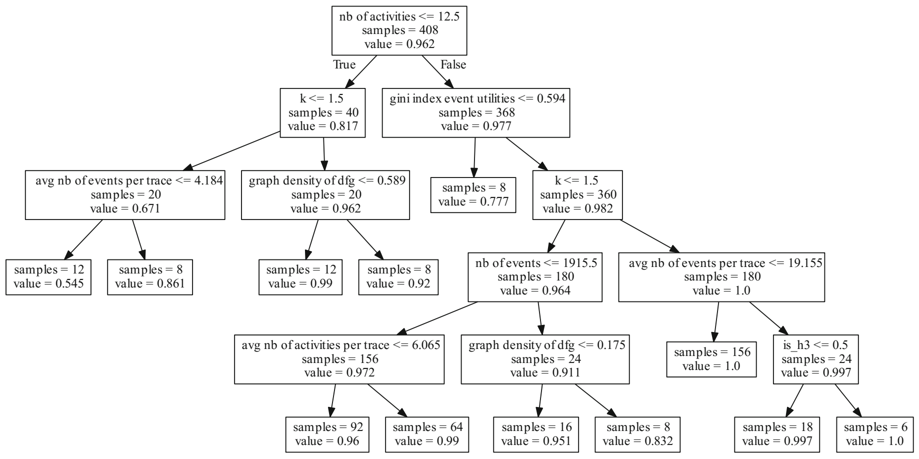


Fig. 7. Regression tree for the relation between log properties and the quality of the mined HU-LPMs (in terms of nDCG_{100}).

events per trace. This shows that regardless of the heuristic configuration used, the logs structural properties have a larger impact on the quality of the HU-LPM ranking extracted than the chosen heuristic. The tree shows that mining with $k = 2$ leads to higher nDCG values than $k = 1$. The tree also surprisingly shows that the nDCG is higher for event logs with 13 or more activities. Event logs where the Gini index of the event utilities is low, i.e., where the utility if equally distributed over the events, result in lower nDCG values. This can be explained by the fact that with more evenly distributed event utility, there are more strong HU-LPMs in the log. This leads to a larger unpruned search space size, and to a harder pruning task where it is easier to miss LPMs with high utility.

Table 4 presents the prediction results for the search space ratio and nDCG_{100} predictions in terms of MAPE. The table contains two baseline prediction methods. The first baseline is the Global Mean (GM), i.e., respectively the search space ratio and average nDCG_{100} values averaged over all event logs, used as a static predictor. As a second baseline, we use the Configuration-based Mean (CBM), i.e., the average search space ratio and average nDCG_{100} values for all the runs with identical heuristic and value of parameter k. The CBM baseline performs better than the global mean, which shows that the search space ratio and nDCG_{100} values are dependent on the heuristic and the value of parameter k. The regression trees that are shown in Figs. 6 and 7 make more accurate predictions than the configuration-based mean, which shows that there is a benefit in using information about the event log in predicting the search space size and the nDCG values. However, a linear regression model outperforms the regression tree models for both prediction tasks. It is noticeable that the MAPE values for predicting the search space ratio are very high, while the errors for predicting the nDCG_{100} values are very small. The reason for this is that the search space ratio covers a large interval, with some values close to zero and others close

Table 4. MAPE average and standard deviation values of different prediction techniques

Technique	MAPE of the search space ratio prediction		MAPE for the nDCG_{100} prediction	
	Mean	Std. dev.	Mean	Std. dev.
Global mean	825.83	927.21	4.32	7.84
Configuration-based mean	382.75	463.84	3.97	7.62
Regression tree	121.63	159.41	3.57	8.20
Linear regression	**86.60**	21.81	**3.15**	7.12

to one, while the values of nDCG are much more concentrated towards values close to one. This makes the prediction of search space ratio percentage-wise a much harder prediction task than predicting nDCG. However, the fact that the prediction errors when using log-based information are several orders smaller than the baseline methods shows that the search space ratio can be predicted with reasonable accuracy when log properties are taken into account. While this predictive experiment was performed with the aim of generating insights in the relation between log properties and the performance of the heuristics, one could also envision these predictive models to be used as guidance for heuristic selection for the process analyst. Given an event log, the prediction models could provide the analyst with the predicted running time (proportional to the search space ratio) and the predicted HU-LPM quality for each heuristic and value of k, aiding the process analyst in making this choice.

6 Conclusion and Future Work

In this paper, we have shown that the High-Utility Local Process Model (HU-LPM) mining problem is not anti-monotonic when event-level or trace-level utility functions are used. We introduced four heuristic configurable mining techniques to reduce the search space. We have shown on a collection of 57 event logs that the heuristics techniques can be used to extract HU-LPMs by exploring a search space several times smaller than the full search space, without much loss in the quality of the mined HU-LPMs. Additionally, we have discovered several properties of event logs that influence which of the four heuristic mining techniques work well, and have developed a predictive model that can predict the expected search space size and the expected quality of results based on log properties.

In future work, we intend to take the idea to make the approach log-specific one step further, by using the event log properties and a predictive model to predict at each step of the HU-LPM mining algorithm which LPMs can be pruned and which ones need to be explored. In this way, a log-specific heuristic mining technique would arise.

References

1. van der Aalst, W.M.P.: Process Mining: Data Science in Action. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49851-4
2. van der Aalst, W.M.P., Adriansyah, A., van Dongen, B.F.: Replaying history on process models for conformance checking and performance analysis. WIREs: DMKD **2**(2), 182–192 (2012)
3. van der Aalst, W.M.P., Weijters, A.J.M.M., Maruster, L.: Workflow mining: discovering process models from event logs. IEEE TKDE **16**(9), 1128–1142 (2004)
4. Bergenthum, R., Desel, J., Lorenz, R., Mauser, S.: Process mining based on regions of languages. In: Alonso, G., Dadam, P., Rosemann, M. (eds.) BPM 2007. LNCS, vol. 4714, pp. 375–383. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-75183-0_27
5. Buijs, J.C.A.M., van Dongen, B.F., van der Aalst, W.M.P.: A genetic algorithm for discovering process trees. In: IEEE CEC, pp. 1–8. IEEE (2012)
6. Buijs, J.C.A.M., Reijers, H.A.: Comparing business process variants using models and event logs. In: Bider, I., Gaaloul, K., Krogstie, J., Nurcan, S., Proper, H.A., Schmidt, R., Soffer, P. (eds.) BPMDS/EMMSAD -2014. LNBIP, vol. 175, pp. 154–168. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-43745-2_11
7. Burges, C., et al.: Learning to rank using gradient descent. In: ICML, pp. 89–96. ACM (2005)
8. Dalmas, B., Tax, N., Norre, S.: Heuristics for high-utility local process model mining. In: ATAED. CEUR (2017)
9. Dave, U., Patel, S.V., Shah, J., Patel, S.V.: Efficient mining of high utility sequential pattern from incremental sequential dataset. IJCA (2015)
10. van Dongen, B.F., de Medeiros, A.K.A., Verbeek, H.M.W., Weijters, A.J.M.M., van der Aalst, W.M.P.: The ProM framework: a new era in process mining tool support. In: Ciardo, G., Darondeau, P. (eds.) ICATPN 2005. LNCS, vol. 3536, pp. 444–454. Springer, Heidelberg (2005). https://doi.org/10.1007/11494744_25
11. Freeman, L.C.: A set of measures of centrality based on betweenness. Sociometry **40**, 35–41 (1977)
12. Gini, C.: Concentration and dependency ratios. Riv. Di Polit. Econ. **87**, 769–792 (1997)
13. Han, J., Cheng, H., Xin, D., Yan, X.: Frequent pattern mining: current status and future directions. DMKD **15**(1), 55–86 (2007)
14. International Organization for Standardization: ISO/IEC 19505–1:2012 - Information technology - Object Management Group Unified Modeling Language (OMG UML) - Part 1: Infrastructure (2012)
15. Jouck, T., Depaire, B.: PTandLogGenerator: a generator for artificial event data. In: BPM (2016)
16. Keller, G., Scheer, A.W., Nüttgens, M.: Semantische Prozeßmodellierung auf der Grundlage" Ereignisgesteuerter Prozeßketten". Inst. für Wirtschaftsinformatik (1992)
17. Kendall, M.G.: A new measure of rank correlation. Biometrika **30**(1/2), 81–93 (1938)
18. Leemans, M., van der Aalst, W.M.P.: Discovery of frequent episodes in event logs. In: Ceravolo, P., Russo, B., Accorsi, R. (eds.) SIMPDA 2014. LNBIP, vol. 237, pp. 1–31. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-27243-6_1
19. Leemans, S.J.J., Fahland, D., van der Aalst, W.M.P.: Discovering block-structured process models from event logs containing infrequent behaviour. In: Lohmann, N., Song, M., Wohed, P. (eds.) BPM 2013. LNBIP, vol. 171, pp. 66–78. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-06257-0_6

20. Lewis, T.G.: Network Science: Theory and Applications. Wiley, Hoboken (2011)
21. Liesaputra, V., Yongchareon, S., Chaisiri, S.: Efficient process model discovery using maximal pattern mining. In: Motahari-Nezhad, H.R., Recker, J., Weidlich, M. (eds.) BPM 2015. LNCS, vol. 9253, pp. 441–456. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23063-4_29
22. Maggi, F.M., Mooij, A.J., van der Aalst, W.M.P.: User-guided discovery of declarative process models. In: IEEE CIDM, pp. 192–199. IEEE (2011)
23. Măruşter, L., van Beest, N.R.T.P.: Redesigning business processes: a methodology based on simulation and process mining techniques. KIS **21**(3), 267 (2009)
24. Object Management Group: Notation (BPMN) version 2.0. OMG Specification (2011)
25. Page, L., Brin, S., Motwani, R., Winograd, T.: The PageRank citation ranking: bringing order to the web. Technical report, Stanford InfoLab (1999)
26. Srikant, R., Agrawal, R.: Mining sequential patterns: generalizations and performance improvements. In: Apers, P., Bouzeghoub, M., Gardarin, G. (eds.) EDBT 1996. LNCS, vol. 1057, pp. 1–17. Springer, Heidelberg (1996). https://doi.org/10.1007/BFb0014140
27. Tax, N., Sidorova, N., van der Aalst, W.M.P., Haakma, R.: Heuristic approaches for generating local process models through log projections. In: CIDM, pp. 1–8. IEEE (2016)
28. Tax, N., Bockting, S., Hiemstra, D.: A cross-benchmark comparison of 87 learning to rank methods. IPM **51**(6), 757–772 (2015)
29. Tax, N., Dalmas, B., Sidorova, N., van der Aalst, W.M.P., Norre, S.: Interest-driven discovery of local process models. arXiv preprint arXiv:1703.07116 (2017)
30. Tax, N., Sidorova, N., Haakma, R., van der Aalst, W.M.P.: Mining local process models. JIDE **3**(2), 183–196 (2016)
31. Xing, W., Ghorbani, A.: Weighted PageRank algorithm. In: CNSR. pp. 305–314. IEEE (2004)
32. Yin, J., Zheng, Z., Cao, L.: USpan: an efficient algorithm for mining high utility sequential patterns. In: SIGKDD, pp. 660–668. ACM (2012)
33. Zida, S., Fournier-Viger, P., Wu, C.-W., Lin, J.C.-W., Tseng, V.S.: Efficient mining of high-utility sequential rules. In: Perner, P. (ed.) MLDM 2015. LNCS (LNAI), vol. 9166, pp. 157–171. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21024-7_11
34. Zihayat, M., Wu, C.W., An, A., Tseng, V.S.: Mining high utility sequential patterns from evolving data streams. In: ASE BD&SI, p. 52. ACM (2015)

On Stability of Regional Orthomodular Posets

Luca Bernardinello[1](✉), Carlo Ferigato[2], Lucia Pomello[1], and Adrián Puerto Aubel[1]

[1] Dipartimento di Informatica, Sistemistica e Comunicazione, Università degli Studi di Milano-Bicocca, viale Sarca 336-U14, 20126 Milan, Italy
luca.bernardinello@unimib.it

[2] European Commission, Joint Research Centre (JRC), via E. Fermi 2749, 21027 Ispra, VA, Italy

Abstract. The set of regions of a transition system, ordered by set inclusion, is an orthomodular poset, often referred to as quantum logic, here called *regional logic*. Regional logics, which are known to be regular and rich, are the main subject of investigation in this work. Given a regular, rich logic L, one can build a transition system A, such that L embeds into the regional logic of A. Call a logic *stable* if the embedding is an isomorphism. We give some necessary conditions for a logic to be stable, and show that under these, the embedding has some stronger property. In particular, we show that any $\{0,1\}$-pasting of n stable logics is stable, and that, whenever L contains n maximal Boolean sublogics with pairwise identical intersections, L is stable. The full characterization of the class of stable logics is still an open problem.

1 Introduction

Regions of transition systems have been introduced by Ehrenfeucht and Rozenberg [7,8] and applied to the *synthesis of net systems* in several ways, as described in [1,2]. They correspond to *conditions* (or Boolean places) in elementary net systems; if we think of such a net system as a model of a distributed system, regions can be taken as propositions expressing properties that can be observed "locally" by a component of the system.

This notion of locality is subordinated to that of sequential component. As such it can be interpreted as spatial locality. Namely, a property is local whenever it is fully determined by the state of a single component, subject to a single clock. In this sense, the synthesis procedure allows for a full description of the implementation of a distributed system, provided its global behaviour specified by a transition system. The notion of global property is here naturally understood as follows. A state of the transition system is global in that it can be seen as composed by the states of each of its local sequential components. It must then correspond to a collection of locally observable properties, which is

M. Koutny et al. (Eds.): ToPNoC XIII, LNCS 11090, pp. 52–72, 2018.
https://doi.org/10.1007/978-3-662-58381-4_3

complete in that it determines uniquely a property, or region, for each of its components.

We are interested in the order structure of regions both when they are seen concretely as sets ordered by set inclusion and when they are seen abstractly as a partially ordered set of propositions; in general this set is a family of Boolean algebras, partially overlapping, usually called a *quantum logic* [11]. Each Boolean algebra corresponds, intuitively, to a component of the distributed system, and describes the logic of local propositions.

In more detail, we exploit a duality between *condition/event transition systems* and *rich* and *regular quantum* logics, presented in [4]. In that contribution, with some difference in notation that will be explained in a remark at the end of Sect. 2, we proved that, given a rich and regular quantum logic L (we use simply *logic* in what follows), one can build a condition/event transition system A, such that L embeds into the logic generated by the regions of A. In the present work, we explore the case in which this embedding is an isomorphism, in which case we call the logic *stable*. We give some necessary conditions for a logic to be stable, while the full characterization of the class remains an open problem.

The special cases analyzed in this paper exploit the fact that any quantum logic can be seen as a family of partially overlapping Boolean algebras. In particular, we show that the so called $\{0, 1\}$-pasting of n stable logics, is stable, and that, whenever L contains n maximal Boolean sublogics with pairwise identical intersections, L is stable.

In the next section, we recall basic definitions on condition/event transition systems (CETS) and regions, and on quantum logics, in particular on rich and regular logics. In Sect. 3 we recall that to any CETS it is possible to associate the quantum logic of its regions, and recall a synthesis procedure to associate a CETS to any rich and regular logic. Sections 4 and 5 constitute the original part of the paper: the first deals with some results towards the characterization of stable logics, the second considers some particular subclasses of logics, and shows that they are stable. Section 6 concludes the paper, suggesting some further development.

2 Preliminary Definitions and Notations

2.1 Transition Systems

Transition systems are a class of automata representing the *global* behaviour of a system in opposition to the representation via *local states* or *local properties*. A relation between these two ways of representation is extensively reported in the literature, see [7–9] as first papers on the subject and [1] for a complete survey. In this contribution, we will always consider *finite* transition systems.

Definition 1. *A* transition system *is a triple $A = (Q, E, T)$ where Q is a set of* states*, E is a set of* events *and $T \subseteq Q \times E \times Q$ is a set of* transitions*. We assume that the following conditions are respected:*

1. *the underlying graph of the* transition system *is connected;*

2. $\forall (q_1, e, q_2) \in T \quad q_1 \neq q_2$*;*
3. $\forall (q, e_1, q_1), (q, e_2, q_2) \in T \quad q_1 = q_2 \Rightarrow e_1 = e_2$*;*
4. $\forall e \in E \quad \exists\, (q_1, e, q_2) \in T$*.*

In some cases, we will drop axiom 1, which imposes connection.

A *region* of a transition system is a set of its states such that the occurrences of one of its events have the same *crossing relation*, namely entering, leaving the region itself, or otherwise neither of the two. This is formalised as follows.

Definition 2. *A* region *of a transition system* $A = (Q, E, T)$ *is a subset* r *of* Q *such that* $\forall e \in E,\ \forall (q_1, e, q_2), (q_3, e, q_4) \in T$*:*

1. $(q_1 \in r$ *and* $q_2 \notin r) \ \Rightarrow\ (q_3 \in r$ *and* $q_4 \notin r)$*;* and
2. $(q_1 \notin r$ *and* $q_2 \in r) \ \Rightarrow\ (q_3 \notin r$ *and* $q_4 \in r)$*.*

Given a transition system A, its set of regions will be denoted by $\mathcal{R}(A)$; given a state $q \in Q$, the set of regions containing q will be denoted by $\mathcal{R}_q(A)$ and, when the transition system is clear from the context, simply by $\mathcal{R}_q$. Note that the set of regions $\mathcal{R}(A)$ of a transition system $A = (Q, E, T)$ cannot be empty since at least the whole set of states Q is a region.

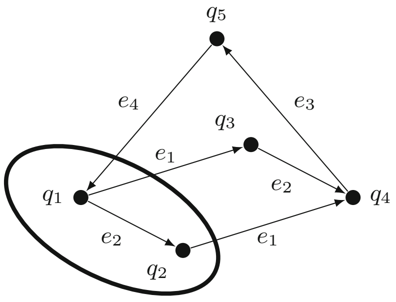


Fig. 1. The CETS A_0 and a region $v = \{q_1, q_2\}$

Example 1. Consider the transition system A_0 given in Fig. 1. The set $v = \{q_1, q_2\}$ is a region, in fact it is such that e_1 leaves it, e_4 enters it, and e_2 and e_3 do not cross its boundary. Clearly, the complement of v: $\{q_3, q_4, q_5\}$ is also a region, as well as $\{q_5\}$; whereas, for example, $\{q_4, q_5\}$ is not a region since transition (q_3, e_2, q_4) is entering whereas (q_1, e_2, q_2) does not cross its boundary. $\mathcal{R}_{q_1}(A_0) = \{\{q_1, q_2\}, \{q_1, q_3\}, \{q_1, q_2, q_5\}, \{q_1, q_3, q_5\}, \{q_1, q_2, q_3, q_4, q_5\}\}$.

Definition 3. *Let* $A = (Q, E, T)$ *be a transition system. The* pre-set *and* post-set *operations, denoted respectively by the operators* ${}^\bullet(\,.\,)$ *and* $(\,.\,)^\bullet$*, applied to regions* $r \in \mathcal{R}(A)$ *and events* $e \in E$ *are defined by:*

1. ${}^\bullet r = \{e \in E \mid \exists\, (q_1, e, q_2) \in T$ *such that* $q_1 \notin r$ *and* $q_2 \in r\}$*;*

2. $r^\bullet = \{e \in E \mid \exists\, (q_1, e, q_2) \in T$ *such that* $q_1 \in r$ *and* $q_2 \notin r\}$*;*
3. $^\bullet e = \{r \in \mathcal{R}(A) \mid e \in r^\bullet\}$*;*
4. $e^\bullet = \{r \in \mathcal{R}(A) \mid e \in {}^\bullet r\}$*.*

Example 2. Considering again Fig. 1 and the region $v = \{q_1, q_2\}$, $^\bullet v = \{e_4\}$, $v^\bullet = \{e_1\}$, $^\bullet e_1 = \{v, \{q_1, q_2, q_5\}\}$ and $e_1{}^\bullet = \{\{q_3, q_4\}, \{q_3, q_4, q_5\}\}$.

Condition/event transition systems have been introduced as the class of transition systems isomorphic to the *sequential case graphs* of *condition/event net systems* [8,10].

Definition 4. *A* Condition/Event Transition System *(CETS) is a transition system such that the following conditions are satisfied:*

1. $\forall\, q_1, q_2 \in Q \;\; \mathcal{R}_{q_1} = \mathcal{R}_{q_2} \Rightarrow q_1 = q_2$*;*
2. $\forall\, q_1 \in Q, \forall\, e \in E \;\; {}^\bullet e \subseteq \mathcal{R}_{q_1} \Rightarrow \exists\, q_2 \in Q$*:* $(q_1, e, q_2) \in T$*;*
3. $\forall\, q_1 \in Q, \forall\, e \in E \;\; e^\bullet \subseteq \mathcal{R}_{q_1} \Rightarrow \exists\, q_2 \in Q$*:* $(q_2, e, q_1) \in T$*.*

Basic facts concerning regions of transition systems [3] are listed in the following:

Proposition 1. *Let* $A = (Q, E, T)$ *be a transition system and* $\mathcal{R}(A)$ *its set of regions, then:*

1. $\emptyset \in \mathcal{R}(A)$*;*
2. $Q \in \mathcal{R}(A)$*;*
3. $r \in \mathcal{R}(A) \;\Rightarrow\; Q \setminus r \in \mathcal{R}(A)$*;*
4. $r_1, r_2 \in \mathcal{R}(A) \;\Rightarrow\; (r_1 \cap r_2 \in \mathcal{R}(A) \Leftrightarrow r_1 \cup r_2 \in \mathcal{R}(A))$*.*

We consider $\mathcal{R}(A)$ as enriched with the usual concrete structure. Elements, seen as subsets of A, are ordered by inclusion. Set union and intersection are here partial operations; $\mathcal{R}(A)$ is closed by set complement.

2.2 Quantum Logics and States on a Logic

We will follow the notation and definitions given in [11], but for the fact that we will consider only *finite* structures. In particular, the name used for the basic order structure defined in this section will be *quantum logic*, or simply *logic*. In the literature, quantum logics are known as well as *orthomodular posets*. This class is larger than that of *orthomodular lattices* since the operators of *greatest lower bound*, denoted $\wedge$, and *lowest upper bound*, denoted $\vee$—induced by the order relation—are not always defined. The following definition is taken from [11] (Definition 1.1.1).

Definition 5. *A* quantum logic *(or* logic*)* $L = (L, \leq, 0, 1, (\,.\,)')$ *is a partially ordered finite set* $(L, \leq)$ *endowed with a least and a greatest element, denoted by* 0 *and* 1*, respectively, and a unary operation* $(\,.\,)'$ *(called* orthocomplement*), such that the following conditions are satisfied:*
$\forall\, x, y \in L$

1. $x \leq y \ \Rightarrow \ y' \leq x'$*;*
2. $(x')' = x$*;*
3. $x \leq y' \ \Rightarrow \ x \vee y \in L$*;*
4. $x \leq y \ \Rightarrow \ y = x \vee (x' \wedge y)$*.*

This latter condition is sometimes referred to as orthomodular law*.*

Let $x, y \in L$ be such that $x \leq y'$, then we say that they are *orthogonal*, denoted $x \perp y$.

A *sublogic* of L is a subset $\hat{L}$ of L that is itself a logic with respect to the restrictions of the operation $(.)'$ and the relation $\leq$ to $\hat{L}$. In particular, $x \in \hat{L} \Rightarrow x' \in \hat{L}$ and, $\forall x, y \in \hat{L} \ \ x \leq y' \Rightarrow x \vee y \in \hat{L}$. A sublogic is *Boolean* if it is a Boolean algebra.

An element a of a logic is an *atom* if, for any element b of L such that $b \leq a$ either $b = a$ or $b = 0$. Atoms are the least elements in the logic except for the bottom element. A logic is said to be *atomic* if any element, except the bottom, is greater or equal to some atom. Finite logics are atomic. Atomic logics present the practical advantage to be determined by the relations among their *atoms*, and thus allow for a concise representation.

We say that two elements x and y in L are *compatible*, denoted $x \ \$ \ y$ if, and only if, there exist three mutually orthogonal elements $\hat{x}$, $\hat{y}$ and z in L such that $x = \hat{x} \vee z$ and $y = \hat{y} \vee z$. Intuitively, we can see maximal sets of mutually compatible elements in L as *maximal Boolean sublogics* of L. The following definition is taken from [11] (Definition 1.3.26).

Definition 6. *A logic L is called* regular *if, for any set $\{x, y, z\} \subseteq L$ of pairwise compatible elements, we have that $x \ \$ \ (y \vee z)$.*

The relation between compatible subsets of a logic L and Boolean sublogics of L is made clear by the following proposition, to be found in [11] (Proposition 1.3.29).

Proposition 2. *A logic L is* regular *if and only if every subset of pairwise compatible elements of L admits an enlargement to a* Boolean sublogic *of L.*

Example 3. The poset shown in Fig. 2 is a regular quantum logic. The set of its atoms is $\{v, w, x, y, z\}$. Examples of orthogonal pairs of elements are: $x \perp y$, $x \perp w$, $v \perp w$. It contains two maximal Boolean sublogics: $\{0, v, w, x, v', w', x', 1\}$ and $\{0, x, y, z, x', y', z', 1\}$, which intersect in the (non-maximal) Boolean sublogic $\{0, x, x', 1\}$. Note that x, x' respectively, is compatible with any other element of the logic, whereas for example $v \not\$ \ y$ and $v \not\$ \ z$.

Morphisms of logics are defined in such a way that they preserve order (and consequently orthogonality) and compatibility. The formal definition is borrowed from [11] (Definition 1.2.7)

Definition 7. *Let L_1 and L_2 be logics. A mapping $f : L_1 \rightarrow L_2$ is a* morphism of logics *if the following conditions are satisfied:*

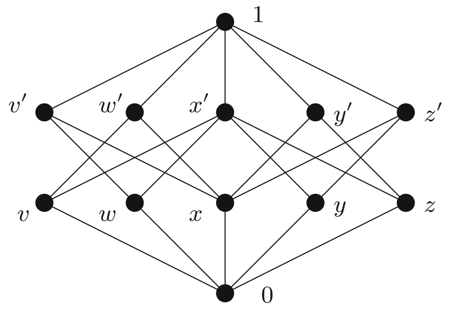


Fig. 2. A regular quantum logic.

1. $f(0) = 0$;
2. $\forall x \in L_1 \; f(x') = f(x)'$;
3. $\forall x, y \in L_1 \; x \perp y \;\Rightarrow\; f(x \vee y) = f(x) \vee f(y)$.

A morphism $f : L_1 \to L_2$ is an *isomorphism* if f is bijective, and f^{-1} is a morphism. Moreover, f is an *embedding* if $f(L_1)$ is a sublogic of L_2 and $f : L_1 \to f(L_1)$ is an isomorphism.

Example 4. Let us consider the logic L given in Fig. 2 and a logic L' isomorphic to the Boolean sublogic $\{0, x, x', 1\}$. Then, obviously there is an embedding $f : L' \to L$.

A crucial notion in the following is the concept of *two-valued state* which, in the case of regional logics, will allow to identify subsets of regions corresponding to states of the related condition/event transition systems. The following definition can be found in [11] (Definition 2.1.1).

Definition 8. *A* two-valued state *on a quantum logic L is a mapping $s : L \to \{0, 1\}$ such that:*

1. $s(1) = 1$;
2. $\forall x, y \in L \;\; x \perp y \;\Rightarrow\; s(x \vee y) = s(x) + s(y)$.

An immediate consequence of the definition is that a state s preserves order.

In [11] a distinction is made between *states*, that is mappings defined on L whose co-domain is the interval $[0, 1]$, and two-valued states as in Definition 8 above. Since we are using exclusively two-valued states, in what follows we will not make this distinction and we will call *states* the two-valued states of Definition 8.

If L is atomic, then every state will assign 1 to exactly one atom per maximal Boolean sublogic.

Given a logic L and $x \in L$, we denote by $\mathcal{S}(L)$ the set of all states on L and by $\mathcal{S}_x$ the set $\{s \in \mathcal{S}(L) \mid s(x) = 1\}$.

Definition 9. *Let L be a logic, and $X \subseteq L$, then the* up-closure *of X, denoted $\uparrow X$, is the set of all elements in L greater or equal to some element in X. Formally:*

$$\uparrow X := \{a \in L \mid \exists x \in X : x \leq a\}$$

This definition is particularly useful, for it allows to represent states in a concise way. This representation relies on a result from [4] (Proposition 29), slightly reformulated in the following proposition.

Proposition 3. *Any state s of a finite logic L is the characteristic function of a set $\uparrow X$, where X is a maximal set of pairwise incompatible atoms, such that it intersects each maximal set of mutually orthogonal atoms.*

Example 5. Consider the logic shown in Fig. 2. $v \not\lessgtr y$, $v \not\lessgtr z$, $w \not\lessgtr y$ and $w \not\lessgtr z$, and indeed its states are: $\uparrow\{v, y\} = \{v, y, w', x', z', 1\}$, $\uparrow\{v, z\}$, $\uparrow\{w, y\}$, $\uparrow\{w, z\}$ as well as $\uparrow\{x\}$. $\mathcal{S}_v = \{\uparrow\{v, y\}, \uparrow\{v, z\}\}$, $\mathcal{S}_x = \{\uparrow\{x\}\}$.

Logics which have "enough" states, in such a way that the order relation can be re-constructed by the evaluation of the states, are called *rich*.

Definition 10. *Let L be a logic and $x, y \in L$. L is* rich *if:*

$$\mathcal{S}_x \subseteq \mathcal{S}_y \;\Rightarrow\; x \leq y.$$

The converse property: $x \leq y \;\Rightarrow\; \mathcal{S}_x \subseteq \mathcal{S}_y$ is a consequence of condition 2. in Definition 8 above. A characterisation of rich logics, which will be of use in the contributions of the present work, is provided by the following Theorem ([11], Sect. 2.4.12).

Theorem 1. *Let L be a logic. Then L is rich if and only if $\forall a, b \in L \;\; a \not\lessgtr b \Rightarrow \mathcal{S}_a \cap \mathcal{S}_b \neq \emptyset$.*

Example 6. Consider again the logic shown in Fig. 2. It is rich, in fact, for example, $v \not\lessgtr y$ and $\mathcal{S}_v \cap \mathcal{S}_y = \{\uparrow\{v, y\}\}$ since $\mathcal{S}_v = \{\uparrow\{v, y\}, \uparrow\{v, z\}\}$ and $\mathcal{S}_y = \{\uparrow\{v, y\}, \uparrow\{w, y\}\}$.

A particular type of logic is known as *concrete logic.* A logic is called concrete if it can be represented as a collection of subsets of a given set Ω. In this case, the order relation is the set inclusion between subsets of Ω while the orthocomplement $(\,.\,)'$ is the set complement in Ω.

Definition 11. *The tuple $(\Delta, \subseteq, \emptyset, \Omega, (.)')$, where Δ is a collection of subsets of a given set Ω and $A \in \Delta \;\Rightarrow\; (A)' = \Omega \setminus A$, is a* concrete logic *if and only if the following conditions are satisfied:*

1. $\emptyset \in \Delta$;
2. $A \in \Delta \;\Rightarrow\; \Omega \setminus A \in \Delta$;
3. $\forall A_1, A_2 \in \Delta \;\; A_1 \cap A_2 = \emptyset \;\Rightarrow\; A_1 \cup A_2 \in \Delta$.

Note that in this *concrete* representation of a logic, orthogonality is equivalent to being pairwise disjoint, and two elements $A_1, A_2 \in \varDelta$ will be compatible if and only if $A_1 \cap A_2 \in \varDelta$. A more detailed presentation of *concrete logics* and the discussion on the satisfaction by *concrete logics* of properties 1–4 in Definition 5 can be found in [11], p. 2.

As shown in Sect. 3 below, the set of regions $\mathcal{R}(A)$ of a CETS A is a *concrete logic*. This is the reason why we are interested in concrete logics and in the following theorem (due to Stanley Gudder and reported in [11] as Theorem 2.2.1) that relates richness to concreteness:

Theorem 2. *A logic L is isomorphic (as a logic) to a concrete logic if and only if it is rich.*

The proof of this theorem uses a property of *duality* between L and the set of its states $\mathcal{S}(L)$: each element in $x \in L$ can be represented by the set $\mathcal{S}_x$ of the $s \in \mathcal{S}(L)$ such that $s(x) = 1$. Conversely, we note that states are in fact characteristic functions, and can therefore be interpreted as subsets of L. We will use the same *duality* in the next sections.

Remark. In the present paper, we have partly changed notation and terminology with respect to [4]. In particular, here we use 'rich and regular logic' to denote what was called 'prime and coherent orthomodular poset'; moreover, 'two-valued state' or 'state' of a logic was called 'prime filter' to stress the connection with the concept of ultrafilter in Boolean algebras.

3 Regional Logics and Synthesis of Saturated Transition Systems

In the following, we recall how to associate to any CETS a concrete regular logic, and how to construct a CETS starting from a rich and regular logic; afterwards, we discuss the relations between these two transformations.

Let us consider the properties of regions recalled in Proposition 1. By using these properties, it is possible to construct a logic starting from the regions of a transition system. More precisely, if $A = (Q, E, T)$ is a finite CETS and $\mathcal{R}(A)$ is its set of regions then $\mathcal{R}(A) = (\mathcal{R}(A), \subseteq, \emptyset, Q, (.)')$ (where $r \in \mathcal{R}(A) \Rightarrow (r)' = Q \setminus r$) is a rich and regular quantum logic as proved in [4]. Moreover, $\mathcal{R}(A)$ is a concrete logic as in Definition 11. We will say that $\mathcal{R}(A)$ is the *regional logic* of A, and that a logic L is *regional* if it is isomorphic to $\mathcal{R}(A)$ for some A.

Example 7. Consider the CETS A_0 given in Fig. 3. Its regional logic $\mathcal{R}(A_0)$ is isomorphic to the logic L given in Fig. 2, where for example v is $\{q_1, q_2\}$, w is $\{q_3, q_4\}$, x is $\{q_5\}$, y is $\{q_1, q_3\}$, z is $\{q_2, q_4\}$, and so on for their complements.

In [4], a *synthesis procedure* allowing to construct a CETS, denoted $\boldsymbol{\mathcal{A}}(L)$, starting from a rich and regular logic L was presented and set in categorical terms by showing the existence of two contravariant functors between the categories of CETS and rich and regular quantum logics.

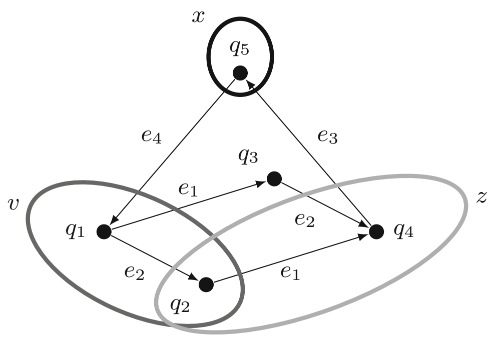


Fig. 3. Some regions of A_0: x and v are disjoint as subsets of states, so they are orthogonal elements of $\mathcal{R}(A_0)$ [see Fig. 2]. Analogously, v and z have a non-empty intersection as subsets of states, and since $v \cap z = \{q_2\}$ is not a region, v and z are incompatible elements of $\mathcal{R}(A_0)$.

In the following, besides illustrating the synthesis procedure, we show by means of examples that in general there is no isomorphism between L and $\mathcal{R}(\boldsymbol{\mathcal{A}}(L))$ as well as between A and $\boldsymbol{\mathcal{A}}(\mathcal{R}(A))$.

The core of the synthesis procedure is in interpreting the states of $\mathcal{S}(L)$ as the states of a transition system. A state of the transition system can be identified with the set of all regions containing it, and, by Proposition 3, it coincides with some state of a logic, as in Definition 8. In fact, the transition system associated to a logic L is constructed by taking $\mathcal{S}(L)$ as the set of states, and symmetric differences between states as events. Formally we have the following.

$$E(L) = \{\langle s_1 \setminus s_2, s_2 \setminus s_1\rangle \mid s_1, s_2 \in \mathcal{S}(L), s_1 \neq s_2\}. \tag{1}$$

The set of transitions is now naturally defined as the set of all pairs of distinct states, each labelled by the corresponding ordered symmetric difference. In the following, $[s_1, s_2]$ will denote $\langle s_1 \setminus s_2, s_2 \setminus s_1\rangle$.

$$T(L) = \{(s_1, [s_1, s_2], s_2) \mid s_1, s_2 \in \mathcal{S}(L), s_1 \neq s_2\} \tag{2}$$

Of course, the same label can have several occurrences. We can now define the transition system

$$\boldsymbol{\mathcal{A}}(L) = (\mathcal{S}(L), E(L), T(L)). \tag{3}$$

The transition system $\boldsymbol{\mathcal{A}}(L)$ includes a transition for each ordered pair of states; hence we call it *saturated* of transitions. $\boldsymbol{\mathcal{A}}(L)$ is a CETS, as shown in [4].

Example 8. The saturated CETS $\boldsymbol{\mathcal{A}}(L)$ synthesised from the logic L in Fig. 2 is given in Fig. 4.

At this point, two natural questions arise about the two opposite transformations and the possibility that they are inverse of each other:

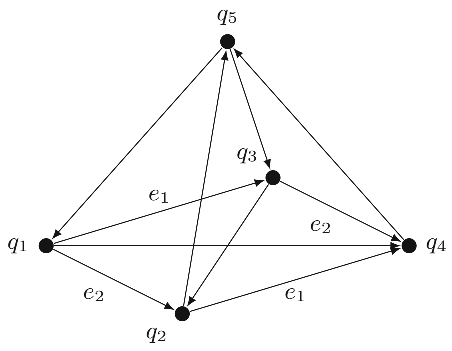


Fig. 4. Saturated Transition System synthesised from the logic in Fig. 2. For clarity, not all transitions have been depicted: for each arc in the figure, there is another one in the opposite direction. Also, only repeated labels have been indicated.

1. Given a rich and regular logic L, build the CETS $\boldsymbol{\mathcal{A}}(L)$ and consider its regional logic $\mathcal{R}(\boldsymbol{\mathcal{A}}(L))$. Is $\mathcal{R}(\boldsymbol{\mathcal{A}}(L))$ isomorphic to L?

2. Given a CETS A_0, construct the CETS associated to its regional logic $\boldsymbol{\mathcal{A}}(\mathcal{R}(A_0))$. Is $\boldsymbol{\mathcal{A}}(\mathcal{R}(A_0))$ isomorphic to A_0?

In the general case, L embeds into $\mathcal{R}(\boldsymbol{\mathcal{A}}(L))$. In fact, for each $x \in L$, $\mathcal{S}_x$ is a region of $\boldsymbol{\mathcal{A}}(L)$ and the embedding is given by $\phi(x) = \mathcal{S}_x \subseteq \mathcal{S}(L)$.

The fact that $\phi(\,.\,)$ is an embedding of logics will be formally proved in the next section in Proposition 4. However, $\phi(\,.\,)$ is not always an isomorphism, and this can be clarified by the following example.

Example 9. Consider the set $\Omega = \{1, 2, \ldots, 6\}$ and define Δ as the collection of the $X \in \mathcal{P}(\Omega)$ such that $|X|$ is an even number. Then $L = (\Delta, \subseteq, \emptyset, \Omega, (.)')$ is a regular concrete logic in which, for $x, y \in \Delta$, $x \ \$\ y \Leftrightarrow x \cap y \in \Delta$. All the states of L are represented by the sets $\delta_i = \{x \in \Delta \,|\, i \in x\}$ for $i = \{1, 2, \ldots, 6\}$. The CETS associated to L, $\boldsymbol{\mathcal{A}}(L)$ has then six states and a set of transitions, computed as in Eq. (2) above, whose labels are all distinct. This means that the regions of the CETS $\boldsymbol{\mathcal{A}}(L)$, defined as in Eq. (3) above, are isomorphic to the power set $\mathcal{P}(\Omega)$ that strictly contains Δ.

Viceversa, by assuming as given a CETS $A = (Q, E, T)$, the general case shows that $\mathcal{S}(\mathcal{R}(A))$, and then also $\boldsymbol{\mathcal{A}}(\mathcal{R}(A))$, can contain states which do not correspond to any state in Q. This means that, in general, A is not isomorphic to $\boldsymbol{\mathcal{A}}(\mathcal{R}(A))$, as for example in the following.

Example 10. Consider the CETS $A = (Q, E, T)$ shown in Fig. 5. Its regions are the trivial ones, $\emptyset$ and Q, plus $x = \{1, 2, 5\}$, $y = \{1, 2, 6\}$, $z = \{1, 3, 5\}$, $w = \{1, 3, 6\}$ and the respective complements $\{3, 4, 6\} = (\{1, 2, 5\})'$, $\{3, 4, 5\} = (\{1, 2, 6\})'$, $\{2, 4, 6\} = (\{1, 3, 5\})'$ and $\{2, 4, 5\} = (\{1, 3, 6\})'$. The corresponding concrete logic is represented in Fig. 5 and its states are formed by choosing exactly one element from each complementary pair of disjoint non-trivial regions. Hence, there are sixteen states. By applying the *synthesis procedure* above to the logic $\mathcal{R}(A) = (\mathcal{R}(A), \subseteq, \emptyset, Q, (.)')$, we find that six out of the sixteen states in $\mathcal{S}(\mathcal{R}(A))$ correspond to the original states of A.

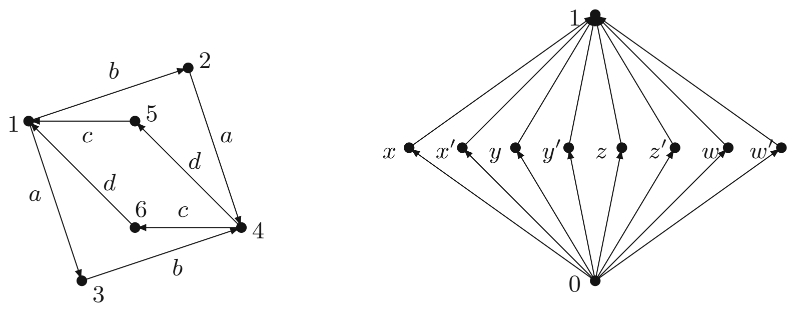


Fig. 5. A CETS and its regional logic.

Concerning the relation between regions and events, we note that pre- and post-regions of an event $e \in E$ can be retrieved in $\mathcal{R}(A)$ as set differences between states of the logic. Remember that $\mathcal{R}_{q_1}$ is the set of regions containing q_1 and $\mathcal{R}_{q_2}$ is the set of regions containing q_2. If (q_1, e, q_2) is a transition in A, then $\mathcal{R}_{q_1} \setminus \mathcal{R}_{q_2}$ gives the set of regions from which e exits, namely the set of pre-regions of e; the difference in the other direction gives the set of post-regions. By definition of region, these differences are independent of the individual occurrence of e in A: let $(q_1, e, q_2), (q_3, e, q_4) \in T$. Then $\mathcal{R}_{q_1} \setminus \mathcal{R}_{q_2} = \mathcal{R}_{q_3} \setminus \mathcal{R}_{q_4} = {}^\bullet e$ and $\mathcal{R}_{q_2} \setminus \mathcal{R}_{q_1} = \mathcal{R}_{q_4} \setminus \mathcal{R}_{q_3} = e^\bullet$. Hence, these set differences allow us to identify different occurrences as corresponding to the same event, in other words, that two transitions carry the same label.

Example 11. For the CETS A in Fig. 5, the case can be exemplified by considering the two events labelled a, leading from state 1 to state 3 and from state 2 to state 4, respectively. In this case, $\mathcal{R}_1$ is the set composed by $\{1,2,5\}, \{1,2,6\}$, $\{1,3,5\}, \{1,3,6\}$ and $\mathcal{R}_3$ is composed by $\{3,4,6\}, \{3,4,5\}, \{1,3,5\}, \{1,3,6\}$. By computing the set differences as above, we find that: $\mathcal{R}_1 \setminus \mathcal{R}_3 = \mathcal{R}_2 \setminus \mathcal{R}_4$, the set of regions from which a exits. The symmetric case leads to $\mathcal{R}_3 \setminus \mathcal{R}_1 = \mathcal{R}_4 \setminus \mathcal{R}_2$. With reference to the logic $\mathcal{R}(A)$ of the CETS A, both represented in Fig. 5, we can compute, in terms of symmetric differences, the transitions between any couple of the sixteen states of the logic $\mathcal{R}(A)$ as in Eq. (2) above. We find, among many others, the transitions corresponding to the identical labels of A. For example, by considering the states in S_x and in S_y, we have that $S_x \setminus S_y = S_{y'} \setminus S_{x'}$.

4 Towards the Characterization of Stable Regional Logics

In the previous section, we have considered two cases: the first one deals with the construction of a logic $\mathcal{R}(A)$ starting from a CETS A and the second one deals with the construction, or *synthesis*, of a CETS $\boldsymbol{\mathcal{A}}(L)$ starting from a regular and rich logic L. In both cases, by applying again the process and computing $\boldsymbol{\mathcal{A}}(\mathcal{R}(A))$ and $\mathcal{R}(\boldsymbol{\mathcal{A}}(L))$ we obtain two embeddings. The original CETS A embeds in terms of both states and transitions into the CETS resulting from the *synthesis procedure* applied to $\mathcal{R}(A)$, and the original logic L embeds into the concrete logic formed by taking Ω as the states of $\boldsymbol{\mathcal{A}}(L)$ and Δ as the set of regions of $\boldsymbol{\mathcal{A}}(L)$. We say that the logic L is *stable* if this embedding is an isomorphism of logics.

Definition 12. *A quantum logic L is* stable *if there is an isomorphism of logics from L onto $\mathcal{R}(\boldsymbol{\mathcal{A}}(L))$.*

Our long-term aim is to characterize the class of stable regional logics. In what follows, we present some new results towards such a goal. First we discuss two necessary conditions for a logic to be regional, and then we prove that one implies the other. Afterwards, in Subsect. 4.2, we prove that there is an embedding of logics from L to $\mathcal{R}(\boldsymbol{\mathcal{A}}(L))$, and that, under the same condition, this embedding is strong in the sense that it preserves also incompatibility.

4.1 Regional Logics Are ETI and TIP

In [6] and [5] it is shown that regional logics satisfy the properties called TIP and ETI, respectively. In the following, after recalling the definitions, we show that any ETI logic is also a TIP logic.

The property TIP results from the translation in the abstract setting of quantum logic of a property of regions of transition systems, which we call *triple intersection property* and which is expressed by the following lemma, the proof of which can be found in [6].

Lemma 1. *Let A be a CETS, and let $a, b, c \in \mathcal{R}(A)$ be such that: $a \cap b = b \cap c = c \cap a$, then $z = a \cap b \cap c \in \mathcal{R}(A)$.*

The triple intersection property (TIP) for a logic L is then obtained by considering for any element in L the set of two-valued states selecting that element and from the fact that this set identifies a region of the CETS $\boldsymbol{\mathcal{A}}(L)$ synthesised from L.

Definition 13. *A logic L is* TIP *if*

$$\forall a, b, c \in L: \ \mathcal{S}_a \cap \mathcal{S}_b = \mathcal{S}_b \cap \mathcal{S}_c = \mathcal{S}_c \cap \mathcal{S}_a \quad \Rightarrow \quad \exists z \in L: \ \mathcal{S}_z = \mathcal{S}_a \cap \mathcal{S}_b \cap \mathcal{S}_c$$

As well as TIP, also ETI is inspired by a property of regions. If q_1, q_2, q_3, and q_4 are distinct states, so that (q_1, e, q_2), and (q_3, e, q_4) are two different transitions in a CETS $A = (Q, E, T)$, then $\mathcal{R}(A)$ must contain two incompatible regions, one containing q_1 and q_2 but not q_3, the other containing q_1 and q_3 but not q_2. Hence, events with multiple occurrences and pairs of incompatible regions are related. In this sense, we say that the events of an abstract logic L *testify incompatibility*; L is said to be ETI if, for any pair of incompatible regions, the set $E(L)$ contains an element witnessing this incompatibility.

Definition 14. *A logic L is* ETI *if* $\quad \forall a, b \in L : \quad a \not\$ b \;\Rightarrow$

$$\exists s_1 \in \mathcal{S}_a \cap \mathcal{S}_b, s_a \in \mathcal{S}_a \cap \mathcal{S}_{b'}, s_b \in \mathcal{S}_b \cap \mathcal{S}_{a'}, s_0 \in \mathcal{S}_{a'} \cap \mathcal{S}_{b'} : \;\; s_a \setminus s_1 = s_0 \setminus s_b$$

Example 12. An example of a concrete logic which is neither TIP nor ETI is the logic of the subsets of even cardinality of $\{1, 2, 3, 4, 5, 6\}$, already seen in Example 9. This logic is regular and rich, but not regional as discussed in [4]. In order to see that L is not TIP, let us consider the elements in L: $\{1, 2\}, \{1, 3\}, \{1, 4\}$, then $\mathcal{S}_{\{1,2\}} \cap \mathcal{S}_{\{1,3\}} = \mathcal{S}_{\{1,3\}} \cap \mathcal{S}_{\{1,4\}} = \mathcal{S}_{\{1,4\}} \cap \mathcal{S}_{\{1,2\}} = \{\delta_1\}$, where δ_1 is the two-valued state of L selecting all the elements containing 1. It is then immediate to see that there is no z in L such that $\mathcal{S}_z = \{\delta_1\}$. L is not ETI since the symmetric differences among the six states of the associated transition systems are all different, and then it is not possible for a pair of incompatible elements of L to find pairs of equal symmetric differences.

Any regional logic is TIP, as shown in [6], and ETI, as shown in [5]. Although we do not know yet if the two properties, ETI and TIP, coincide, we can prove that ETI implies TIP.

Theorem 3. *Let L be a* ETI *logic. Then L is* TIP.

Proof. By contradiction, let L be ETI and not TIP. Not TIP means: $\exists a, b, c \in L$: $\mathcal{S}_a \cap \mathcal{S}_b = \mathcal{S}_b \cap \mathcal{S}_c = \mathcal{S}_c \cap \mathcal{S}_a \neq \emptyset$ and $\forall z \in L : \; \mathcal{S}_z \neq \mathcal{S}_a \cap \mathcal{S}_b \cap \mathcal{S}_c$. We have two cases.

First case: $a \$ b$. Then since $\mathcal{S}_a \cap \mathcal{S}_b \neq \emptyset$, $a \not\perp b$ and then there exist three mutually orthogonal elements $\hat{a}$, $\hat{b}$ and x in L such that $a = \hat{a} \vee x$ and $b = \hat{b} \vee x$. This implies $\mathcal{S}_a$ to be the disjoint union of $\mathcal{S}_{\hat{a}}$ and $\mathcal{S}_x$, and $\mathcal{S}_b$ to be the disjoint union of $\mathcal{S}_{\hat{b}}$ and $\mathcal{S}_x$, then we get the contradiction: $\exists x : \mathcal{S}_x = \mathcal{S}_a \cap \mathcal{S}_b = \mathcal{S}_c \cap \mathcal{S}_a$.

Second case: $a \not\$ b$. Then, since L is ETI, there are four states: $s_1 \in \mathcal{S}_a \cap \mathcal{S}_b$, $s_a \in \mathcal{S}_a \cap \mathcal{S}_{b'}, s_b \in \mathcal{S}_b \cap \mathcal{S}_{a'}$, $s_0 \in \mathcal{S}_{a'} \cap \mathcal{S}_{b'}$ such that: $s_a \setminus s_1 = s_0 \setminus s_b$. Then $s_1 \in \mathcal{S}_a \cap \mathcal{S}_b$ implies $s_1 \in \mathcal{S}_c$. Since $\mathcal{S}_c$ is a region in $\mathcal{R}(\boldsymbol{\mathcal{A}}(L))$, then either $s_a \in \mathcal{S}_c$ and $s_b \notin \mathcal{S}_c$, or $s_b \in \mathcal{S}_c$ and $s_a \notin \mathcal{S}_c$. In any case this contradicts the hypothesis $\mathcal{S}_a \cap \mathcal{S}_b = \mathcal{S}_b \cap \mathcal{S}_c = \mathcal{S}_c \cap \mathcal{S}_a$. □

4.2 Strong Embedding of L into $\mathcal{R}(\boldsymbol{\mathcal{A}}(L))$

At this point, we stress the fact that given a regular, and rich logic L, and its synthesised CETS $\boldsymbol{\mathcal{A}}(L)$, the injection defined by $\phi(x) = \mathcal{S}_x$ is a morphism of logics. Furthermore, we show that $\phi : L \to \mathcal{R}(\boldsymbol{\mathcal{A}}(L))$ is an embedding of logics. To check that ϕ is a morphism, we verify the three properties in Definition 7. First note that $\mathcal{S}_0 = \emptyset$ is the bottom element in $\mathcal{R}(\boldsymbol{\mathcal{A}}(L))$. Second, since any $x \in L$ is orthogonal to its complement x', from point 2. in Definition 8, it stands that $\mathcal{S}_x \cap \mathcal{S}_{x'} = \emptyset$, and furthermore, $\forall x \in L : \mathcal{S}_x \cup \mathcal{S}_{x'} = \mathcal{S}(L)$, so $\mathcal{S}_{x'} = \mathcal{S}(L) \setminus \mathcal{S}_x$. The third point is a direct consequence of Proposition 1.

On the other hand, put $\mathcal{S}_x = \mathcal{S}_y$, then L being rich, it holds from Definition 10 that $x \leq y$ and $y \leq x$, so ϕ is injective. We even see, thanks to the following proposition, that $\phi|_L$ is an isomorphism, and so ϕ is an embedding. To prove so, it is sufficient to show that $\phi^{-1}|_{\phi(L)}$ is a morphism of logics.

Proposition 4. *Let L be a rich and regular logic, and $\phi : L \to \mathcal{R}(\boldsymbol{\mathcal{A}}(L))$ be defined by $\phi(x) = \mathcal{S}_x$ for all $x \in L$. Then $\psi \equiv \phi^{-1}|_{\phi(L)}$ is a logic morphism.*

Proof. Clearly, $\forall x \in L : \psi(\mathcal{S}_x) = x$. Now, since L is rich, the only element whose associated set of states is the bottom element, hence $\psi(\emptyset) = 0$. Also, $\psi(\mathcal{S}(L) \setminus \mathcal{S}_x) = \psi(\mathcal{S}_{x'}) = x'$, so it preserves orthocomplements. Finally, consider two disjoint $\mathcal{S}_x, \mathcal{S}_y$. Then $\mathcal{S}_x \subseteq \mathcal{S}(L) \setminus \mathcal{S}_y = \mathcal{S}_{y'}$, and since L is rich, $x \perp y$. From point 2. in Definition 8 it stands that $\{s \in \mathcal{S}(L) \mid s(x \vee y) = 1\} = \{s \in \mathcal{S}(L) \mid s(x) = 1\} \cup \{s \in \mathcal{S}(L) \mid s(y) = 1\} = \mathcal{S}_x \cup \mathcal{S}_y$. Hence $\psi(\mathcal{S}_x \cup \mathcal{S}_y) = x \vee y$. □

Now, ϕ being an embedding means that L and $\mathcal{R}(\boldsymbol{\mathcal{A}}(L))$ would be isomorphic if ϕ was surjective. We have also seen in the previous section that L being ETI is a necessary condition for that. We shall now see that if L is ETI, the embedding verifies a stronger property, required (but not sufficient) for ϕ to be an isomorphism.

We remind the reader that logic morphisms preserve order, orthogonality and compatibility. Since ϕ is an embedding it shall also reflect these relations. However, in general this is only true when considering them restricted to the image $\phi(L)$. Indeed, if ϕ is an embedding then L is isomorphic to $\phi(L)$, but the lack of surjectivity might, in the general case, allow the images of two incompatible elements to be compatible. We shall explain this notion through an example.

Example 13. Consider the logic $L = \{0, u, u', v, v', 1\}$, and the Boolean logic B whose atoms are $\{a_1, a_2, a_3, a_4\}$ (Fig. 6). Then the mapping given by $\phi(u) = a_1 \vee a_2$, $\phi(u') = a_3 \vee a_4$, $\phi(v) = a_1 \vee a_3$, $\phi(v') = a_2 \vee a_4$ is indeed an embedding. Since $(a_1 \vee a_2)' = a_3 \vee a_4$ and $(a_1 \vee a_3)' = a_2 \vee a_4$, the sublogic $L_1 = \{0, a_1 \vee a_2, a_3 \vee a_4, a_1 \vee a_3, a_2 \vee a_4, 1\}$ of B is isomorphic to L. However, when considered in the whole of B, we see that a_1 is both $a_1 \leq a_1 \vee a_2$ and $a_1 \leq a_1 \vee a_3$, with $\{a_1, a_2, a_3\}$ mutually orthogonal in B. Thus $\phi(u) \$ \phi(v)$, whereas $u \not\$ v$. This does not prevent ϕ from being an embedding because, in fact $a_1, a_2, a_3 \notin L_1$.

This example should justify the following definition (see [11]).

Definition 15. *Let $\phi : L_1 \to L_2$ be an embedding between logics. Then ϕ is said to be a* strong embedding *if*

$$\forall a, b \in L_1 : a \$ b \Leftrightarrow \phi(a) \$ \phi(b)$$

For instance, a logic with incompatible elements cannot embed strongly into a Boolean logic.

We shall now prove that a logic L being ETI is a sufficient condition for $\phi : L \to \mathcal{R}(\boldsymbol{\mathcal{A}}(L))$ to be a strong embedding.

Theorem 4. *Let L be a rich and regular logic. If L is* ETI*, then the embedding $\phi : L \to \mathcal{R}(\boldsymbol{\mathcal{A}}(L))$ defined as $\phi(x) = \mathcal{S}_x$ is strong.*

Proof. We have already shown that ϕ preserves compatibility, it will therefore be sufficient to prove that it also preserves incompatibility. So let $a, b \in L$ verify $a \not\$ b$. Since L is ETI, $\exists s_1 \in \mathcal{S}_a \cap \mathcal{S}_b, s_a \in \mathcal{S}_a \cap \mathcal{S}_{b'}, s_b \in \mathcal{S}_b \cap \mathcal{S}_{a'}, s_0 \in \mathcal{S}_{a'} \cap \mathcal{S}_{b'}$:

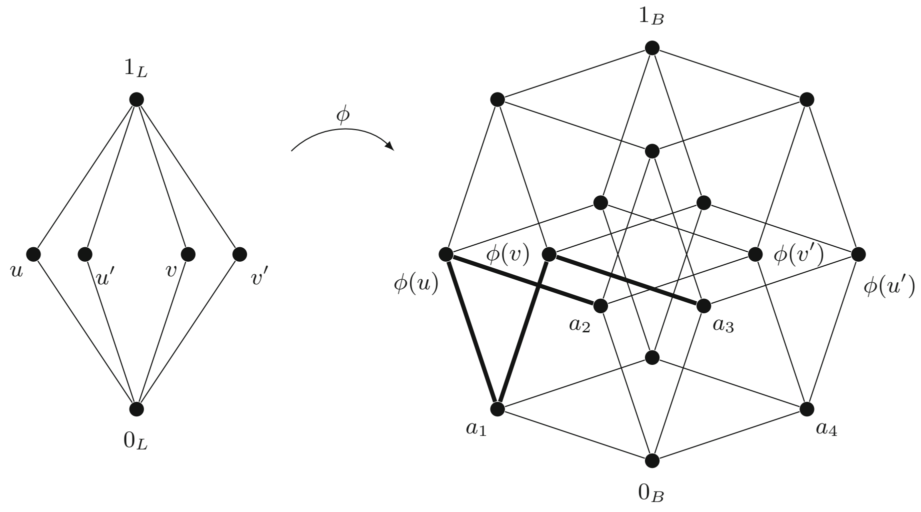


Fig. 6. An example of embedding which is not strong.

$s_a \setminus s_1 = s_0 \setminus s_b$. Then $e = [s_1, s_a] \in E(L)$ will be a label in the saturated transition system $\boldsymbol{\mathcal{A}}(L)$, and the transitions $(s_1, e, s_a), (s_b, e, s_0) \in T(L)$ will prevent $\mathcal{S}_a \cap \mathcal{S}_b$ from being a region. Indeed, (s_1, e, s_a) crosses the border of $\mathcal{S}_a \cap \mathcal{S}_b$, whereas (s_b, e, s_0) does not. Since $\mathcal{R}(\boldsymbol{\mathcal{A}}(L))$ is a concrete logic, we have that $\mathcal{S}_a = \phi(a) \not\subseteq \phi(b) = \mathcal{S}_b$. □

This result implies, in particular, that if new regions are produced by the synthesis procedure, these cannot be contained in the image by ϕ of an atom. Orthocomplementation implies therefore that they cannot contain the images of coatoms (maximal elements except for the top). Thus, the possible lack of surjectivity of ϕ is narrowed down.

5 Classes of Stable Regional Logics

In this section we look at a few subclasses of concrete logics, and show that they are stable.

To start with the simplest example, let L be a finite Boolean logic with k atoms. Then L is a regular rich logic, isomorphic to the power set of $\{1, \cdots, k\}$, in which singletons correspond to atoms. L has exactly k states, each corresponding to $\Uparrow\{x\}$, where x is an atom of L.

This implies that all the ordered symmetric differences between states differ in at least one atom, so that each transition in $\boldsymbol{\mathcal{A}}(L)$ carries a unique label, and all subsets of states are regions. Hence, L and $\mathcal{R}(\boldsymbol{\mathcal{A}}(L))$ are isomorphic, and L is stable.

In the next cases, we will use the notion of *restriction* of a transition system to a subset of events. We will say that $A_1 = (S, E_1, T_1)$ is a restriction of $A =$

(S, E, T) if they have the same set of states, $E_1 \subseteq E$, and T_1 is obtained by removing from T all transitions labelled by events in $E \setminus E_1$. From the definition of region, it follows that $\mathcal{R}(A) \subseteq \mathcal{R}(A_1)$.

The next case we consider is that of a logic obtained as the so-called $\{0,1\}$-pasting of two (or more) logics. In plain words, the $\{0, 1\}$-pasting of L_1 and L_2 is the disjoint union of L_1 and L_2, but for identification of 0_1 with 0_2, and 1_1 with 1_2 (see Fig. 7).

Definition 16. *Let L_1 and L_2 be two disjoint logics with $0_i, 1_i$ for $i = 1, 2$ the respective least and greatest elements. Define on $L_1 \cup L_2$ the equivalence relation*

$$\sim := \{(a, a) \mid a \in L_1 \cup L_2\} \cup \{(0_1, 0_2), (1_1, 1_2)\}.$$

Then the $\{0\text{-}1\}$-pasting *of L_1 and L_2, denoted by $L_1 \,||\, L_2$ is given by the set $(L_1 \cup L_2)/\!\sim$, together with the partial order obtained as the union of the partial orders of L_1 and L_2, up to $\sim$.*

The $\{0\text{-}1\}$-pasting of L_1 and L_2 is again a logic ([11], Proposition 1.2.6). Without loss of generality, in what follows we will simplify the notation concerning the $\{0\text{-}1\}$-pasting of L_1 and L_2 by indicating by the same element 0 and, respectively, 1 the bottom and top elements in *both* L_1 and L_2.

Remark 1. As a useful consequence, the set of states $\mathcal{S}(L_1 \,||\, L_2)$ of $L_1 \,||\, L_2$, consists in sets obtained by taking the union $s_1 \cup s_2$ for any pair of states $s_1 \in \mathcal{S}(L_1)$ and $s_2 \in \mathcal{S}(L_2)$.

The $\{0, 1\}$-pasting of logics is related to a construction on transition systems. Given two transition systems, A_1 and A_2, with disjoint sets of events, we can build a new one by putting them side-by-side and letting them work in parallel. States of this new transition system are pairs of "local" states of the two components. We will call the result the *parallel product* of A_1 and A_2, and denote it by $A_1 \,||\, A_2$ (see Fig. 8).

Definition 17. *Let $A_i = (Q_i, E_i, T_i)$ be a CETS for $i = 1, 2$, with $E_1 \cap E_2 = \emptyset$. Define*

$$A_1 \,||\, A_2 = (Q_1 \times Q_2, E_1 \cup E_2, T)$$

where

$$\begin{aligned} T = &\{((q_1, q_2), e, (q_1', q_2)) \mid (q_1, e, q_1') \in T_1, q_2 \in Q_2\} \quad \cup \\ &\{((q_1, q_2), e, (q_1, q_2')) \mid (q_2, e, q_2') \in T_2, q_1 \in Q_1\} \end{aligned}$$

The next lemma shows that the parallel product of transition systems and the $\{0, 1\}$-pasting of logics are strictly related.

Proposition 5. *Let $A_i = (Q_i, E_i, T_i)$ be a CETS for $i = 1, 2$, with $E_1 \cap E_2 = \emptyset$. Then $\mathcal{R}(A_1 \,||\, A_2)$ and $\mathcal{R}(A_1) \,||\, \mathcal{R}(A_2)$ are isomorphic logics.*

Proof. By construction, if there is a transition $((q_1, q_2), e, (q'_1, q_2))$ in T, then, for each q'_2 in Q_2, T contains a transition $((q_1, q'_2), e, (q'_1, q'_2))$. Hence, for each region r of A_1, the set $r \times Q_2$ is a region of $A_1 \,||\, A_2$, and, for each region r of A_2, the same holds for the set $Q_1 \times r$. For any region r of $A_1 \,||\, A_2$, the projection of its states on the first component must be a region of A_1 (and symmetrically for the projection on the second component), because in any transition only one of the two components of a state will change; hence, the full set of non-trivial regions of $A_1 \,||\, A_2$ is given by $\{r \times Q_2 \mid r \in \mathcal{R}(A_1)\} \cup \{Q_1 \times r \mid r \in \mathcal{R}(A_2)\}$. $\square$

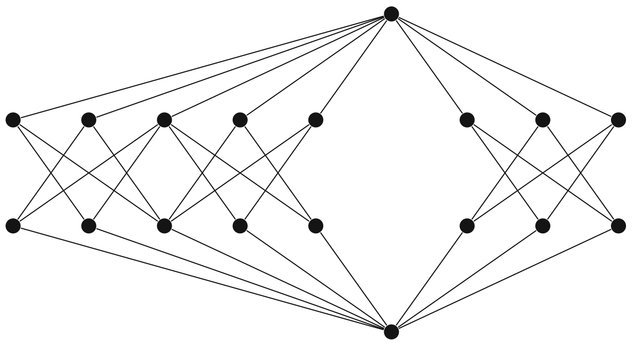


Fig. 7. $\{0, 1\}$-pasting of the logic in Fig. 2, and a Boolean algebra with 3 atoms.

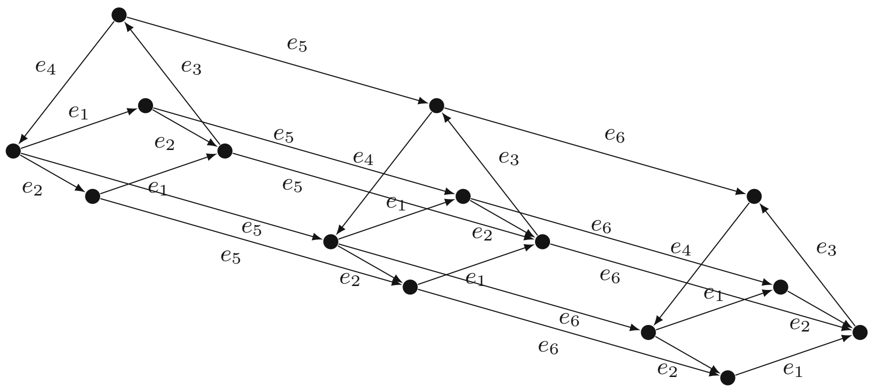


Fig. 8. A CETS such that its poset of regions is the one depicted in Fig. 7. It is the product of the CETS of Fig. 1 and a sequence of three states.

Proposition 6. *Let L_1 and L_2 be stable regional logics. Then $L = L_1 \,||\, L_2$ is a stable regional logic.*

Proof. By Theorem 2, we can identify L with the isomorphic concrete logic where each x in L is represented by $\mathcal{S}_x$. Then, as discussed in Sect. 3, $L \subseteq \mathcal{R}(\boldsymbol{\mathcal{A}}(L))$.

Remark 1 implies that

$$\mathcal{S}(L) = \{s_1 \cup s_2 \mid s_1 \in \mathcal{S}(L_1), s_2 \in \mathcal{S}(L_2)\}.$$

Since $s_1 \cap s_2 = \{1\}$ for each choice of s_1 and s_2 above, we can represent $\mathcal{S}(L)$ as $\mathcal{S}(L_1) \times \mathcal{S}(L_2)$.

We will now define a transition system on $\mathcal{S}(L)$, by taking a subset of the events and transitions of the saturated transition system. The idea is to choose only the "local" transitions, namely transitions that change only one component of a state. Define

$$\begin{aligned} E_M = &\{[(s_1, s_2), (s_1', s_2)] \mid s_1, s_1' \in \mathcal{S}(L_1), s_2 \in \mathcal{S}(L_2)\} \quad \cup \\ &\{[(s_1, s_2), (s_1, s_2')] \mid s_1 \in \mathcal{S}(L_1), s_2, s_2' \in \mathcal{S}(L_2)\} \end{aligned}$$

We must now take all the transitions corresponding to labels in E.

$$\begin{aligned} T_M = &\{((s_1, s_2), [(s_1, s_2), (s_1', s_2)], (s_1', s_2)) \mid s_1, s_1' \in \mathcal{S}(L_1), s_2 \in \mathcal{S}(L_2)\} \quad \cup \\ &\{((s_1, s_2), [(s_1, s_2), (s_1, s_2')], (s_1, s_2')) \mid s_1 \in \mathcal{S}(L_1), s_2, s_2' \in \mathcal{S}(L_2)\}. \end{aligned}$$

The transition system $\boldsymbol{\mathcal{A}}_M(L) = (\mathcal{S}(L), E_M, T_M)$ is a restriction of $\boldsymbol{\mathcal{A}}(L)$, as such, its set of regions is a superset of $R(\boldsymbol{\mathcal{A}}(L))$.

On the other hand, $\boldsymbol{\mathcal{A}}_M$ is isomorphic to $\boldsymbol{\mathcal{A}}(L_1) \,||\, \boldsymbol{\mathcal{A}}(L_2)$. We can then apply Proposition 5, and the hypothesis of stability of L_1 and L_2 to derive

$$\mathcal{R}(\boldsymbol{\mathcal{A}}_M(L)) = \mathcal{R}(\boldsymbol{\mathcal{A}}(L_1) \,||\, \boldsymbol{\mathcal{A}}(L_2)) = \mathcal{R}(\boldsymbol{\mathcal{A}}(L_1)) \,||\, \mathcal{R}(\boldsymbol{\mathcal{A}}(L_2)) = L_1 \,||\, L_2 = L$$

and

$$L \subseteq \mathcal{R}(\boldsymbol{\mathcal{A}}(L)) \subseteq \mathcal{R}(\boldsymbol{\mathcal{A}}_M(L)) = L$$

so that $L = \mathcal{R}(\boldsymbol{\mathcal{A}}(L))$, and L is stable. □

The construction and the argument above can be generalised to the case of the $\{0, 1\}$-pasting of K logics, noting that $L_1 \,||\, L_2 \,||\, L_3$ is isomorphic to $L_1 \,||\, (L_2 \,||\, L_3)$.

Let us now suppose that L is a rich, regular quantum logic, which is the union of Boolean algebras such that their pairwise intersections are all the same Boolean algebra. More formally, $L = \bigcup_{i \leq n} B_i$, where $\{B_i\}_{i \leq n}$ is a finite family of finite Boolean algebras, and there is a Boolean algebra B such that $\forall i \neq j : B_i \cap B_j = B$. B corresponds to what is called the *centre* of L in [11], it is a sublogic of any B_i. An example of such a logic with $n = 2$ and $B = \{0, x, x', 1\}$ has been given in Fig. 2 and discussed in Examples 1, 2, and 3.

We shall prove that such logics are stable, for which we require that the center B contains at least an atom of L. This is, however, always the case, as shown in the following lemma.

Lemma 2. *Let $\{B\} \cup \{B_i\}_{i \leq n}$ be a finite family of finite Boolean algebras, and L be a logic such that $L = \bigcup_{i \leq n} B_i$, and $\forall i \neq j : B_i \cap B_j = B$. Then there is at least one atom of B which is an atom of L.*

Proof. Note that by construction, each B_i is a maximal Boolean subalgebra of L. Hence, for any $x_i \in B_i \setminus B$, and $x_j \in B_j \setminus B$: $i \neq j \Rightarrow x_i \not\$ x_j$. We proceed by reductio ad absurdum, so suppose that no atom of B is an atom of L. Let x be an atom of B, then neither x nor x' are atoms of L. Then $\exists\, x_1 \in L \setminus B : x_1 < x$, (hence $x' < x_1'$). Suppose w.l.g. that $x_1 \in B_1$. Since $x_1 \notin B$ we have that $\forall j \neq 1 : x_1 \notin B_j$, and so $x_1' \notin B_j$. It holds, in particular for $j = 2$. On the other hand, $B \subsetneq B_2$ and $x \notin \{0_L, 1_L\}$ imply that $\exists x_2 \in B_2 \setminus B : x \$ x_2$, hence $x_2 \notin B_1$. We distinguish two cases. Either $x \perp x_2$, or $\exists y_2 \in B_2 : y_2 = x \wedge x_2$ with $y_2 \neq 0_{B_2} = 0_L$, and since x is an atom of B: $y_2 \notin B_1$. From $x \perp x_2$, it follows that $x_2 \leq x' \leq x_1'$, so in particular $x_1' \$ x_2$, hence $x_1 \$ x_2$, which is in contradiction with $(x_1 \notin B_2) \wedge (x_2 \notin B_1)$. If $y_2 = x \wedge x_2$, we have that $x' < x_1'$ and $x' < y_2'$ with $x_1' \in B_1 \setminus B$ and $y_2' \in B_2 \setminus B$. Since, by hypothesis, x' is not an atom of L, there must be an atom y of L such that $y < x'$, and $y \notin B$. Now, there must be some i such that $y \in B_i \setminus B$. If $i = 1$ then $y \notin B_2$, so $y \not\$ y_2'$, but $y < y_2'$, which is a contradiction. If $i \neq 1$ then $y \notin B_1$, and inconsistency follows from $y < x_1'$ and $y \not\$ x_1'$. □

We can now prove the following result.

Proposition 7. *Let $\{B\} \cup \{B_i\}_{i \leq n}$ be a finite family of finite Boolean algebras, and L be a logic such that $L = \bigcup_{i \leq n} B_i$, and $\forall i \neq j : B_i \cap B_j = B$. Then L is stable.*

Proof. Let $\mathrm{AT}(L)$ denote the atoms of L. Lemma 2 shows, that some atom of B is in $\mathrm{AT}(L)$. Since $\forall i \leq n : B \subseteq B_i$, any pair of atoms $x \in B \cap \mathrm{AT}(L)$, $y \in \mathrm{AT}(L)$ are orthogonal. Hence, a state containing an atom of L belonging to B will contain no other atom in L. This allows us to partition the states of L into two classes. On one hand, the class S_B of states that contain exactly one atom $x \in \mathrm{AT}(L) \cap B$. Since the family $\{B_i \setminus B\}$ is pairwise disjoint, the class S of remaining states will contain exactly one atom in each of the set differences $B_i \setminus B$.

Now, the carrier of $\boldsymbol{\mathcal{A}}(L)$, is the disjoint union of S and S_B: $\mathcal{S}(L) = S \cup S_B$. Denote the states in S by $\upharpoonleft\{x_i\}_{i \leq n}$ where each x_i is an atom of B_i, and the states in S_B by $\upharpoonleft\{x\}$ where x is an atom of both B and L.

Let $EM(L) := \{\langle \upharpoonleft\{x\} \setminus \upharpoonleft\{y\}, \upharpoonleft\{y\} \setminus \upharpoonleft\{x\} \rangle \mid \exists i \leq n : x, y \in B_i \setminus B \text{ are atoms}\}$ be the set of events associated to local transitions among states of S, and define $TM(L)$ as the set of all transitions in $\boldsymbol{\mathcal{A}}(L)$ with labels in $EM(L)$. Finally, define $\boldsymbol{\mathcal{A}}_M(L) = (\mathcal{S}(L), EM(L), TM(L))$. Clearly, $\boldsymbol{\mathcal{A}}_M(L)$ is a generalised transition system, which is a restriction of $\boldsymbol{\mathcal{A}}(L)$. In particular, every region of $\boldsymbol{\mathcal{A}}(L)$ is also a region of $\boldsymbol{\mathcal{A}}_M(L)$.

$\boldsymbol{\mathcal{A}}_M(L)$ is not connected because the states $\upharpoonleft\{x\} \in S_B$ are all isolated.

Each transition in $\boldsymbol{\mathcal{A}}(L)$ starting from, or leading to, any $\upharpoonleft\{x\}$ carries a unique label, which has no other occurrence. Hence, the singleton formed by this state is a region in $\boldsymbol{\mathcal{A}}(L)$, as it is in $\boldsymbol{\mathcal{A}}_M(L)$. Furthermore, such singleton is disjoint to any other minimal region $\boldsymbol{\mathcal{A}}(L)$, and so the corresponding region is orthogonal to all other atoms of $\mathcal{R}(\boldsymbol{\mathcal{A}}(L))$. It therefore belongs to its center. In fact, all subsets of S_B are regions of $\boldsymbol{\mathcal{A}}(L)$, and as a power set, they form a Boolean algebra.

Now, each transition $t \in TM(L)$ corresponds to an ordered pair of states of S. For each such transition $t = (\upharpoonleft\{x_i\}_{i\leq n}, e, \upharpoonleft\{y_i\}_{i\leq n})$, there is an index $i \leq n$ such that $\forall j \neq i : x_j = y_j$. In this way, two transitions $t = (\upharpoonleft\{x_i\}_{i\leq n}, e, \upharpoonleft\{y_i\}_{i\leq n})$ and $\tilde{t} = (\upharpoonleft\{u_i\}_{i\leq n}, e, \upharpoonleft\{v_i\}_{i\leq n})$ carry the same label $e \in EM(L)$ if and only if there is an $i \leq n$, such that $x_i = u_i$, $y_i = v_i$, and $\forall j \neq i : x_j = y_j$ and $u_j = v_j$.

For each $j \leq n$, let A_j be the sequential transition system synthesized from the Boolean algebra generated by the atoms of $B_j \setminus B$. Then $EM(L)$ prevents any subset of S, which is not the disjoint union of sets of the form $\{\upharpoonleft\{x_i\}\} \times \Pi_{j\neq i} A_j$ from being a region.

In this way, we can define a natural bijection from the atoms of L to the atoms of $\mathcal{R}(\boldsymbol{\mathcal{A}}(L))$, which maps the $x \in B$ to the singletons $\{\upharpoonleft\{x\}\}$, and the $x_i \in B_i \setminus B$ to $\{\upharpoonleft\{x_i\}\} \times \Pi_{j\neq i} A_j$. Such a bijection preserves orthogonality and incompatibility, so that the logics generated by these atoms are isomorphic. $\square$

6 Conclusions

With the results presented in this paper we have done a further step towards the characterization of the rich and regular quantum logics, which result to be isomorphic to the orthomodular posets of the regions of the CETSs synthesised starting from the logics themself.

We are particularly interested in such a characterization because it is the basis for founding a logic of distributed systems. While the regions of a single sequential component constitute a Boolean logic, the presence of concurrency leads to a family of partially overlapping Boolean sublogics, each one corresponding to a sequential component.

Acknowledgements. This work is partially supported by MIUR.

References

1. Badouel, E., Bernardinello, L., Darondeau, P.: Petri Net Synthesis. TTCSAES. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-47967-4
2. Badouel, E., Darondeau, P.: Theory of regions. In: Reisig, W., Rozenberg, G. (eds.) ACPN 1996. LNCS, vol. 1491, pp. 529–586. Springer, Heidelberg (1998). https://doi.org/10.1007/3-540-65306-6_22
3. Bernardinello, L.: Synthesis of net systems. In: Ajmone Marsan, M. (ed.) ICATPN 1993. LNCS, vol. 691, pp. 89–105. Springer, Heidelberg (1993). https://doi.org/10.1007/3-540-56863-8_42
4. Bernardinello, L., Ferigato, C., Pomello, L.: An algebraic model of observable properties in distributed systems. Theoret. Comput. Sci. **290**(1), 637–668 (2003)
5. Bernardinello, L., Ferigato, C., Pomello, L., Puerto Aubel, A.: Synthesis of transition systems from concrete quantum logics. Fundam. Inform. **154**(1–4), 25–36 (2017)
6. Bernardinello, L., Pomello, L., Rombolà, S.: On orthomodular posets generated by transition systems. Electr. Notes Theor. Comput. Sci. **270**(1), 147–154 (2011)

7. Ehrenfeucht, A., Rozenberg, G.: Partial (set) 2-structures. Part I: basic notions and the representation problem. Acta Inf. **27**(4), 315–342 (1990)
8. Ehrenfeucht, A., Rozenberg, G.: Partial (set) 2-structures. Part II: state spaces of concurrent systems. Acta Inf. **27**(4), 343–368 (1990)
9. Nielsen, M., Rozenberg, G., Thiagarajan, P.S.: Elementary transition systems. Theoret. Comput. Sci. **96**(1), 3–33 (1992)
10. Petri, C.A.: Concepts of net theory. In: MFCS, pp. 137–146. Mathematical Institute of the Slovak Academy of Sciences (1973)
11. Pták, P., Pulmannová, S.: Orthomodular Structures as Quantum Logics. Kluwer Academic Publishers, Dordrecht (1991)

Decision Diagrams for Petri Nets: A Comparison of Variable Ordering Algorithms

Elvio Gilberto Amparore[1(✉)], Susanna Donatelli[1], Marco Beccuti[1], Giulio Garbi[1], and Andrew Miner[2]

[1] Dipartimento di Informatica, Università di Torino, Turin, Italy
{amparore,susi,beccuti}@di.unito.it
[2] Iowa State University, Ames, USA
asminer@iastate.edu

Abstract. The efficacy of decision diagram techniques for state space generation is known to be heavily dependent on the variable order. Ordering can be done a-priori (static) or during the state space generation (dynamic). We focus our attention on static ordering techniques. Many static decision diagram variable ordering techniques exist, but it is hard to choose which method to use, since only fragmented performance information is available. In the work reported in this paper we used the models of the Model Checking Contest 2017 edition to conduct an extensive comparison of 18 different algorithms, in order to better understand their efficacy. Comparison is based on the size of the decision diagram of the reachable state space, which is generated using the Saturation method provided by the Meddly library.

Keywords: Decision diagrams · Static variable ordering
Heuristic optimization · Saturation

1 Introduction

A binary decision diagram (BDD) [12] is a well-known data structure that has been extensively used in industrial hardware verification thanks to its ability of encoding complex boolean functions on very large domains. In the context of discrete event dynamic systems in general, and of Petri nets in particular, BDDs and various extensions (e.g. Multi-way Decision Diagrams, or MDDs) were proposed to efficiently generate and store the state space of complex systems. Indeed, symbolic state space generation techniques exploit Decision Diagrams (DDs) because they allow to encode and manipulate entire sets of states at once, instead of storing and exploring each state explicitly.

The intermediate and final sizes of DD representations are known to be strongly dependent on the choice of variable order: a good ordering can significantly change the memory consumption and the execution time needed to generate and encode the state space of a system. Unfortunately finding an optimal variable ordering is known to be NP-complete [11]. Therefore, efficient DD generation is usually reliant on various heuristics for the selection of (sub)optimal orderings. In this paper we will only consider *static* variable ordering, i.e. once the variable ordering l is selected, the MDD construction starts without the possibility of changing l. In the literature several papers were published to study the topic of variable ordering. An overview of these works can be found in [28], and more recently in [21]. In particular the latter work considers a new set of variable ordering algorithms, based on Bandwidth-reduction methods [29], and observes that they can be successfully applied to variable ordering. We also consider the work published in [20] (based on the ideas in [30]), which are state-of-the-art variable ordering methods specialized for Petri nets.

The motivation of this work was to understand how these different algorithms for variable orderings behave. Also, we wanted to investigate whether the availability of structural information of the Petri net model could make a difference. As far as we know there is no extensive comparison of these specific methods.

In particular we have addressed the following research objectives:

1. Build an environment (a *benchmark*) in which different algorithms can be checked on a vast number of models.
2. Investigate whether structural information like P-semiflows can be exploited to define better algorithms for variable orderings.
3. Develop metrics to compare variable ordering algorithms in the most fair manner.

To achieve these objectives we have built a benchmark in which 18 different algorithms for variable orderings have been implemented and compared on state space generation of the Petri nets taken from the models of the Model Checking context (both colored and uncolored), 2017 edition [23]. The implementation is part of RGMEDD [6], the model-checker of GreatSPN [5], and uses MDD saturation [14]. The ordering algorithms are either taken from the literature (both in their basic form and with a few new variations) or they were already included in GreatSPN. Figure 1, left, depicts the workflow we have followed in the benchmark.

Given a net system $\mathcal{S} = (\mathcal{N}, m_0)$ all ordering algorithms in $\mathcal{A}$ are run (box 1), then the reachability set RS_l of the system is computed *for each* ordering $l \in \mathcal{L}$ (box 2) and algorithms are ranked according to some MDD metrics $\mathrm{MM}(RS_l)$, (box 3). The best algorithm a^* is then the best algorithm for solving the PN system $\mathcal{S} = (\mathcal{N}, m_0)$ (box 4) and its state space RS_l could be the one used to check properties.

This workflow allows to: (1) provide indications on the best performing algorithm for a given model and (2) compare the algorithms in $\mathcal{A}$ on a large set of models to possibly identify the algorithm with the best average performance. The problem of defining a ranking among algorithms (or of identifying the "best" algorithm) is non-trivial and will be explored in Sect. 3.

Figure 1, right, shows a high level view of the approach used to compare variable ordering algorithms in the benchmark. Columns represent algorithms,

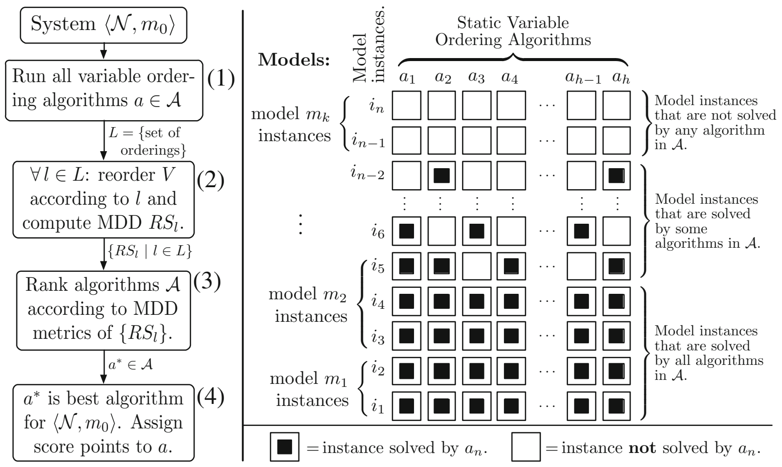

System $\langle \mathcal{N}, m_0 \rangle$
Run all variable ordering algorithms $a \in \mathcal{A}$ (1)
$L = \{$set of orderings$\}$
$\forall l \in L$: reorder V according to l and compute MDD RS_l. (2)
$\{RS_l \mid l \in L\}$
Rank algorithms $\mathcal{A}$ according to MDD metrics of $\{RS_l\}$. (3)
$a^* \in \mathcal{A}$
a^* is best algorithm for $\langle \mathcal{N}, m_0 \rangle$. Assign score points to a. (4)
Models:
Model instances.
Static Variable Ordering Algorithms
a_1 a_2 a_3 a_4 a_{h-1} a_h
model m_k instances
i_n i_{n-1} i_{n-2} i_6 i_5 i_4 i_3 i_2 i_1
model m_2 instances
model m_1 instances
Model instances that are not solved by any algorithm in $\mathcal{A}$.
Model instances that are solved by some algorithms in $\mathcal{A}$.
Model instances that are solved by all algorithms in $\mathcal{A}$.
■ = instance solved by a_n.
□ = instance **not** solved by a_n.

Fig. 1. Workflow for analysis and testing of static variable ordering algorithms.

and rows represent *model instances*, that is to say a Petri net model with an associated initial marking. A square in position (j, k) represents the state space generation for the j^{th} model instance done by GreatSPN using the variable ordering computed by algorithm a_k. A black square indicates that the state space was generated within the given time and memory limits.

In the analysis of the results from the benchmark we shall distinguish among model instances for which no variable ordering was good enough to allow GreatSPN to generate the state space (only white squares on the model instance row, as for the first two rows in the figure), model instances for which at least one variable ordering was good enough to generate the state space (at least one black square in the row), and model instances in which GreatSPN generates the state space with all variable orderings (all black squares in the row), that we shall consider "easy" instances.

In the analysis it is also important to distinguish whether we evaluate ordering algorithms w.r.t. all possible instances or on a representative set of them. Figure 1, right, highlights that instances are not independent, since they are often generated from the same "model" that is to say the same Petri net $\mathcal{N}$ by varying the initial marking m_0 or some other parameter (like the cardinality of the color classes). As we shall see in the experimental part, collecting measures over all instances, in which all instances have the same weight, may lead to a distortion of the observed behaviour, since the number of instances per model can differ significantly. A measure "per model" is therefore also considered.

This work could not have been possible without the models made available by the Model Checking Contest, the functions of the Meddly MDD library and the GreatSPN framework. We shall now review them in the following.

Model Checking Contest. The Model Checking Contest [23] is a yearly scientific event whose aim is to provide a comparison among the different available verification tools. The 2017 edition employed a set of 817 PNML instances generated from 75 (un)colored models, provided by the scientific community. The participating tools are compared on several examination goals, i.e. state space, reachability, LTL and CTL formulas. The MCC team has designed a score system to evaluate tools that we shall employ in a simplified version as one of the considered benchmark metrics for evaluating the algorithms, as evaluating the orderings can be reduced to evaluating the same tool, GreatSPN, in as many variations as the number of ordering algorithms considered.

Meddly Library. Meddly (Multi-terminal and Edge-valued Decision Diagram LibrarY) [10] is an open-source library implementation of Binary Decision Diagrams (BDDs) and several variants, including Multi-way Decision Diagrams (MDDs, implemented "natively") and Matrix Diagrams (MxDs, implemented as MDDs with an identity reduction rule). Users can construct one or more forests (collections of nodes) over the same or different domains (collections of variables). Several "apply" operations are implemented, including customized and efficient relational product operations and saturation [14] for generating the set of states (as an MDD) reachable from an initial set of states according to a transition relation (as an MxD). Saturation may be invoked either with an already known ("pre-generated") transition relation, or with a transition relation that is built "on the fly" during saturation [15], although this is currently a prototype implementation. The transition relation may be specified as a single monolithic relation that is then automatically split [26], or as a relation partitioned by levels or by events [16], which is usually preferred since the relation for a single Petri net transition tends to be small and easy to construct.

GreatSPN Framework. GreatSPN is a well-known collection of tools for the design and analysis of Petri net models [5,6]. The tools are aimed at the qualitative and quantitative analysis of Generalized Stochastic Petri Net (GSPN) [1] and Stochastic Symmetrical Net (SSN) through computation of structural properties, state space generation and analysis, analytical computation of performance indices, fluid approximation and diffusion approximation, symbolic CTL model checking, all available through a common graphical user interface [4]. The state space generation [9] of GreatSPN is based on Meddly. In this paper we use the collection of variable ordering heuristics implemented in GreatSPN. This collection has been enlarged to include all the variable ordering algorithms described in Sect. 2.

The paper is organized as follows: Sect. 2 reviews the considered algorithms; Sect. 3 describes the benchmark (models, scores and results); and Sect. 4 concludes the paper outlining possible future directions of work.

2 The Set $\mathcal{A}$ of Variable Ordering Algorithms

In this section we briefly review the algorithms considered by the benchmark. Although our target model category is that of Petri nets, we describe the algorithms in a more general form (as some of them were not defined specifically for Petri nets). We therefore consider the following high level description of the model.

Let V be the set of *variables*, that translates directly to MDD levels. Let E be the set of events in the model. Events are connected to *input* and *output* variables. Let $V^{\text{in}}(e)$ and $V^{\text{out}}(e)$ be the sets of input and output variables of event e, respectively. Let $E^{\text{in}}(v)$ and $E^{\text{out}}(v)$ be the set of events to which variable v participates as an input or as an output variable, respectively. For some models, structural information is known in the form of disjoint variable partitions named *structural units*. Let Π be the set of structural units. With $\Pi(v)$ we denote the unit of variable v. Let $V(\Pi)$ be the set of vertices in unit $\pi \in \Pi$. In this paper we consider three different types of structural unit sets. Let Π_{PSF} be the set of units corresponding to the P-semiflows of the net, obtained ignoring the place multiplicity. On some models, structural units are known because the model is composed of smaller parts. We mainly refer to [18] for the concept of Nested Units (NU). Let Π_{NU} be this set of structural units, which is defined only for a subset of models. Finally, structural units can be derived using a clustering algorithm. Let Π_{Cl} be the set of such units. We will discuss clustering in Sect. 2.6.

Following these criteria, we subdivide the set of algorithms $\mathcal{A}$ into $\mathcal{A}_{\text{Gen}}$, the set of algorithms that do not use any structural information; $\mathcal{A}_{\text{PSF}}$, the set of algorithms that require Π_{PSF}; and $\mathcal{A}_{\text{NU}}$, the set of algorithms that require Π_{NU}. Since clustering can be computed on almost any model, we consider methods that use Π_{Cl} as part of $\mathcal{A}_{\text{Gen}}$.

In our context, the set of MDD variables V corresponds to the place set of the Petri net, and the set of events is the transition set of the Petri net. Let $l : V \to \mathbb{N}$ be a variable order, i.e. an assignment of consecutive integer values to the variables V.

2.1 Breadth-First and Depth-First Orderings

Breadth-first and Depth-first search orderings (`BFS` and `DFS`) are two of the simplest possible variable ordering heuristics. They consist of a traversal of the net graph, starting from the first place, recording the visited places in breadth and depth order. These two methods usually show poor performance, but are included nonetheless in our tests since they are sometimes employed for BDD generation.

2.2 Force-Based Orderings

The Force heuristic, introduced in [3], is a n-dimensional graph layering technique based on the idea that variables form a hyper-graph, such that variables connected by the same event are subject to an "attractive" force, while variables not directly connected by an event are subject to a “repulsive” force. Events and variables are positioned over a real-valued line, and then sorted to get the ordering.

Algorithm 1. Pseudocode of the Force heuristic.

```
Function Force:
    Shuffle the variables randomly.
    repeat:
        for each event $e \in E$:
            compute center of gravity $cog_e = \frac{1}{|e|} \sum_{v \in e} l(v)$
        for each variable $v \in V$:
            compute hyper-position $p(v) = \frac{1}{E(v)} \sum_{e \in E(v)} cog_e$
        Sort vertices according the their $p(v)$ value.
        Compute $PTS^{(i)} = \sum_{e \in E} \sum_{v \in e} \left| cog_e - p(v) \right|$.
    until series of $PTS^{(i)}$ values monotonically decreases.
    return the variable order that had the smallest $PTS^{(i)}$ value.
```

Algorithm 1 gives the general skeleton of the Force algorithm. The algorithm starts by shuffling the variable set, then it iterates trying to achieve a convergence of a metric. Usually, different initial orders produce different final orders, so Force can be seen as a *factory* of variable orders. In addition, starting from a non-random initial order usually produces a better order. The metric is the total distance between the transition points and the variable points, known as *Point-Transition Spans* (PTS).

Structural information of the model can be used to establish additional *centers of gravity*. We tested the following three variations of the Force method:

– `Force`: Events are used as centers of gravity, as described in Algorithm 1.
– `Force-P`: P-semiflows are used as centers of gravity, along with the Petri net events. The method is tested only for those models that have P-semiflows.
– `Force-NU`: Structural units are used as centers of gravity, along with the events. This intuitively tries to keep together those variables that belong to the same structural unit. Again, this variation can be used only for those models that have Nested Units.

The set $\mathcal{A}$ of algorithms considered in the benchmark includes: `Force` in $\mathcal{A}_{\text{Gen}}$; the method `Force-P` in $\mathcal{A}_{\text{PSF}}$; and the method `Force-NU` in $\mathcal{A}_{\text{NU}}$, for a total of three variations of this method.

2.3 Bandwidth-Reduction Methods

Bandwidth-reduction(BR) methods are a class of algorithms that try to permute a sparse matrix into a band matrix, i.e. where most of the non-zero entries are confined in a (hopefully) small band near the diagonal. It is known [13] that reducing the event span in the variable order generally improves the compactness of the generated MDD. Therefore, event span reduction can be seen as an equivalent of a bandwidth reduction on a sparse matrix. A first connection between bandwidth-reduction methods and variable ordering has been tested in [21] and in [25] on the model bipartite graph. In these works the considered BR methods are:

- `CM`, `CM2` and `ACM`: The Reverse Cuthill-Mckee [17]. We test three implementations of this method: The one implemented in the Boost-C++ library (CM), the one implemented in the ViennaCL library (CM2), and the Advanced Cuthill-Mckee (ACM);
- `KING`: The King algorithm [22];
- `SLO` and `SLO-16`: The Sloan algorithm [29], with two parametric variations. The first version uses $W_1 = 1$ and $W_2 = 2$ (default values), while the second version uses $W_1 = 1$, $W_2 = 16$. Further information on these two variants can be found in [7].
- `GPS`: the Gibbs-Poole-Stockmeyer algorithm [19].

The choice was motivated by their ready availability in the Boost-C++ and ViennaCL libraries. In particular, Sloan, which is the state-of-the-art method for bandwidth reduction, showed promising performance as a variable ordering method. Sloan almost always outperforms [21] the other BR methods, but for completeness of our benchmark we have decided to test all of them. We concentrate our review on the Sloan method only, due to its effectiveness and its parametric nature.

The goal of the Sloan algorithm is to condense the entries of a symmetric square matrix $\mathbf{A}$ around its diagonal, thus reducing the matrix *bandwidth* and *profile* [29]. It works on symmetric matrices only, hence it is necessary to impose some form of translation of the model graph into a form that is accepted by the algorithm. The work in [25] adopts the symmetrization of the dependency graph of the model, i.e. the input matrix $\mathbf{A}$ for the Sloan algorithm will have $(|V| + |E|) \times (|V| + |E|)$ entries. We follow instead a different approach. The size of $\mathbf{A}$ is U, with $|V| \leq U \leq |V| + |E|$. Every event e generates entries in $\mathbf{A}$: when $|V^{\text{in}}(e)| \times |V^{\text{out}}(e)| < T$, where T is a threshold value, all entries in the cross product $V^{\text{in}}(e) \times V^{\text{out}}(e)$ are set to nonzero in $\mathbf{A}$. If instead $|V^{\text{in}}(e)| \times |V^{\text{out}}(e)| \geq T$, a pseudo-vertex v_e is added, and all $V^{\text{in}}(e) \times \{v_e\}$ and $\{v_e\} \times V^{\text{out}}(e)$ entries in $\mathbf{A}$ are set to be nonzero. Usually U will be equal to V, or just slightly larger. This choice better represents the variable–variable interaction matrix, while avoiding degenerate cases where a highly connected event could generate a dense matrix $\mathbf{A}$. In our implementation, the threshold T is set to 100. The matrix is finally made symmetric using: $\mathbf{A}' = \mathbf{A} + \mathbf{A}^{\mathrm{T}}$. As we shall see in Sect. 3, the computational cost of Sloan remains almost always bounded.

A second relevant characteristic of Sloan is its *parametric priority function* $P(v')$, which guides variable selection in the greedy strategy. A very compact pseudocode of Sloan is given in Algorithm 2. A more detailed one can be found in [24]. The method follows two phases. In the first phase it determines a pseudo-diameter of the $\mathbf{A}$ matrix graph, i.e. two vertices v, u that have an (almost) maximal distance. Usually, a heuristic approach based on the construction of

Algorithm 2. Pseudocode of the Sloan algorithm.

```
Function Sloan:
    Select a vertex $u$ of the graph.
    Select $v$ as the most-distant vertex to $u$ with a graph traversal.
    Establish a gradient from 0 in $v$ to $d$ in $u$ using a breadth-first traversal.
    Initialize traversal frontier $Q = \{v\}$
    repeat until $Q$ is empty:
        Remove from the frontier $Q$ the vertex $v'$ that minimizes $P(v')$.
        Add $v'$ to the variable ordering $l$.
        Add the unexplored adjacent vertices of $v'$ to $Q$.
```

the *root level structure* of the graph is employed. The method then performs a traversal, starting from v, exploring in sequence all vertices in the traversal frontier Q that maximize the priority function:

$$P(v') = -W_1 \cdot \mathit{incr}(v') + W_2 \cdot \mathit{dist}(v, v')$$

where $\mathit{incr}(v')$ is the number of unexplored vertices adjacent to v', $\mathit{dist}(v, v')$ is the distance between the initial vertex v and v', and W_1 and W_2 are two integer weights. The weights control how Sloan prioritizes the traversal of the local cluster (W_1) and how much the selection should follow the gradient (W_2). Since the two weights control a linear combination of factors, in our analysis we shall consider only the ratio $\frac{W_1}{W_2}$. Two ratios are tested: $\frac{W_1}{W_2} = \frac{1}{2}$, named `SLO`, and $\frac{W_1}{W_2} = \frac{1}{16}$, named `SLO-16`. An analysis of the parametric variations of Sloan for variable ordering selection can be found in [7].

2.4 P-Semiflows Chaining Algorithm

In this subsection we propose a new heuristic algorithm exploiting the Π_{PSF} set of structural units obtained by the P-semiflows computation. A P-semiflow is a positive, integer, left annuler of the incidence matrix of a Petri net, and it is known that, in any reachable marking, the sum of tokens in the net places, weighted by the P-semi-flow coefficients, is constant and equal to the weighted sum of the initial marking (P-invariant). Its main idea is to maintain the places shared between two Π_{PSF} units (i.e. P-semiflows) as close as possible in the final MDD variable ordering, since their markings cannot vary arbitrarily. The pseudo-code is reported in Algorithm 3.

The algorithm takes as input the Π_{PSF} set and returns as output a variable ordering (stored in the ordered l). Initially, the π_i unit sharing the highest number of places with another unit is removed by Π_{PSF} and saved in π_{curr}. All its places are added to l.

Then the main loop runs until Π_{PSF} becomes empty. The loop comprises the following operations. The π_j unit sharing the highest number of places with π_{curr} is selected. All the places of π_j in l, which are not currently in C (i.e. the list of currently discovered common places) are removed. The common places between

Algorithm 3. Pseudocode of the P-semiflows chaining algorithm.

```
Function P-chaining($\mathit{\Pi}_{\text{PSF}}$):
    $l = \varnothing$ is the ordered list of places.
    $C = \varnothing$ is the set of current discovered common places.
    Select a unit $\pi_i \in \mathit{\Pi}_{\text{PSF}}$ s.t. $\max_{\{i,j\} \in |\mathit{\Pi}_{\text{PSF}}|} \pi_i \cap \pi_j$ with $i \neq j$
    $\mathit{\Pi}_{\text{PSF}} = \mathit{\Pi}_{\text{PSF}} \setminus \{\pi_i\}$
    $\pi_{curr} = \pi_i$
    Append $V(\pi_{curr})$ to $l$
    repeat until $\mathit{\Pi}_{\text{PSF}}$ is empty:
        Select a unit $\pi_j \in \mathit{\Pi}_{\text{PSF}}$ s.t. $\max_{j \in |\mathit{\Pi}_{\text{PSF}}|} \pi_{curr} \cap \pi_j$
        Remove $(l \cap V(\pi_j)) \setminus C$ from $l$
        Append $V(\pi_{curr} \cap \pi_j) \setminus C$ to $l$
        Append $V(\pi_j) \setminus (C \cap V(\pi_{curr}))$ to $l$
        Add $V(\pi_{curr} \cap \pi_j)$ to $C$
        $\pi_{curr} = \pi_j$
        $\mathit{\Pi}_{\text{PSF}} = \mathit{\Pi}_{\text{PSF}} \setminus \{\pi_j\}$
    return $l$
```

π_i and π_j not present in C are appended to l. Then the places present only in π_j are added to l. After these steps, C is updated with the common places in π_i and π_j, and π_j is removed by $\mathit{\Pi}_{\text{PSF}}$. Finally π_{curr} becomes π_j, completing the iteration. This algorithm is named `P` and belongs to the $\mathcal{A}_{\text{PSF}}$ set.

2.5 The Noack and the Tovchigrechko Greedy Heuristics Algorithms

The Noack [27] and the Tovchigrechko [30] methods are greedy heuristics that build up the variable order sequence by picking, at every iteration, the variable that minimizes an objective function. A detailed description can be found in [20]. A pseudo-code is given in Algorithm 4.

Algorithm 4. Pseudocode of the Noack/Tovchigrechko heuristics.

```
Function NOACK-TOV:
    $S = \varnothing$ is the set of already selected places.
    for $i$ from 1 to $|V|$:
        compute weight $W(v) = f(v, S)$ for each $v \notin S$.
        find $v$ that maximizes $W(v)$.
        $l(i) = v$.
        $S \leftarrow S \cup \{v\}$.
    return the variable order $l$.
```

The main difference between the Noack and the Tovchigrechko methods is the weight function $f(v, S)$, defined as:

$$f_{\text{Noack}}(v,S) = \sum_{\substack{e \in E^{\text{out}}(v) \\ k_1(e) \wedge k_2(e)}} \big(g_1(e) + z_1(e)\big) \quad + \sum_{\substack{e \in E^{\text{in}}(v) \\ k_1(e) \wedge k_2(e)}} \big(g_2(e) + c_2(e)\big)$$

$$f_{\text{Tov}}(v,S) = \sum_{\substack{e \in E^{\text{out}}(v) \\ k_1(e)}} g_1(e) \quad + \sum_{\substack{e \in E^{\text{out}}(v) \\ k_2(e)}} c_1(e) \quad + \sum_{\substack{e \in E^{\text{in}}(v) \\ k_1(e)}} g_2(e) \quad + \sum_{\substack{e \in E^{\text{in}}(v) \\ k_2(e)}} c_2(e)$$

where the sub-terms are defined as:

$$g_1(e) = \frac{\max\big(0.1,\ |S \cap V^{\text{in}}(e)|\big)}{|V^{\text{in}}(e)|}, \qquad g_2(e) = \frac{1+|S \cap V^{\text{in}}(e)|}{|V^{\text{in}}(e)|}$$
$$c_1(e) = \frac{\max\big(0.1,\ 2 \cdot |S \cap V^{\text{out}}(e)|\big)}{|V^{\text{out}}(e)|}, \qquad c_2(e) = \frac{\max\big(0.2,\ 2 \cdot |S \cap V^{\text{out}}(e)|\big)}{|V^{\text{out}}(e)|}$$
$$z_1(e) = \frac{2 \cdot |S \cap V^{\text{out}}(e)|}{|V^{\text{out}}(e)|}, \quad k_1(e) = |V^{\text{in}}(e)| > 0, \quad k_2(e) = |V^{\text{out}}(e)| > 0$$

Not much technical information is known about the criteria that were followed for the definition of the f_{Noack} and f_{Tov} functions. An important characteristic is that both functions have different criteria for input and output event conditions, i.e. they do not work on the symmetrized event relation, like the Sloan method. The Noack and Tovchigrechko heuristics will be called `NOACK` and `TOV` in the benchmark.

2.6 Markov Clustering Heuristic

The heuristic `MCL` is based on the idea of exploring the effectiveness of clustering algorithms to improve variable order technique. The hypothesis is that in some models, it could be beneficial to first group places that belong to connected clusters. For our tests we selected the Markov Cluster algorithm [31]. The method works as a modified version of Sloan, where clusters are first ordered according to their average gradient, and then places belonging to the same cluster will appear consecutively on the variable ordering, following the cluster orders. This method is named `MCL` and belongs to the $\mathcal{A}_{\text{Gen}}$ set.

2.7 Gradient-*Π* Ordering

The Gradient-Π heuristic is a new heuristic that mixes a set of structural information Π with a gradient-like approach similar to the Sloan method. A detailed description can be found in [8]. We tested two variations of this method:

– `Grad-P`: the set Π is the set Π_{PSF} of P-semiflows of the net.
– `Grad-NU`: the set Π is the set Π_{NU} of Nested Units of the net.

A pseudo-code is given in Algorithm 5.

Gradient-*Π* shares with the P-chaining method the idea of ordering the variables taking one invariant at a time. The main differences are that (1) the structural units are ordered according to a *score*(π) function that is based on the gradient, and (2) the variables inside each unit π are again ordered in gradient order.

Algorithm 5. Pseudocode of the Gradient-Π heuristics.

```
Function Gradient-Π($v_0, \Pi$):
    Select $v$ as the most-distant vertex to $v_0$ with a graph traversal.
    Establish a gradient from 0 in $v$ to $d$ in $v_0$ using a breadth-first traversal.
    $l \leftarrow \{\}$
    while exists at least one $\pi \in \Pi$ with $\pi \setminus S \neq \varnothing$:
        for each element $\pi \in \Pi$ with $\pi \setminus S \neq \varnothing$:
            Compute $score(\pi) = \sum_{v \in \pi \cap S} grad(v) - \sum_{v \in \pi \setminus S} grad(v)$
        Let $\pi_{\max}$ be the element with maximum $score(\pi)$ value.
        Append variables in $(\pi_{\max} \setminus S)$ to $l$ in ascending gradient order.
        $S \leftarrow S \cup \pi_{\max}$.
    Append all variables in $(V \setminus S)$ to $l$ in ascending gradient order.
    return $l$.
```

3 The Benchmark

The considered model instances are that of the Model Checking Contest, 2017 edition [23], which consists of 817 PNML files. We discarded several instances that our tool was not capable to solve in the imposed time and memory limits, because either the net was too big or the RS MDD was too large under any considered ordering. Thus, we considered for the benchmark the set $\mathcal{I}$ made of 393 instances, belonging to a set $\mathcal{M}$ of 69 models. These 393 instances run for the 18 tested algorithms for 20 min, with 4 GB of memory and a decision diagram cache of 2^{26} entries. In the 393 instances of $\mathcal{I}$ two sub-groups are identified: The set $\mathcal{I}_{\text{PSF}} \subset \mathcal{I}$ of instances for which P-semiflows are available, with 315 instances generated from a subset $\mathcal{M}_{\text{PSF}}$ of 62 models; The set $\mathcal{I}_{\text{NU}} \subset \mathcal{I}$ of instances for which nested units are available, with 109 instances generated from a subset $\mathcal{M}_{\text{NU}}$ of 15 models.

The overall tests were performed on OCCAM [2] (Open Computing Cluster for Advanced data Manipulation), a multi-purpose flexible HPC cluster designed and maintained by a collaboration between the University of Torino and the Torino branch of the National Institute for Nuclear Physics. OCCAM contains slightly more than 1100 CPU cores including three different types of computing nodes: standard Xeon E5 dual-socket nodes, large Xeon E5 quad-sockets nodes with 768 GB RAM, and multi-GPU NVIDIA nodes.

Scores. Typically, the most important parameter that measures the performance of variable ordering is the MDD peak size, measured in nodes. The peak size represents the maximum MDD size reached during the computation, and it therefore represents the memory requirement. It is also directly related to the time cost of building the MDD. For these reasons we preferred to use the peak size alone instead of weighted measures of time, memory and peak size, that would make interpretation of the results more complex. The peak size is, however, a quantity that is strictly related to the model instance. Different instances of the same model will have unrelated peak sizes, often with different magnitudes. To treat instances in a balanced way, some form of normalized score is needed. We consider three different score functions: for all of them the score of an algorithm is first normalized against the other algorithms on the same instance, which gives a score per instance, and then summed over all instances. Let i be an instance, solved by algorithms $\mathcal{A} = \{a_1, \dots, a_m\}$ with peak nodes $P_i = \{p_{a_1}(i), \dots, p_{a_m}(i)\}$. The scores of an *algorithm a for an instance i* are:

- The *Mean Standard Score of instance i* is defined as: $\mathrm{MSS}_a(i) = \frac{\mu_{\mathcal{A}}(i) - p_a(i)}{\sigma_{\mathcal{A}}(i)}$, where $\mu_{\mathcal{A}}(i)$ and $\sigma_{\mathcal{A}}(i)$ are the mean and standard deviations for instance I summed over all algorithms that solved instance i.
- The *Normalized Score for instance i* is defined as: $\mathrm{NS}_a(i) = \frac{\min\{p \in P_i\}}{p_a(i)}$, which just rescales the peak nodes taking the minimum as the scaling factor.
- The *Model Checking Contest score*[1] *for instance i* is defined as: $\mathrm{MCC}_a(i) = 48$ if a terminates on i in the given time bound, plus 24 if $p_a(i) = \min\{p \in P_i\}$.

The final score used for ranking algorithms over a set $\mathcal{I}' \subseteq \mathcal{I}$ is then determined as the sum over $\mathcal{I}'$ of the scores per instance:

- The *Mean Standard Score* of algorithm a: $\mathrm{MSS}_a = \frac{1}{|\mathcal{I}'|} \sum_{i \in \mathcal{I}'} \mathrm{MSS}_a(i)$
- The *Normalized Score* of algorithm a: $\mathrm{NS}_a = \frac{1}{|\mathcal{I}'|} \sum_{i \in \mathcal{I}'} \mathrm{NS}_a(i)$
- The *Model Checking Contest* score of a: $\mathrm{MCC}_a = \frac{1}{|\mathcal{I}'|} \sum_{i \in \mathcal{I}'} \mathrm{MCC}_a(i)$.

MSS requires a certain amount of samples to be significant. Therefore, we apply it only for those model instances were all our tested algorithms terminated in the time bound. The set of instances where all algorithms finish is named *"Easy instances"* hereafter. We use MSS^*_a, NS^*_a and MCC^*_a to denote that the score is computed on the set of Easy instances only. If we instead consider all the instances, where some algorithms could not finish in time, we may apply only the NS and the MCC score. We decided to test the MCC score to check if it is a good or a biased metric, when compared to the standard score.

Score Normalization: One problem of this benchmark setup is that the MCC model set is made by multiple parametric instances of the same model, and the number of model instances per model vary largely. Some models have just one instance, while other models have up to 40 instances. Usually, the relative performance of each algorithm on different instances of the same model are similar, according to our experience. Thus, an instance-oriented benchmark is biased toward those models that have more instances. Therefore, we consider two benchmark settings for the computation of the scores:

- *Scores "By Instance"*: each model instance has the same weight, so those models with many instances will be more important. This reflects more closely the MCC score model.

[1] We actually use a simplified version where weights, model categories and answer correctness are not considered.

– *Scores "By Model"*: each score value is normalized against the number of instances $\mathcal{I}_m$ of each considered model $m \in \mathcal{M}'$. Therefore, MSS_a is redefined as:

$$\overline{\text{MSS}}_a = \tfrac{1}{|\mathcal{M}'|} \sum_{m \in \mathcal{M}'} \tfrac{1}{|\mathcal{I}_m|} \sum_{i \in \mathcal{I}_m} \text{MSS}_a(i)$$

Analogously, $\overline{\text{NS}}_a$ and $\overline{\text{MCC}}_a$ are by-model versions of the NS and MCC scores.

In our opinion, the normalization of the by-model scores reflects more closely the idea of "average behaviour" of a variable ordering heuristic when applied to a new, unknown problem.

Assumptions: The computations are carried out by the GreatSPN model checker, which uses saturation by-events to generate the MDD representation of the state space. Since we use the MDD peak size for score computation, this is relevant, since other RS algorithms could produce different scores. We use the "optimistic" heuristic for cache management of Meddly, which means that created nodes are kept in cache even if they are not used any longer, and are reclaimed only when the garbage collector is called. Therefore, the peak size is approximatively the total number of generated nodes, except when the memory consumption is too large and the garbage collector needs to be called. We employ Multi-terminal Decision Diagrams for storing token values, with one level per place. Other techniques (BDD with multi-level encoding of token counts, MDD with multiple places per level, etc.) are not considered. We assume also that the model selection made for the MCC model set is somewhat "fair". In principle this is true, in the sense that the models have been selected using various criteria (variety, interest, case studies, ...) that have nothing to do with the performance of the saturation algorithm.

Results: Table 1 describes the general summary of the benchmark results. For each algorithm, we report again its requirement class ($\mathcal{A}_{\text{Gen}}, \mathcal{A}_{\text{PSF}}, \mathcal{A}_{\text{NU}}$). The table reports the average results for the "By instances" and "By Models" normalization. For the"By instances" group, it reports the number of instances where the algorithm is applied, the number of solved instances in the time and memory bounds, and the number of times the algorithm finds the best ordering among the others. The next four columns report the NS score on all instances NS_a, the MSS score on the easy instances (MSS^*_a), the NS score on the easy instances (NS^*_a), and the MCC_a score on all instances. The structure is repeated for the "By model" normalization. First the number of applied models is reported, followed by the number of models solved by each algorithm (which can be fractional, since when an algorithm solves only some instances of a model, it gets a fractional score) and the number of best orders found. Finally, the scores are repeated, normalized by model.

From the table emerges that the Sloan algorithm has the best average performance on the MCC models, with a clear margin. It is then followed by the `TOV/NOACK` heuristics, and the `Force` heuristics. Methods that exploit structural information also show very positive results. Gradient-P (which is applied only

Table 1. Performance of the ordering algorithms using the MCC2017 models.

Algorithms		By instances							By models						
		Instances			Average scores				Models			Average scores			
Name	$\mathcal{A}$	N	solv.	Best	NS_a	MSS^*_a	NS^*_a	MCC_a	N	solv.	Best	$\overline{\mathrm{NS}}_a$	$\overline{\mathrm{MSS}}^*_a$	$\overline{\mathrm{NS}}^*_a$	$\overline{\mathrm{MCC}}_a$
`Force`	$\mathcal{A}_{\mathrm{Gen}}$	393	289	41	0.37	−0.16	0.21	37.80	69	27.31	9.89	0.79	−0.39	0.44	41.34
`BFS`	$\mathcal{A}_{\mathrm{Gen}}$	393	188	12	0.10	0.46	0.07	23.69	69	5.80	1.34	0.52	1.03	0.11	25.50
`DFS`	$\mathcal{A}_{\mathrm{Gen}}$	393	187	3	0.08	0.41	0.07	23.02	69	5.11	0.62	0.50	0.76	0.12	24.34
`CM`	$\mathcal{A}_{\mathrm{Gen}}$	393	263	42	0.31	−0.13	0.18	34.69	69	16.35	2.81	0.68	−0.25	0.31	33.79
`CM2`	$\mathcal{A}_{\mathrm{Gen}}$	393	274	46	0.27	0.08	0.13	36.27	69	16.30	7.82	0.70	0.27	0.21	36.46
`ACM`	$\mathcal{A}_{\mathrm{Gen}}$	393	275	46	0.27	0.08	0.13	36.40	69	16.25	7.82	0.70	0.27	0.21	36.49
`GPS`	$\mathcal{A}_{\mathrm{Gen}}$	393	276	46	0.27	0.08	0.13	36.52	69	16.31	7.82	0.70	0.27	0.21	36.53
`KING`	$\mathcal{A}_{\mathrm{Gen}}$	393	249	27	0.28	−0.13	0.18	32.06	69	15.43	2.16	0.66	−0.21	0.30	32.62
`SLO`	$\mathcal{A}_{\mathrm{Gen}}$	393	327	36	0.39	−0.13	0.20	42.14	69	24.76	4.65	0.88	−0.32	0.37	43.99
`SLO-16`	$\mathcal{A}_{\mathrm{Gen}}$	393	**331**	43	0.40	−0.12	0.19	43.05	69	26.24	8.31	0.89	−0.32	0.39	45.56
`NOACK`	$\mathcal{A}_{\mathrm{Gen}}$	393	290	37	0.37	−0.12	0.21	37.68	69	29.18	6.87	0.77	−0.32	0.45	39.55
`TOV`	$\mathcal{A}_{\mathrm{Gen}}$	393	293	46	0.38	−0.12	0.21	38.60	69	29.30	8.90	0.80	−0.32	0.46	41.59
`MCL`	$\mathcal{A}_{\mathrm{Gen}}$	393	223	13	0.20	0.11	0.12	28.03	69	13.00	2.38	0.64	0.16	0.22	31.38
Algorithms that require P-semiflows:															
`P-chain`	$\mathcal{A}_{\mathrm{PSF}}$	315	234	18	0.22	0.05	0.14	37.03	62	13.30	4.35	0.80	0.24	0.26	40.09
`GradP`	$\mathcal{A}_{\mathrm{PSF}}$	315	260	**73**	**0.55**	−0.18	**0.25**	45.18	62	**32.98**	**13.97**	0.87	−0.40	**0.51**	47.27
`ForceP`	$\mathcal{A}_{\mathrm{PSF}}$	315	255	35	0.39	**−0.19**	0.22	41.52	62	23.19	6.44	0.82	**−0.41**	0.43	42.01
Algorithms that require Nested Units:															
`GradNU`	$\mathcal{A}_{\mathrm{NU}}$	109	92	39	0.66	−0.07	0.11	**49.10**	15	10.19	6.67	**0.94**	−0.21	0.34	**55.82**
`ForceNU`	$\mathcal{A}_{\mathrm{NU}}$	109	85	1	0.23	−0.07	0.05	37.65	15	2.35	0.09	0.80	−0.18	0.12	38.45

to 315 instances out of 393) is particularly effective in finding the best variable orders most of the time. Similarly, Gradient-NU has again very positive scores.

Other methods, like `BFS` and `DFS` have usually a bad average behaviour, even if they perform well on a small subset of instances. Considering the column of "best" instances, some methods seem to perform well, like `CM`, but this is a bias caused by the uneven number of instances per model (i.e. some models have more instances than others). In fact, the behaviour of the `CM` algorithm when weighted by-models is much more modest. To our surprise, the `P-chaining` method shows only a mediocre average performance even though there are several instances where it performs particularly well. In general, most instances are solved by more than one algorithm. In the rest of the section we go into model details, first considering the performance of the algorithms, and then by ranking the methods considering either instances($\mathcal{I}$, $\mathcal{I}_{\mathrm{PSF}}$, $\mathcal{I}_{\mathrm{NU}}$) or models($\mathcal{M}$, $\mathcal{M}_{\mathrm{PSF}}$, $\mathcal{M}_{\mathrm{NU}}$).

3.1 Results of the Benchmark

Figure 2 shows the benchmark results, separated by instance class and algorithm class. The plots on the left (1, 3, and 5) report the results on the Easy instances, while those on the right report the results for all the instances. In the left plots

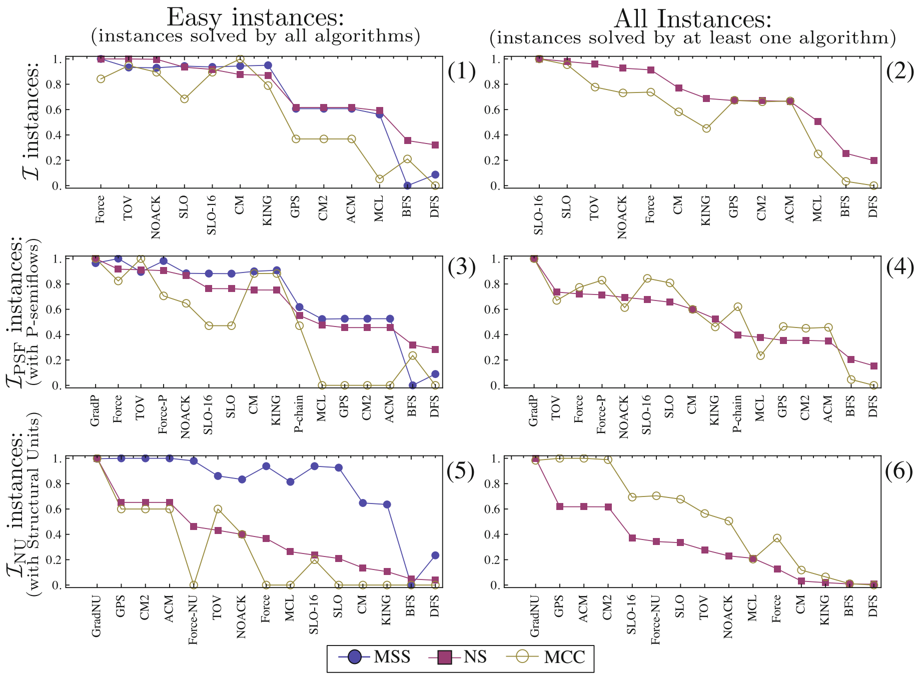


Fig. 2. Benchmark results, weighted by instance.

we report the three tested metrics, while on the right plots we discard the MSS metric, since the available samples for each instance may vary and could be too low for the Gaussian assumption. To fit all scores in a single plot we have rescaled the score values in the [0,1] range.

Algorithms are sorted by their NS score, best one on the left. The top row (plot 1 and 2) considers the $\mathcal{A}_{\mathrm{Gen}}$ methods on all $\mathcal{I}$ instances. The center row considers the $\mathcal{A}_{\mathrm{Gen}} \cup \mathcal{A}_{\mathrm{PSF}}$ methods on the $\mathcal{I}_{\mathrm{PSF}}$ instances. The bottom row considers the $\mathcal{A}_{\mathrm{Gen}} \cup \mathcal{A}_{\mathrm{NU}}$ methods on the $\mathcal{I}_{\mathrm{NU}}$ instances. Plots 1, 3, and 5 consider 152, 122 and 15 instances, respectively, which are the "easy" instances. The Easy instances are in a certain sense a biased set, since instances are dropped (not easy) every time any algorithm fails in generating a reasonable variable order. However, from these plots it is possible to observe that the trend of the NS score is close to that of the MSS score. Therefore, we will mainly consider the NS score only, since it can be computed on the whole set of instances. Plots 5 and 6 use fewer samples than the others, and the algorithm ranking is slightly different. We suspect that this discrepancy can be explained by the small number of available instances. Therefore, we mainly focus our attention on the other plots. The Sloan methods and the Gradient-Π methods appear to have the best performance, in terms of both the NS score and the MCC score. When we look at the average behaviour on all instances (plot 2), we may also observe that `TOV`/`NOACK` and `Force` have close-to-best performance. This

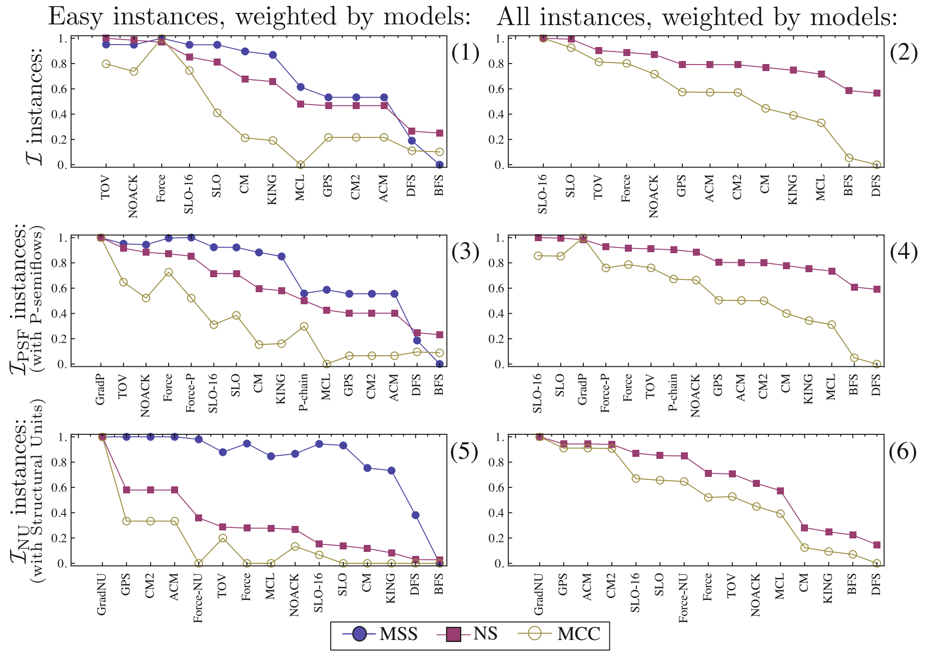


Fig. 3. Benchmark results, weighted by model.

shows that being capable of using the structural information of the model for the purpose of variable ordering can be an effective strategy.

Figure 3 shows the benchmark results weighted "By models". Plots 1, 3, and 5 consider 56, 50 and 6 models, respectively, which are those models which have at least one "easy" instance. Some methods have a different ranking due to the bias introduced by the uneven number of model instances. For example, the performance of the `CM` method appears to be worse when weighting "By models". This is explained by the fact that `CM` performs well on three models (BridgeAndVehicle, Diffusion2D and SmallOperatingSystem) that have a large number of instances, but on average it does not produce very good variable orders. Again, the ranking has the Sloan method on top for the $\overline{\text{NS}}$ score on plots 2 and 4. The Gradient-Π methods show a more modest NS score, even if they are still on top of the rankings. However, the MCC score of these two methods are very high, suggesting their effectiveness (plot 4 and 6). Again, the number of Easy instances is very small and does not allow to make conclusions on the results of plots 1, 3 and 5.

As stated before, the ranking of the NS score is not the same as the MCC score. To better investigate this behaviour we look at the NS score distributions of the algorithms in Fig. 4.

Figure 4 shows the NS score distributions on the 393 model instances. The bar of each algorithm a shows the distribution of the NS scores obtained by

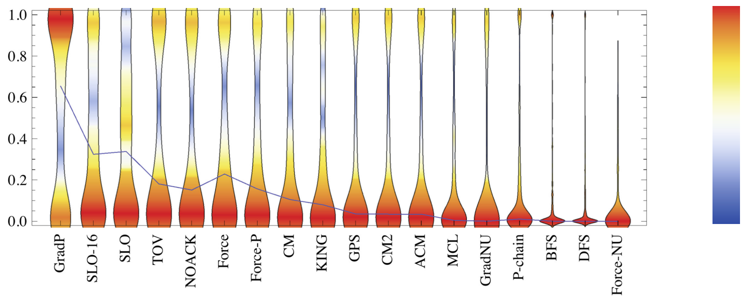


Fig. 4. NS score distributions for the "By Instance" case, on all models.

a for all the instances. From the distribution, it is clear that most algorithms have very polarized behaviours (either they produce a very good order, or they fail). However, Sloan has a more continuous distribution, meaning that on many instances it is capable of producing a reasonable variable order, even if it is not the best among the generated ones. Therefore, a method that *on-average* performs reasonably well will have an high MCC score, like `SLO-16` in Fig. 2(2). Also, `Grad-P` has a very high chance of finding the best solution among the tested ones, showing that it is a very promising heuristic.

4 Conclusions and Future Works

In this paper we presented a comparative benchmark of 18 variable ordering algorithms. Some of these algorithms are popular among Petri net based model checkers, while others have been defined to investigate the use of structural information for variable orderings. We observed that among the generic methods, the Sloan method, the Tovchigrechko/Noack methods and the Force method have the best performance, and their effectiveness covers different subsets of model instances. While the methods of Tovchigrechko/Noack were designed for Petri net models, the method of Sloan is a standard algorithm for bandwidth reduction of matrices, whose effectiveness for variable ordering was pointed out only recently in [21] and [25]. We conjecture that a key ingredient of the Sloan method is the gradient. This conjecture has been used in [8] to design the new heuristic Gradient-Π, which is an hybrid between Sloan and P-chain. This heuristic proves to have very good performance. We conjecture that other algorithms, like `TOV` or `Force`, could be improved by using a superimposed gradient. When the net has some structural information, like P-semiflows or Nested Units, we observed that only some specialized algorithms could take a significant advantage from it. Surprisingly, the P-chaining method (one of the original heuristics of GreatSPN) showed poor performance when compared to more modern algorithms like Gradient-Π. However, the general results of other heuristics (Force-P, Gradient-Π) allow us to conclude that structural information is useful in deriving good variable orders. Of course, exploitation of structural information cannot be a general technique, since it is not available for all models.

We also tested three different scoring metrics. We observed an agreement between the MSS and the NS score, which is nice since NS can be used even when few algorithms complete. We also observed that (our simplified version of) MCC is a good score, that favours a different aspect than MSS/NS, i.e. MCC favours the average behaviour over finding better ordering. The use of a per-model weight on the scores has helped in identifying a bias in the benchmark results. We think that some form of per-model weight is necessary when using the MCC model set. Another important observation is that all these algorithms show a significant instability: they could work nicely for some models, and produce terrible orders for other models. The consequences of this instability are two-fold. In principle, using a single variable order algorithm may prove to be effective on a subset of models. In addition, the results are highly influenced by the MCC model set. It remains unknown if the observations made in this paper remains the same when using a different model set.

It should be noted that we observed a ranking similar to the one reported in [25]. That paper deals with metrics for variable ordering (without RS construction), but in the last section the authors report a small experimental assessment similar to our benchmark. In that assessment Tovchigrechko was not tested, and the best algorithms were mostly Sloan and Force. For Sloan, they used the default parameter setting with a rather different symmetrization of the adjacency matrix. In addition, the tested model set was different (106 instances). However, the final observations drawn in that paper are close to the ones we get from our tests, confirming the effectiveness of the Sloan method for variable ordering.

Reproducibility: All results (tool execution logs, raw csv data, processed data) are available at: www.di.unito.it/~amparore/PNSE17/. The source code of the tool is available at: github.com/greatspn/SOURCES.

Acknowledgement. We would like to thank the MCC team and all colleagues that collaborated with them for the construction of the MCC database of models, and the Meddly library developers.

References

1. Ajmone Marsan, M., Conte, G., Balbo, G.: A class of generalized stochastic Petri nets for the performance evaluation of multiprocessor systems. ACM Trans. Comput. Syst. **2**, 93–122 (1984)
2. Aldinucci, M., Bagnasco, S., Lusso, S., Pasteris, P., Vallero, S., Rabellino, S.: The open computing cluster for advanced data manipulation (OCCAM). In: 22nd International Conference on Computing in High Energy and Nuclear Physics, San Francisco (2016)
3. Aloul, F.A., Markov, I.L., Sakallah, K.A.: FORCE: a fast and easy-to-implement variable-ordering heuristic. In: Proceedings of GLSVLSI, pp. 116–119. ACM, New York (2003)
4. Amparore, E.G.: Reengineering the editor of the GreatSPN framework. In: PNSE@ Petri Nets, pp. 153–170 (2015)

5. Amparore, E.G., Balbo, G., Beccuti, M., Donatelli, S., Franceschinis, G.: 30 years of GreatSPN. In: Fiondella, L., Puliafito, A. (eds.) Principles of Performance and Reliability Modeling and Evaluation. SSRE, pp. 227–254. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-30599-8_9
6. Amparore, E.G., Beccuti, M., Donatelli, S.: (Stochastic) model checking in GreatSPN. In: Ciardo, G., Kindler, E. (eds.) PETRI NETS 2014. LNCS, vol. 8489, pp. 354–363. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07734-5_19
7. Amparore, E.G., Donatelli, S., Beccuti, M., Garbi, G., Miner, A.: Decision diagrams for Petri nets: which variable ordering? In: Petri Net Performance Engineering Conference (PNSE), pp. 31–50. CEUR-WS (2017)
8. Amparore, E.G., Beccuti, M., Donatelli, S.: Gradient-based variable ordering of decision diagrams for systems with structural units. In: D'Souza, D., Narayan Kumar, K. (eds.) ATVA 2017. LNCS, vol. 10482, pp. 184–200. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68167-2_13
9. Baarir, S., Beccuti, M., Cerotti, D., Pierro, M.D., Donatelli, S., Franceschinis, G.: The GreatSPN tool: recent enhancements. Perform. Eval. **36**(4), 4–9 (2009)
10. Babar, J., Miner, A.: Meddly: multi-terminal and edge-valued decision diagram library. In: International Conference on, Quantitative Evaluation of Systems, pp. 195–196. IEEE Computer Society, Los Alamitos (2010)
11. Bollig, B., Wegener, I.: Improving the variable ordering of OBDDs is NP-complete. IEEE Trans. Comput. **45**(9), 993–1002 (1996)
12. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. IEEE Trans. Comput. **35**, 677–691 (1986)
13. Burch, J.R., Clarke, E.M., Long, D.E.: Symbolic model checking with partitioned transition relations. In: IFIP TC10/WG 10.5 Very Large Scale Integration, pp. 49–58. North-Holland (1991)
14. Ciardo, G., Lüttgen, G., Siminiceanu, R.: Saturation: an efficient iteration strategy for symbolic state—space generation. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 328–342. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45319-9_23
15. Ciardo, G., Marmorstein, R., Siminiceanu, R.: Saturation unbound. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 379–393. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-36577-X_27
16. Ciardo, G., Yu, A.J.: Saturation-based symbolic reachability analysis using conjunctive and disjunctive partitioning. In: Borrione, D., Paul, W. (eds.) CHARME 2005. LNCS, vol. 3725, pp. 146–161. Springer, Heidelberg (2005). https://doi.org/10.1007/11560548_13
17. Cuthill, E., McKee, J.: Reducing the bandwidth of sparse symmetric matrices. In: Proceedings of the 1969 24th National Conference, pp. 157–172. ACM, New York (1969)
18. Garavel, H.: Nested-unit petri nets: a structural means to increase efficiency and scalability of verification on elementary nets. In: Devillers, R., Valmari, A. (eds.) PETRI NETS 2015. LNCS, vol. 9115, pp. 179–199. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-19488-2_9
19. Gibbs, N., Poole, W., Stockmeyer, P.: An algorithm for reducing the bandwidth and profile of a sparse matrix. SIAM J. **13**(2), 236–250 (1976)
20. Heiner, M., Rohr, C., Schwarick, M., Tovchigrechko, A.A.: MARCIE's secrets of efficient model checking. In: Koutny, M., Desel, J., Kleijn, J. (eds.) Transactions on Petri Nets and Other Models of Concurrency XI. LNCS, vol. 9930, pp. 286–296. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53401-4_14

21. Kamp, E.: Bandwidth, profile and wavefront reduction for static variable ordering in symbolic model checking. University of Twente, Technical report, June 2015
22. King, I.P.: An automatic reordering scheme for simultaneous equations derived from network systems. J. Numer. Methods Eng. **2**(4), 523–533 (1970)
23. Kordon, F., et al.: Complete Results for the 2017 Edition of the Model Checking Contest, June 2017. http://mcc.lip6.fr/2017/results.php
24. Kumfert, G., Pothen, A.: Two improved algorithms for envelope and wavefront reduction. BIT Numer. Math. **37**(3), 559–590 (1997)
25. Meijer, J., van de Pol, J.: Bandwidth and Wavefront reduction for static variable ordering in symbolic reachability analysis. In: Rayadurgam, S., Tkachuk, O. (eds.) NFM 2016. LNCS, vol. 9690, pp. 255–271. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40648-0_20
26. Miner, A.S.: Implicit GSPN reachability set generation using decision diagrams. Perform. Eval. **56**(1–4), 145–165 (2004)
27. Noack, A.: A ZBDD package for efficient model checking of Petri nets (in German). Ph.D. thesis, BTU Cottbus, Department of CS (1999)
28. Rice, M., Kulhari, S.: A survey of static variable ordering heuristics for efficient BDD/MDD construction. University of California, Technical report (2008)
29. Sloan, S.W.: An algorithm for profile and wavefront reduction of sparse matrices. Int. J. Numer. Methods Eng. **23**(2), 239–251 (1986)
30. Tovchigrechko, A.: Model checking using interval decision diagrams. Ph.D. thesis, BTU Cottbus, Department of CS (2008)
31. Van Dongen, S.: A cluster algorithm for graphs. Inform. Syst. **10**, 1–40 (2000)

Model Synchronization and Concurrent Simulation of Multiple Formalisms Based on Reference Nets

Pascale Möller, Michael Haustermann(✉), David Mosteller, and Dennis Schmitz

Faculty of Mathematics, Informatics and Natural Sciences, Department of Informatics, University of Hamburg, Hamburg, Germany
`{2moeller,haustermann,mosteller,schmitz}@informatik.uni-hamburg.de`
`http://www.informatik.uni-hamburg.de/TGI/`

Abstract. Nowadays, modeling complex systems requires a combination of techniques to facilitate multiple perspectives and adequate modeling. Therefore, UML and other formalisms are used in combination with Petri nets. Often the different models are transformed into a single formalism to simulate the resulting models within a homogeneous execution environment. For UML, the mapping is usually done via the transformation to some programming language.

Anyhow, the problem with generative techniques is that the different perspectives that are provided by the applied modeling techniques can hardly be retained once the models are transformed into a single formalism. In this contribution we elaborate on how multiple formalisms can be used together in their original representation.

One of the main challenges for our approach is the provision of means for coupling mentioned formalisms so they can be executed together. We utilize the synchronization features of Reference Nets to couple multiple modeling techniques. Therefore, we present an approach to transform modeling languages into Reference Nets, which can be executed with the simulation environment Renew. The simulation events are forwarded to the original representation in the form of graphical user feedback and interaction.

This results in a simultaneous and concurrent execution of models featuring a combination of multiple modeling formalisms. A finite automata modeling and simulation tool is presented to showcase the application of our concept. Based on our results, we present a case study that utilizes finite automata in combination with Reference Nets and activity diagrams.

Keywords: Petri nets · Multi-formalism · Model synchronization Simulation · Reference Nets · Finite automata · Activity diagrams

1 Introduction

Modeling complex systems requires a separation of concerns and demands support for taking into account multiple perspectives on the system including various levels of abstraction. This is achieved by combining several modeling techniques.

Looking back at the original ideas of Petri [29], automata are enhanced by a communication mechanism, thus introducing the modeling of concurrency. The new formalism, he calls “net”, facilitates the modeling of additional concepts: locality (of time and place), asynchronism and concurrency. With his conceptual extension to the formalism comes a shift in perspective. In consequence, nets may serve for other purposes than automata.

Petri nets in their various forms have proven to be an adequate technique to model, analyze and understand concurrent systems. They provide an operational semantics, and by utilizing high-level Petri nets, modelers have a technique at hand to cover multiple abstraction levels and perspectives. Anyhow, Petri nets are not always the optimal solution for every modeling task. We agree with Milner, who says: “I reject the idea that there can be a unique conceptual model, or one preferred formalism, for all aspects of something as large as concurrent systems modeling” [25, p. 78].

Depending on the context and the application area, various requirements to the modeling technique have to be met. Extending a formalism (or creating a new one) allows to shift the focus to other aspects, thus providing a different abstraction. Multiple techniques may be combined to facilitate various perspectives in order to cover different levels of abstraction (vertical) or for a combination of complementing views (horizontal).

Today, modeling techniques are quite elaborate, and the means for modeling are systematically developed and researched. The purposes for constructing *conceptual models* and their usage areas are manyfold. Krogstie [19, p. 3] has put them into categories: models may be used for system (forward) design, for the (computer-assisted) analysis, to support communication, for quality assurance, model deployment and activation, or just for making sense out of something.

The complexity of systems has dramatically increased since the beginning of computer science. This raises the necessity to provide several perspectives at the same time. Taking into account different perspectives only makes sense if they can be related or linked in some way.

Standard modeling languages like UML (Unified Modeling Language) offer a portfolio of techniques to provide multiple modeling perspectives covering various levels of abstraction. Model driven approaches facilitate the development of domain specific languages that can be related on the basis of meta-models. Usually, they are transformed into a single target language. If no reverse transformation for visualizing the simulation state exists, the transformation comprises a loss of the desired abstraction level and may blur the previously given perspective. Model simulation environments support the execution of models, but models usually do not provide the means to be used in combination.

Let us assume, a software application is modeled using three different techniques. The structure is modeled with a Component Diagram, the components' behavior is modeled by means of Sequence Diagrams and the data is modeled by an Entity Relation Diagram. In order to execute the modeled application, the three different techniques are transformed into a single target programming language. This results in a number of files/classes/functions from which we cannot easily derive the former perspectives (structure, behavior, data). On the technical level of the execution language, it is challenging to investigate a running application without proper tool support.

The concurrent execution of multiple modeling techniques in their original representation using explicit model coupling is promising in order to combine the advantages of linking models and direct simulation. Consequently, tool support is required not only to support the modeling of multiple perspectives, but also to integrate the different views and to keep them consistent with each other.

We identify the following two main challenges: (1) For the generic coupling of multiple modeling techniques an adequate mechanism has to be found. (2) The coupled models from various modeling techniques have to be concurrently simulated in one environment without losing the original representation.

This work is an extended version of a workshop publication [26]. We propose a conceptual extension of current modeling techniques by synchronous channels [7] as they are provided in the context of Reference Nets. Synchronous channels can supply synchronous communication between models in order to exchange data or control other models. Furthermore, we propose to map the constructs of the modeling techniques to Reference Net constructs. It should be noted that this mapping is not equivalent to the transformation that we reject above, since the original representation of the model is preserved and even visually observable during simulation including the possibility to interact with the simulation. For modelers it should appear as if a model is being simulated in its original modeling technique. In order to achieve this, we propagate the simulation events of the underlying Reference Net back to the original model.

2 Proposal for Coupling Through Synchronization

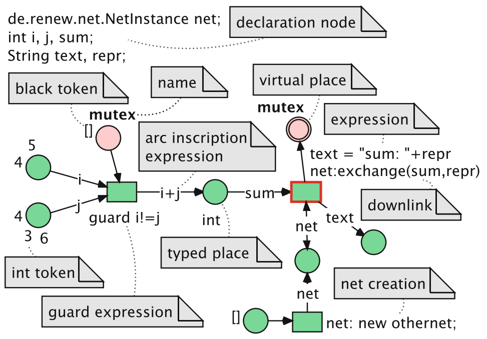
In this section we briefly introduce the conceptual and technical background of our work and present a simple introductory example of the coupling of multiple formalisms. First, we present the Renew environment for modeling and execution of Reference Nets and other modeling techniques. (Java) Reference Nets as our main modeling and simulation formalism are introduced afterwards. Last, with a simple example, we demonstrate our idea of multi-formalism execution, which is presented in general in Sect. 3.

2.1 Renew

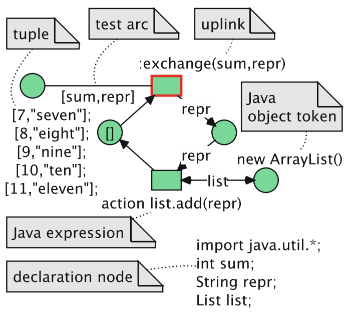
Renew is a continuously developed extensible modeling and simulation environment for Petri nets and other modeling techniques [21]. Due to its recent enhancements and extensions it has evolved into an IDE (integrated development environment) for the development of Petri net-based software [6]. Renew is focused on Reference Nets and fully supports that formalism in terms of modeling and execution with or without graphical feedback. Benefiting from its plugin architecture, Renew has been extended with various modeling techniques and formalisms over the past years. With recent efforts towards model-driven language engineering [27], Renew has become an environment for the development of modeling techniques and tools as well. The current development version supports the concurrent and coupled simulation of multiple formalisms. Renew is written in Java and available including the source code for various platforms, such as Linux, Mac OS and Windows.[1]

2.2 Reference Nets

The (Java) Reference Net formalism [20] is a high-level Petri net formalism that combines the nets-within-nets paradigm [31] with synchronous channels [7] and a Java inscription language. This formalism makes it possible to build complex systems using dynamic net hierarchies. The nets-within-nets concept is implemented using a reference semantics so that tokens can be references to nets. With the Java inscription language, it is possible to use Java objects as tokens and execute Java code during the firing of transitions. The synchronous channels enable the synchronous firing of multiple transitions distributed among multiple nets and a bidirectional exchange of information. An introduction to Reference Nets is available in the Renew manual [22], the formal definition can be found in [20] (in German). In the following we give a brief overview of the features of Reference Nets.



(a) somenet



(b) othernet

Fig. 1. Example Reference Net system showcasing selected features (adapted from [12, p. 77])

The Reference Net system depicted in Fig. 1 exhibits a selection of Reference Net constructs and focuses on those that are relevant for this contribution. It consists of two corresponding nets, *somenet* and *othernet*. The places can hold

[1] The full multi-formalism feature will be part of Renew version 2.6 and then be available at http://renew.de. The current development version can be found at http://paose.net/wiki/MultiFormalism.

black tokens, values of elementary data types or object and net references. Java types are imported and declared in the declaration node. The Reference Net formalism makes use of templates and instances analogously to classes and objects in object-oriented programming languages. Net instances are created from net templates using the `new` keyword as shown in the lower part of Fig. 1a. Transitions have a flexible inscription language featuring guards, synchronous channels, and Java expressions. Places and arcs can also hold inscriptions. Virtual places are references to other places depicted as double bordered circles, where the same semantic place may have multiple graphical figures in order to prevent long and crossing arcs.

The two highlighted transitions from *somenet* and *othernet* form a synchronous channel, which always consists of two parts: *downlink* and *uplink*. The downlink must hold a reference to the net containing the uplink. The net *somenet* in Fig. 1a holds the `:exchange(sum,repr)` channel's downlink and a reference (`net`) to the net *othernet* in Fig. 1(b), which contains the uplink. Even though the invocation of the synchronous channel is directed, due to the required reference, the information exchange is bidirectional.

The unification algorithm searches for bindings of a transition that satisfy the possible assignments of variables with respect to the declared inscriptions on transitions, arcs, and places. If both transitions participating in a synchronous channel are enabled, they can fire synchronously and exchange information in both directions. In this example the variable `sum` in *somenet* that is used in the channel `:exchange(sum,repr)` is unified with the variable `sum` from the tuple of `sum` and `repr` from *othernet*.

The mechanism of synchronous channels provides powerful features for the coupling of multiple models. As described above, a synchronous channel consists of a pair of up- and downlink. The main reason for this is an efficient implementation (with mostly polynomial complexity instead of exponential complexity) that has been described in [20]. If multiple uplinks of one net can participate in the firing of synchronized transitions, one of the possible uplinks is chosen non-deterministically. For each downlink (and its parameters) there is exactly one uplink that will be bound when a transition is fired, and both must participate since the firing of the transitions is atomic. In the following we will briefly sketch our approach to coupling multiple modeling techniques by introducing our running example, before we generalize from this idea in order to develop our conceptual approach.

2.3 Coupling Finite Automata with Reference Nets

The coupling of finite automata and Reference Nets serves as a simple example of linking multiple formalisms. We facilitate the coupling through enhancing the finite automata formalism with synchronous channels. More precisely, we develop a concept to synchronize finite automata state transitions with the firing of Reference Net transitions.

In this section we outline the coarse idea by presenting a useful example for the coupling of finite automata and Reference Nets. The general concept for the coupling and simulation of multiple formalisms is described in Sect. 3. An exemplary implementation that allows the simultaneous and synchronized execution of both – finite automata and Reference Nets – is described in Sect. 4.

One application area of multi-formalism simulation is the controlling of systems to avoid unwanted behavior such as deadlocks or security violations. To ensure orderly behavior, it is possible to use a controlling instance to restrict the possibilities of the controlled system. Ezpeleta et al. use controllers with Reference Nets in such a way [10].

We present a simple example to motivate a goal of controlling nets – avoidance of deadlocks and unwanted situations, as mentioned by Burkhard [5]. The Reference Net in Fig. 2 depicts a simple production and consumption process. The consumption (lower right part) requires at least one preceding production (upper right part) to prevent the system from running into a deadlock.

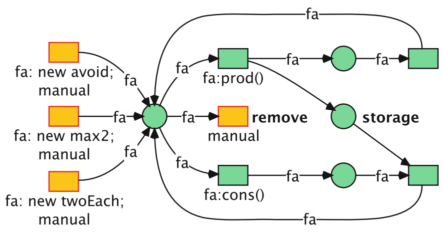


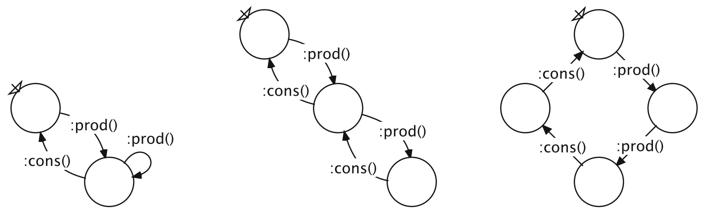
Fig. 2. Reference Net with potential deadlock

At the left hand side there are three manually fireable transitions, each of which instantiates one of the controller automata, which are shown in Fig. 3. The token that is passed through the Reference Net holds a reference to one of the controller automata. Producing (`fa:prod()`) puts a token into the storage place. Consuming is processed in two steps. In the first step the consumption process is entered (`fa:cons()`). In a second step a token from the storage is consumed. If there is none, the net is stuck in a deadlock.

To avoid a deadlock, one may use a finite automaton. Three dead-lock-a-voi-dance strategies are shown in Fig. 3. The strategy *avoid* (Fig. 3a) constrains the net to produce at least once before each consumption. The strategy *max2* (Fig. 3b) limits the number of tokens in the storage to 2, so that a consumption process takes place at least after every second production. The third strategy *twoEach* (Fig. 3c) predefines the order to two productions followed by two consumptions repeatedly. With all three strategies a deadlock is avoided.

3 Concept for Multi-formalism Simulation

In the following we elaborate on our approach to multi-formalism simulation that we sketch in Sect. 2. We generalize the idea of using finite automata to control and visualize parts of a system. On the basis of Reference Nets we present a method to link multiple formalisms through synchronized actions.



(a) avoid (b) max2 (c) twoEach

Fig. 3. Different deadlock avoidance strategies

3.1 Coupling via Synchronous Channels

In order to capture the overall structure and behavior of complex software systems, several modeling techniques are applied in combination. Coupling of models supports the modeling process in general by providing modeling techniques that exactly match the requirements of the modeling purpose. Each created model covers a distinct, yet partly overlapping perspective of the system. All models need to be integrated in a consistent way to cover the complete system.

In a common setting – e.g. when using UML – the models are, more or less, modeled in isolation. The relations to other modeling techniques and therefore between the created models are not shown explicitly. While at first sight this appears to make it easier for the modeler, this is a problem. Modelers must know how models interact with each other. Usually, this relation is established by the compiler of the models if the models can be used directly for code generation. In most modeling environments this is not the case. In consequence, the models are complemented with implementation details to facilitate their execution.

As motivated in the introduction, the transformation of models into a single target language may lead to blurred perspectives and difficulties in examining the system.

While, of course, a common execution language may work in the background, we propose to directly execute the models and to make the coupling explicit in order to retain the perspectives. The combined simulation in their original representation requires an operational semantics of the applied techniques and a mechanism for the coupling of models.

Coupling multiple techniques requires considerable conceptual and technical support. We use, in addition to the operational semantics of the modeling techniques, an execution environment that properly supports multi-formalism simulation, Renew. A drawback of the explicit coupling of multiple modeling techniques is that it involves an extension of the modeling languages.

Direct simulation is not applicable for every modeling technique. For example, the execution of a solely structural diagram seems not to be useful. In this contribution we mainly consider behavioral techniques that are discrete and state-based. This covers many of the UML behavioral diagrams and process modeling languages, such as BPMN and EPC.

3.2 Model Coupling with Graphical Feedback

In our group we have studied the necessary and sufficient solutions to implement modeling techniques within our framework(s). To extend high-level Petri nets by synchronous channels elaborate algorithms were needed. The specification, design and implementation was a highly complex task. However, on top of this a very powerful semantics for the description of other modeling techniques is available now. With previous contributions we have shown the development of new formalisms on the basis of Renew by providing operational semantics through a mapping to Petri nets [16,27]. In this contribution we propose to create formalisms that use a mapping to Reference Nets in the background in order to provide the operational semantics for the modeling technique but present the original representation to the user.

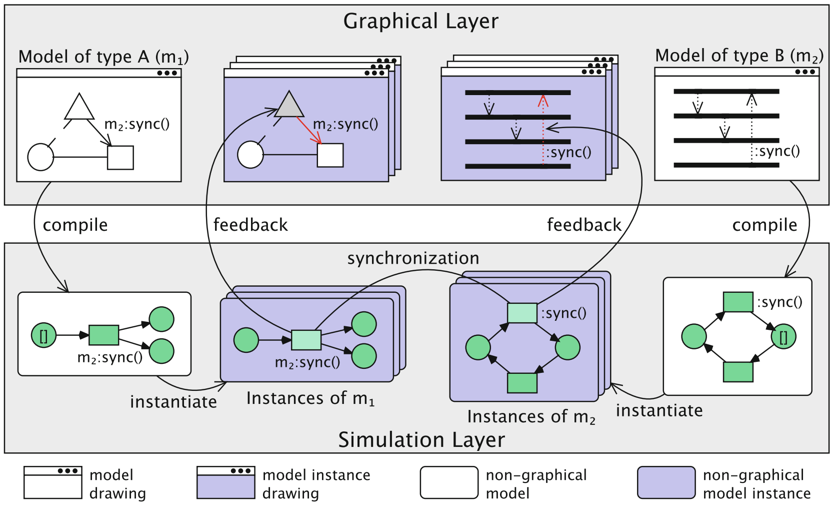


Fig. 4. Conceptual model of the model synchronization

Reference Nets are well-suited to cover multiple properties of modeling techniques, especially those of discrete, state-based behavioral techniques that we consider in this contribution. In order to simulate an arbitrary number of models concurrently, the execution language has to provide an intuitive mechanism to support the implementation of concurrency. The requirement for atomicity of activities and resource allocation emerges from the possibility of concurrent activities in a set of models. The simulated models should not affect each other, unless it is explicitly intended via synchronization. Thus, the execution language has to support strict data encapsulation for models in simulation. For process modeling languages often dynamic hierarchization is required (e.g. for subprocesses in BPMN 2.0). Due to the nets-within-nets concept, encapsulation of simulated models and dynamic hierarchization is provided naturally by Reference

Nets. Reference Nets as a Petri net formalism provide a powerful mechanism to implement concurrent behavior, resource allocation and atomicity of actions.

Figure 4 summarizes the general idea of our approach to multi-formalism modeling and execution. The upper part (Graphical Layer) of the figure contains the graphical models (model drawings) and model instance drawings that are visible to the user and permit user interaction. Model drawings are artifacts that are created from a graphical editor within Renew or may be imported from an external tool – comparable to classes in Java. The model instance drawings are instantiated drawings – comparable to objects in Java. These reflect the simulation state to the user and allow control over the simulation, e.g. by triggering a certain simulation step.

The displayed model drawings on the top-left and top-right hand side are arbitrary in a sense that they do not have a real application or semantics and serve as representative for any modeling technique a modeler may want to use. A simple example of how a concrete modeling technique is implemented is provided for communicating automata in Sect. 4. For a specific modeling technique a mapping from the constructs of the modeling technique to Reference Net constructs is needed in order to obtain an executable model. This may be compared to a code generation approach. The development of such a mapping is a part of the modeling effort, a developer of a modeling language is obliged to perform.

Graphical models are compiled into non-graphical Reference Nets, which can be instantiated and executed in the Renew simulator (Simulation Layer). Synchronization is performed on the level of the Reference Net instances (originating from the same or other modeling techniques). A mapping of the dynamic view of generated Reference Nets back to the modeling technique constructs even allows a direct feedback. The presented solution enables the simulator to pass simulation events from the Reference Net to the original model, e.g. to highlight a corresponding graphical figure (described in Sect. 4.4).

Using model transformation with Reference Nets as target formalism to provide semantics and support for coupled simulation of modeling languages comes with several benefits. Synchronous channels provide a common conceptual basis for combining multiple formalisms. Following the presented approach, all formalisms share this mechanism for communication, they are designed to be used in combination. The mapping to Reference Net constructs makes it easier to develop formalisms and to concentrate on the modeling language engineering. It supports a prototypical approach because language constructs may be designed in their Reference Net representation to be later on replaced with the constructs of the formalism in development. Users of the implemented modeling languages directly profit from the *native* feedback in comparison to generative approaches.

3.3 Limitations

Using Reference Nets as the target formalism for various modeling techniques in order to introduce coupled simulation also comes with a few limitations.

By using Petri nets as the target language, their advantages and disadvantages are inherited to the source languages. An essential feature of Petri nets is the principle of locality. While this has many advantages regarding the modeling of concurrency, it is a limitation to the implementation of global behavior. Our concept is based on the premise that the source languages only exhibit local behavior, in the sense that the constructs depend only on their direct environment. With Reference Nets as target formalism, global effects can be achieved but require encoding in data structures or extensive use of the Java inscription language.

In Reference Nets there exists no means for global synchronization. As described in Sect. 2.2, one synchronous channel, comprising up- and downlink, synchronizes exactly two transitions. Consequently, there is no broadcast communication synchronizing all transitions by default. A synchronous channel only provides the synchronized action of two elements. The synchronization of more than two transitions is accomplished by combining multiple channel inscriptions on one transition (having at most one uplink, but several downlinks). In many cases, these restrictions do not constrain the general possibilities of modeling or easy alternatives exist. For instance, when a model instance can/shall not hold references to other models, communication may be provided by a system net that holds references to all participants.

A property that is inherent in continuous systems and also in many modeling techniques is time. An extension of Reference Nets with timed expressions exists [22]. However, Renew supports the execution of these nets, but the formalism can only be simulated sequentially, and Renew's implementation of the Reference Net formalism focuses on concurrency. The presented concept is thus not well-suited for timed modeling techniques using Reference Nets as target language.

Reference Nets do not support probabilities and priorities in a natural way. Modeling these is only possible with great effort and thus not intended. Hence, it is not advisable to use the presented approach for probabilistic and prioritized modeling techniques.

In general transforming models into a single formalism has the advantage of being able to verify within a single formalism. For Reference Nets, however, there are not many verification tools available yet. The development of verification tools is one of our current research topics. We already provide a prototypical implementation for the generation of reachability graphs and simple CTL model checking tasks for restricted net formalisms.

The limited scope of variables to one transition and its connected arcs is a sensible property for Petri net formalisms because it fits the general concept of locality, which is inherent in Petri nets. For other modeling techniques this may be a limitation. In some cases, Reference Nets can be used to provide operational semantics to modeling techniques with global variables. With synchronous channels it is possible to provide global access interfaces to data objects residing in places.

4 Development of a Finite Automata Plugin

In order to demonstrate the basic concepts and tasks that have to be applied to couple models we describe the development of a Finite Automata plugin (FA plugin). Although finite automata do not add respective advantages to Petri nets, they are well-suited for the illustration of our concepts. This plugin extends Renew with the capabilities for modeling and simulating finite automata that have the ability to synchronize with Reference Nets. Here we focus on the development of the simulation and synchronization abilities.

The development consists of four main tasks. (1) Extend the finite automata modeling language, (2) develop a mapping from finite automata concepts to Reference Net concepts, (3) implement a compiler that compiles the finite automata constructs according to the previously developed mapping to Reference Net constructs and (4) implement the feedback from simulation events to a graphical representation. With the completion of these four tasks, the FA plugin for Renew provides the concurrent simulation and synchronization of finite automata in combination with Reference Net-based formalisms. The FA plugin as described in this section is part of the Renew package referenced in Sect. 2.1.

4.1 Extending the Finite Automata Modeling Language

In order to couple finite automate along with Reference Nets we extend the finite automata modeling language with Reference Net inscriptions (cf. Sect. 2.3). The direct mapping of inscriptions from state transitions to inscriptions on Reference Net transitions makes the use of synchronous channels and other Reference Net inscription types possible. Consequently, Reference Nets or other modeling techniques that provide compatible synchronous channels can synchronize with finite automata in the same way.

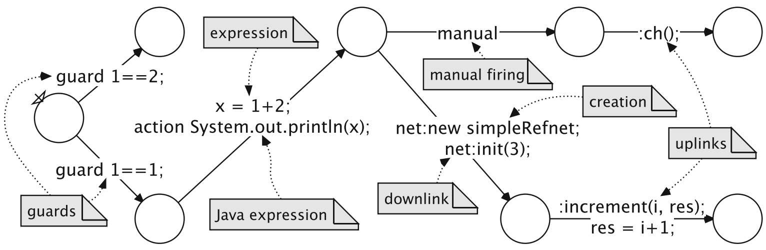


Fig. 5. Available inscription types for finite automata

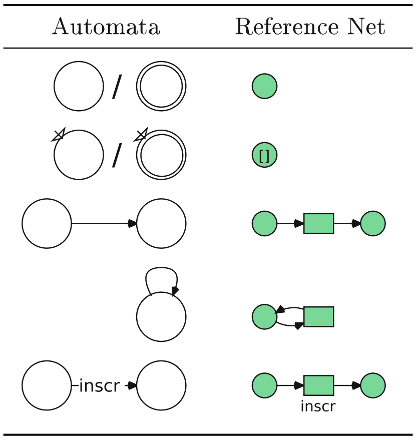
Since the inscriptions of the finite automata are inherited from Reference Nets, the syntax and the semantics are similar (see [22, p. 48 ff.] for details). In general this creates multiple possibilities to inscribe state transitions of automata as depicted in Fig. 5. Due to the restricted expressiveness of finite automata, some of these inscription types are only of limited use. An important consideration is that variables in finite automata can not be stored and only accessed arc-locally (see Sect. 3.3). As a result, the applicability of downlinks is limited. Additionally, the application context affects which inscription types are reasonable to use. For the following examples presented in this contribution we solely use the manual and uplink inscriptions. These can be used analogously to the Reference Nets (see Sect. 2.2).

4.2 Mapping Finite Automata Concepts to Reference Net Concepts

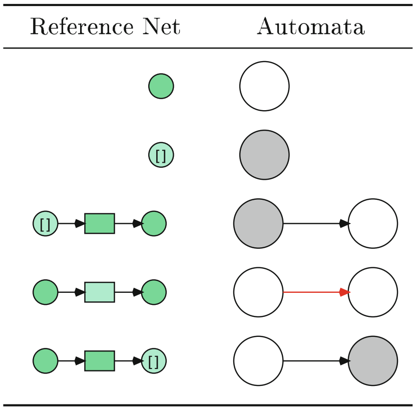
The second task includes the provision of semantics via a mapping to Reference Nets in order to exploit their synchronization features.

Table 1. Mapping of finite automata and Reference Net constructs

(a) Static view



(b) Dynamic view



The mapping of finite automata to Reference Nets is straightforward as displayed in Table 1a. A finite automaton's state is mapped to a Reference Net's place. We do not distinguish between regular and end states because we are more interested in the dynamic behavior than in the investigation of the formal language an automaton generates. Start states are mapped to places that are initially marked with a black token (represented as `[]`). A state transition between two states is mapped to a transition that is connected to the places representing the corresponding states. This also holds for the special case of a self-loop. The inscriptions of the state transitions are mapped to inscriptions of net transitions. Thus, for example, a synchronous channel that is attached to a finite automata state transition is attached to the corresponding Reference Net transition (holds for up- and downlinks).

In general the provision of a mapping to Reference Nets is the difficult part of the development process. For languages that already have a formal semantics, the language developer has to ensure that the Petri net-based semantics equals the original semantics but this task is out of scope in this contribution. In many cases the source language does not have a formal semantics at all (e.g. for domain specific modeling languages) and the semantics is defined with the transformation.

4.3 Implement Compiler

For the implementation of a compiler for FA models within Renew we benefit from its plugin architecture, which allows to plug in additional formalisms without the requirement to change the simulator code. For the simulation in Renew, a graphical model is first converted into an abstract model without the graphical information. This is the input for a specific formalism compiler that translates the abstract models into executable models. This task includes the parsing of inscriptions. Following the approach presented in this paper the transformation into Reference Nets is an additional task.

The compiler for the FA formalism processes the FA model construct by construct and maps them to the corresponding Reference Net elements according to the presented mapping (cf. Table 1). In this step the extension of the modeling language comes into play because the constructs from the language extension need to be considered as well. For the extended FA models the inscriptions on state transitions are directly attached to the resulting net transitions. The application of the mapping in this step is hard-coded in the compiler class. However, we are working on a model-based generic compiler that allows the user to provide the mapping in the form of net templates [28]. The compilation of the target elements is then delegated to the Reference Net compiler.

Another important step in this compilation task is the management of the identifiers of the FA constructs and their executable Reference Net counterparts. This is important for the implementation of the simulation feedback. The graphical simulation environment of Renew uses the identifiers of net elements to find the corresponding graphical constructs. Therefore, the IDs of the target Reference Net elements have to match the source FA constructs. In general, a modeling construct may be mapped to multiple Reference Net constructs. For this purpose the element IDs are complemented with additional group IDs to realize a bijective mapping.

4.4 Implement Simulation Feedback

In addition to the compiler that realizes the static mapping of FA and Reference Net constructs, we have to manually implement the visual representation of the simulation state. This gives the user graphical feedback in the original representation of the modeling technique (i.e. the state of the simulation of the Reference Net at runtime has to be mapped back on the automaton).

To provide the simulation feedback we have to implement the classes for the graphical model instances. These classes define their representation and the changes in reaction to simulation events. Renew has a simple mechanism to provide graphical representation classes for instance models. The net elements in the simulation are registered to the graphical instance components as listeners, which is possible due to the bijective mapping via the IDs. With this mechanism the representations of the static constructs are used and a generic highlighting mechanism can change its colors to represent activity in the corresponding construct. The language developer can specify in the instance representation classes on which simulation event they should change the representation using the generic highlighting mechanism. Since one construct from the modeling language may be mapped to multiple net elements (places and transitions) the developer has to decide which events the construct should reflect. For the FA tool the reflection of the simulation events is depicted in Table 1b. FA states reflect the marking of the corresponding places resulting in a gray filling and FA state transitions are highlighted during the firing of the respective net transitions with red color. The specific form of visualization and the color mapping are automatically derived by Renew's graphical simulator. We have been working on solutions for the model-based specification of visual properties in simulated models [28], but this goes beyond the scope of this contribution.

Additionally, the possibility to interact with the simulation has to be implemented. The classes for the graphical elements instances may contain methods to provide support for firing of transitions as a reaction to e.g. a mouse click. Therefore, the developer has to define which transition of the Reference Net construct shall fire.

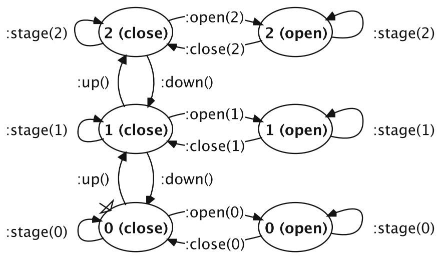
Due to the implementation of the simulation feedback and interaction as presented here, the modeling and simulation tool behaves as if it was a native implementation for the modeling language.

5 Elevator Application

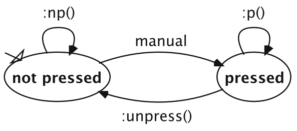
In this section we present a more complex example for the use of multi-formalism execution. The presented system is a model of an elevator, which is partly created as automaton and partly as Reference Net. Subsequently, we discuss exchanging the Reference Net part with an activity diagram.

The elevator can move up- and downwards between three floors and open its door on each of the floors. It is possible to call the elevator on a floor by pressing the button on the respective floor. For reasons of simplicity, we assume that pressing the button on one floor has the same effect as pressing the button for the destination floor from inside the elevator. Thus, we do not model the touch panel inside the elevator explicitly. In this scenario the status of the elevator and the status of the button on each floor is simple. These are systems without any concurrency and with a defined state. Therefore, these components are modeled as finite automata. The complex part in this scenario is the control mechanism that ensures a reasonable serving strategy. Thus, a more expressive technique is required to model this part. We examine the utilization of Reference Nets and activity diagrams.

The automata models depicted in Fig. 6 have multiple state transitions attached with uplink inscriptions in order to be synchronized with the Refer-



(a) Elevator automaton



(b) Floor status automaton

Fig. 6. Automata models of the elevator example

ence Net. With the loop at each state, it is possible to implement state dependent behavior (e.g. opening the door is only possible when the button on the respective floor is pressed). State transitions may occur nondeterministically (no inscription), due to manual interaction by the user (`manual` inscription) or triggered through the call of a synchronous channel (uplink inscription) initiated by the controlling Reference Net. Pressing the button in a floor is the only possible interaction with the system. Therefore, the transition from *pressed* to *not pressed* in Fig. 6b is inscribed with the `manual` keyword, which has the effect that the firing has to be initiated manually by clicking the transition and the simulator does not fire this transition automatically.

5.1 Elevator Control Modeled as Reference Net

The Reference Net in Fig. 7 implements an elevator control that serves a floor when the elevator is requested by a pressed button and prefers to keep the movement direction. At the top of the net, instances of the automaton models are created (one instance of elevator, three instances of floor).

The actual elevator control is divided vertically into the three floors. Each floor consists of three or four control places (green places) representing the elevator on the respective floor on its way downwards (leftmost place), on its way upwards (rightmost place), or with an open door (places in the center). Changes to the elevator and floor models or state queries to these models require access to the elevator and floors places. This is achieved by using virtual places (virtual copies of places depicted as circles with a double border, see Sect. 2.2).

The elevator is initialized with closed doors in the ground floor. With Transition *o*, in the bottom, the door is opened initially. Transition *cu0* closes the door of the elevator (`elevator:close()`), which can then start its way upwards. This is only possible, when the elevator was requested on one of the higher floors (i.e. the button is pressed on one of the higher floors, `fn:p(); guard n > 0`). If the door is closed, the elevator may move upwards one floor with Transition *u1* (`elevator:up()`) if the elevator was requested from a higher floor (`fn:p(); guard n > 0`) and it was not requested on the same floor (e.g. somebody wants to get in, `f0:np()`).

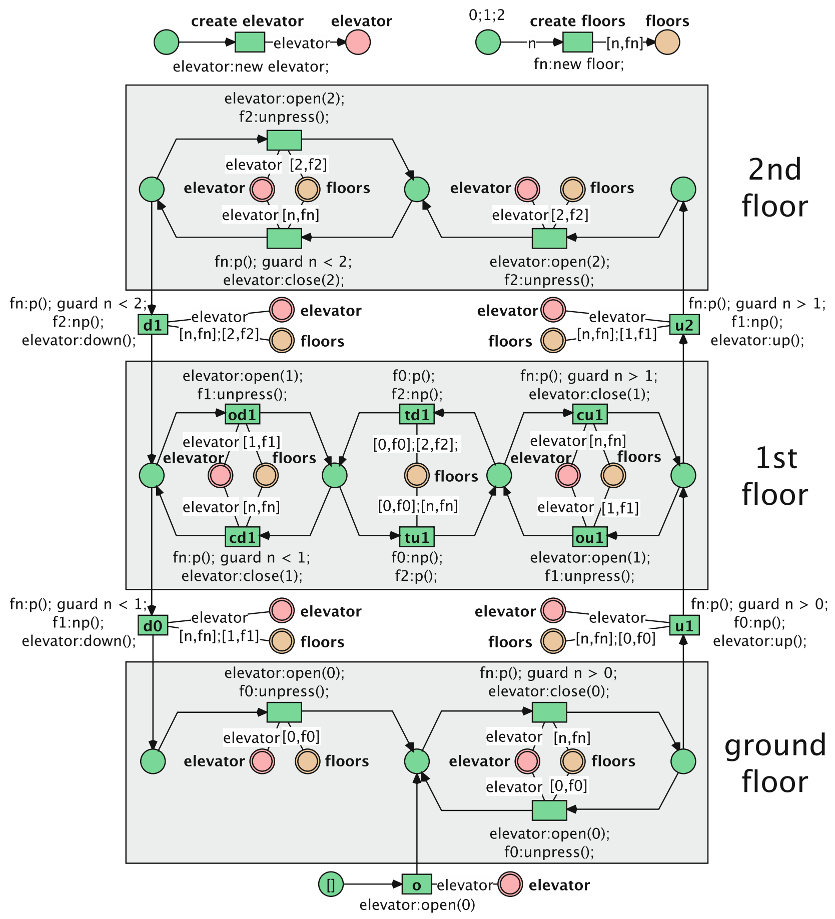


Fig. 7. Reference Net for controlling the elevator

With this mechanism, the elevator can move upwards until it reaches a floor where the button is pressed (e.g. on the second floor). There, it has to open the door (`elevator:open(2)`) and the request for the floor is reset (`f2:unpress()`).

On the first floor the elevator has two places representing an open door to distinguish the two directions. The elevator is allowed to change its direction (with Transitions *td1*, *tu1*) if there is no request in the current direction (e.g. `f2:np()`) and a request in the other direction (`f0:p()`). Analogously to the right hand side of the elevator control, the left side implements the controlling of the elevator moving downwards.

The Reference Net displayed in Fig. 7 models a low-level perspective on the elevator control and provides an implementation that is ready to be executed in the Renew simulation environment. In the following Section we present an abstraction of this perspective that focuses on the activities and decisions in the model and emphasizes the control flow.

5.2 Elevator Control Modeled as Extended Activity Diagram

The provision of another abstraction for such a complex model as the elevator control may improve the readability, and the expressiveness gained by additional constructs from the applied language may decrease the modeling effort. In the previous section we utilize finite automata in order to encapsulate the state of the elevator system. UML activity diagrams focus on the activities in a process and provide useful constructs for modeling an elevator control. Thus, we exchange the modeling technique for controlling the elevator, the Reference Net (depicted in Fig. 7), with an activity diagram. Analogously to the finite automata, we extend activity diagrams by synchronous channels and Reference Net-specific inscriptions.

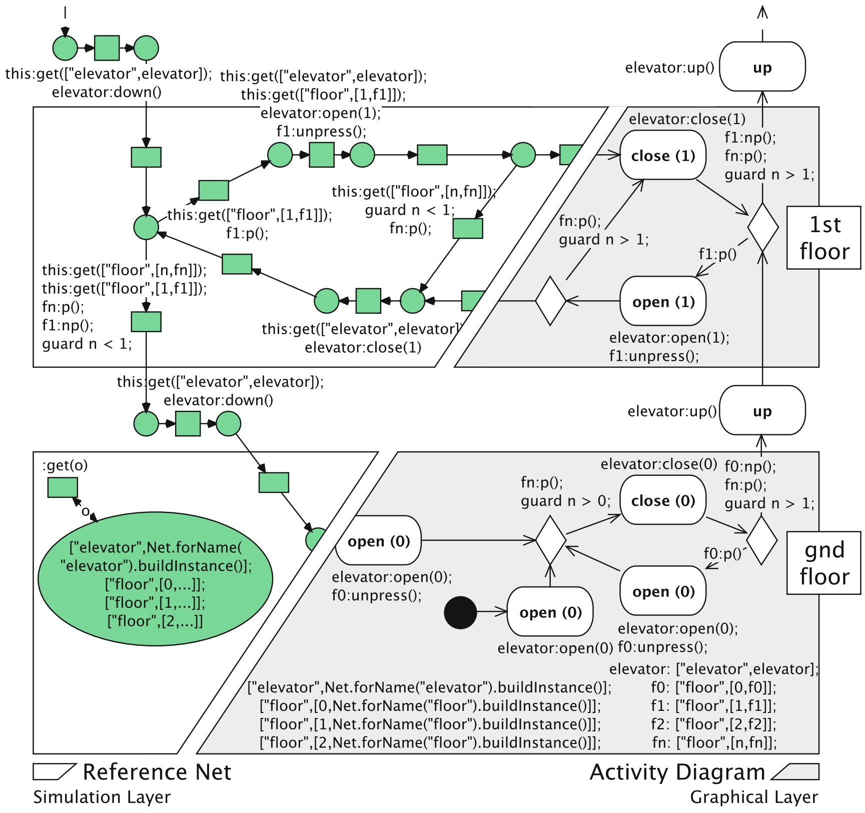

this:get(["elevator",elevator]); elevator:down()
this:get(["elevator",elevator]); this:get(["floor",[1,f1]]); elevator:open(1); f1:unpress();
this:get(["floor",[1,f1]]); f1:p();
this:get(["floor",[n,fn]]); guard n < 1; fn:p();
this:get(["floor",[n,fn]]); this:get(["floor",[1,f1]]); fn:p(); f1:np(); guard n < 1;
this:get(["elevator",elevator]); elevator:close(1)
elevator:up()
up
elevator:close(1)
close (1)
f1:np(); fn:p(); guard n > 1;
fn:p(); guard n > 1;
f1:p()
1st floor
open (1)
elevator:open(1); f1:unpress();
this:get(["elevator",elevator]); elevator:down()
:get(o)
o
["elevator",Net.forName("elevator").buildInstance()]; ["floor",[0,...]]; ["floor",[1,...]]; ["floor",[2,...]]
elevator:close(0)
close (0)
f0:np(); fn:p(); guard n > 1;
fn:p(); guard n > 0;
open (0)
elevator:open(0); f0:unpress();
f0:p()
gnd floor
open (0)
elevator:open(0)
open (0)
elevator:open(0); f0:unpress();
elevator: ["elevator",elevator]; f0: ["floor",[0,f0]]; f1: ["floor",[1,f1]]; f2: ["floor",[2,f2]]; fn: ["floor",[n,fn]];
["elevator",Net.forName("elevator").buildInstance()]; ["floor",[0,Net.forName("floor").buildInstance()]]; ["floor",[1,Net.forName("floor").buildInstance()]]; ["floor",[2,Net.forName("floor").buildInstance()]];
Reference Net
Simulation Layer
Activity Diagram
Graphical Layer

Fig. 8. Excerpt of the extended activity diagram for controlling the elevator (right side) and the generated Reference Net implementation (left side).

Figure 8 shows a combined drawing of the graphical and the simulation layer of the activity diagram that is an alternative model to the Reference Net elevator control (cf. Fig. 7). Depicted to the right is an excerpt of the elevator model as activity diagram, which covers a part of the ground and first floor. The left side displays the generated Reference Net that corresponds to the left side of the activity diagram, which is not shown. Note that the partly displayed Reference Net model is not exactly the same as the one from Fig. 7, but the behavior is similar in the sense that it implements the same serving strategy. This net is composed of components that originate from a direct mapping of activity diagram constructs, comparable to the semantic mapping we present for finite automata in Table 1. The complete mapping is not displayed here because we do not want to focus on Petri net semantics for specific modeling languages in this contribution. The mapping for the used constructs can be deduced from the generated Reference Net in Fig. 8.

Utilizing activity diagrams as an alternative modeling technique comes with several effects. The elevator control modeled as Reference Net (cf. Fig. 7) contains multiple transitions, which serve different functions. They may act as activities or decisions. For example, the call `f0:p()` is a query for a specific state, whereas the call `elevator:up()` is an actual activity. These functions become visible in the Reference Net only by looking at the context. The exchange of the Reference Net with an activity diagram emphasizes the separation of activities and decisions.

For the activity diagram model we make use of a global storage for referenced models. The model references are stored in a single place and a synchronous channel provides global access. This feature requires the specification of the elements to be initialized and the interfaces to access the elements in the activity diagram model. These declarations can be found in the lower part of Fig. 8. At the left side of the activity diagram part the used models are declared as key value tuples in a data declaration node. The declaration of the elevator automaton for example is a tuple `["elevator",Net.forName("elevator").buildInstance()]`, where the first element is the identifier to access the model and the second element is the code for the creation of a net instance. The information from the data declaration node is used to generate a single place containing all the objects and a channel to access these objects into the Reference Net (`:get(o)`). Additionally an interface to the data/models has to be specified which happens in the variable declaration node in the bottom-right corner. The entries in this node correspond to variables that are available in the activity diagram. The variable `f0` for example is declared with `f0: ["floor",[0,f0]]`, where the part before the colon corresponds to the variable name and the part behind is the pattern corresponding to the data declaration. Based on this pattern, the effective call to the `:get(o)` channel in the Reference Net can be generated into every transition inscription where the variable is used. This type of data interface declaration facilitates construction such as `this:get(["floor",[n,fn]]); guard n > 1` to reference any floor above the first.

Beyond the improved readability, the UML activity diagram formalism can provide constructs to reduce modeling effort for the extension of the elevator system, for example to implement a more elaborated control strategy, cover additional aspects of the system or handle multiple elevators. *Time event actions* may be used to implement waiting times of the elevator before it opens or closes doors, *exception handler* elements can be used to model failures in the system (e.g. an elevator moving up or down with an open door) and *swimlanes* may enhance the separation of multiple floors or elevators.

6 Related Work

In this section we briefly relate our proposal to work in the area of multi-formalism modeling and execution.

Zeigler [32] proposes the multifaceted modeling methodology, which is an approach to simulation and modeling by integrating multiple models. The hereby used *Discrete Event System Specification* (DEVS) formalism is capable of constructing coupled models that are composed of atomic DEVS models. In his work an abstract simulation concept for the DEVS formalism was developed. The concept includes the coupling of several simulators for each component in a system of systems to facilitate simulation using a global coordinator for synchronization. Our approach is different from his as we attach importance to concurrent simulation, in particular to true concurrency of Renew.

Lara et al. [23] introduce a tool that supports the combined use of multi-formalism modeling and meta-modeling, called AToM^3. Due to the definition of *graph grammar models*, formalisms can be transformed into an appropriate formalism for which simulation is already supported. They suggest the DEVS formalism as the central modeling formalism that can be universally used for simulation purposes. AToM^3 also supports code generation, a meta-modeling layer that can be used to model formalisms in a graphical manner, and the possibility to transform models by preserving the behavior [2]. In contrast to our approach, Lara et al. focus on providing meta-modeling and model-transformation features. The transformed models have to be simulated in an external environment.

Möbius is a tool for modeling and simulating systems composed of multiple formalisms. The project focuses on extensibility by new formalisms and solvers, which is demonstrated in [8]. An *abstract functional interface* is implemented, which transforms models to framework components to allow for addition of formalisms and interaction between models. Their approach differs from ours in the way of having an overall system state. The Möbius tool enables selective sharing of model states, so that solvers (i.e. simulators) are able to access them.

One practical implementation of a multi-formalism modeling and simulation concept was done by the GEMOC Initiative [13]. This initiative's vision is to advance the research on the coordinated use of modeling languages. They recognized a problem in the unavailability of a generic runtime service for multiple modeling languages. GEMOC Studio is proposed as a tool to create meta-models for both the representation and the operational semantics of modeling languages. Created models of multiple languages can then be executed in coordination, while being debugged and graphically visualized. They use the *Behavioral Coordination Operator Language* (BCOoL [24]) to explicitly specify the coordination patterns between heterogeneous languages. The used execution engine operates as a coordinator of multiple language-specific engines.

Frank [11] proposes a method for multi-perspective modeling that extends the classical approach (of e.g. UML) to conceptual modeling by including the organizational environment. He is also interested in tool support and states the following requirement we share: "A tool environment for enterprise modeling should allow for creating multi-language diagrams, i.e., diagrams that integrate diagrams of models that were created with different DSML" [11, S. 946]. The linking of techniques he proposes is established on the level of a meta-model, but the method lacks an environment to properly support the execution. "However, there are other paradigms that come with specific advantages, too. [...] Languages used for creating simulation models would allow for supplementing enterprise models with simulation features. Petri nets provide mature support for process analysis and automation" [11, S. 960]. We find the combination of such approaches to modeling language engineering with the provision of operational semantics for a dedicated execution environment most promising.

Jeusfeld [18] proposes the linking of multiple perspectives through declarative constraints in the context of meta-modeling domain-specific languages for the ADOxx platform. He distinguishes the relation between model and external environment from the internal model validity and focuses on the latter. The constraint language is used to define a Petri net firing rule and to sketch a firing rule of BPMN constructs.

There are other researchers who have tried to combine various formalisms with Petri nets. The set of all firings of a Petri net can be considered to be a language. The research of automata and Petri nets as language descriptions has led to the control of Petri nets by (finite) automata [5]. The results suggest that controlling Petri nets through finite automata can be beneficial. A combination of finite automata and high-level Petri nets can be seen as a vertical composition, as both techniques basically provide the behavioral modeling perspective according to Krogstie [19], just on another level of abstraction.

While first the idea of language intersections were of interest, the idea of a *controller* was used in the context of application modeling [30]. An elevator system was modeled, however, just the control was addressed and no other modeling techniques were applied. In other research, flexible manufacturing systems (FMS) were used to demonstrate central aspects of control theory (cf. [14] for discrete event system discussion). Most of the authors' work concentrates on techniques that provide one specific modeling perspective. A production system can be controlled by a simple model to ensure that no deadlocks can occur [9,15].

Petri nets are often used as target language for approaches that transform abstract modeling languages into a single formalism. Often the motivation is to be able to formally verify the generated Petri nets. Activity diagrams in particular are covered by multiple authors (e.g. [1,17]).

Overall our approach presented here allows to provide modelers with the coupling and simulation of their favorite modeling techniques. The complexity can be reduced by using an appropriate modeling technique, like finite automata, workflows or other modeling techniques like BPMN, eEPCs, activity diagrams, etc. However, each formalism needs to be connected via the synchronous channels to be executed within our environment. Systems complying to our interfaces could also mimic our approach if sufficient support for the execution of modeling techniques is available, for example with CO-OPN/2 [3] or the Zero-safe net formalism [4]. Our illustrations of the principle usage in Sects. 4 and 5 can be seen as proof of concepts.

7 Conclusion

In this contribution we conceptually present how various formalisms can be used together not only for modeling, but also for simulation while preserving their original representation.

In our opinion, providing a solution for the simulation of these formalisms can increase the benefit of using multiple formalisms for the modeling of complex systems. Instead of transforming the models of multiple formalisms into one single formalism, we provide simulation feedback in the original representation of the models. Therefore, the various formalisms are mapped to Reference Nets and the simulation events are returned to the original representation of the model (cf. challenge (2) in Sect. 1).

We argue that the synchronization and data exchange between models of various formalisms should not be achieved through the development of several individual coupling mechanisms. Instead, we are of the opinion, that this should be achieved through *one* coupling mechanism.

We propose the generic coupling of models using the synchronization mechanism from Reference Nets: synchronous channels (cf. challenge (1) in Sect. 1). Synchronous channels facilitate the synchronization of several models and the bidirectional exchange of data – regardless of whether the models are of various or the same formalism.

As a proof-of-concept, we practically demonstrate how this approach can be implemented for the coupling of finite automata and Reference Nets as well as the coupling of finite automata and activity diagrams. The latter results in a system model in which Reference Nets are no longer utilized to model the system, but rather are only applied for the actual implementation.

Considering the example of activity diagrams, we present a possibility to support techniques with data and global access to this data. In the future, we will examine how to support the development of techniques through meta-modeling and to provide a close adaption to the simulation environment of Renew, i.e. to model the techniques themselves, the drawing tools and their operational semantics [27,28]. In this context we develop a Renew plugin that integrates the concept of dynamic hierarchies from Reference Nets into the multi-formalism approach to provide the modeling and simulation of statecharts.

References

1. Agarwal, B.: Transformation of UML activity diagrams into Petri nets for verification purposes. Int. J. Eng. Comput. Sci. **2**(3), 798–805 (2013)
2. AToM^3 website. http://atom3.cs.mcgill.ca. Accessed 19 Jan 2017
3. Biberstein, O., Buchs, D., Guelfi, N.: Object-oriented nets with algebraic specifications: the CO-OPN/2 formalism. In: Agha, G.A., De Cindio, F., Rozenberg, G. (eds.) Concurrent Object-Oriented Programming and Petri Nets. LNCS, vol. 2001, pp. 73–130. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45397-0_3
4. Bruni, R., Montanari, U.: Zero-safe nets: the individual token approach. In: Presicce, F.P. (ed.) WADT 1997. LNCS, vol. 1376, pp. 122–140. Springer, Heidelberg (1998). https://doi.org/10.1007/3-540-64299-4_30
5. Burkhard, H.D.: Control of Petri nets by finite automata. Annales Societatis Mathematicae Polonae Series IV: Fundamenta Informaticae **VI.2**, 185–215 (1983)
6. Cabac, L., Haustermann, M., Mosteller, D.: Renew 2.5 – towards a comprehensive integrated development environment for Petri Net-based applications. In: Kordon, F., Moldt, D. (eds.) PETRI NETS 2016. LNCS, vol. 9698, pp. 101–112. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-39086-4_7
7. Christensen, S., Damgaard Hansen, N.: Coloured Petri Nets extended with channels for synchronous communication. In: Valette, R. (ed.) ICATPN 1994. LNCS, vol. 815, pp. 159–178. Springer, Heidelberg (1994). https://doi.org/10.1007/3-540-58152-9_10
8. Courtney, T., Gaonkar, S., Keefe, K., Rozier, E.W.D., Sanders, W.H.: Möbius 2.3: an extensible tool for dependability, security, and performance evaluation of large and complex system models. In: 2009 IEEE/IFIP International Conference on Dependable Systems Networks, pp. 353–358, June 2009
9. Ezpeleta, J., Colom, J.M., Martínez, J.: A Petri net based deadlock prevention policy for flexible manufacturing systems. IEEE Trans. Robot. Autom. **11**(2), 173–184 (1995)
10. Ezpeleta, J., Moldt, D.: A proposal for flexible testing of deadlock control strategies in resource allocation systems. In: Pahlavani, Z. (ed.) Proceedings of International Conference on Computational Intelligence for Modelling Control and Automation, Vienna, Austria, 12–14 February (2003)
11. Frank, U.: Multi-perspective enterprise modeling: foundational concepts, prospects and future research challenges. Softw. Syst. Model. **13**(3), 941–962 (2014)
12. Friedrich, M., Moldt, D.: Introducing refactoring for reference nets. In: Cabac, L., Kristensen, L.M., Rölke, H. (eds.) Proceedings of the Petri Nets and Software Engineering. International Workshop, PNSE 2016, Toruń, Poland, 20–21 June 2016. CEUR Workshop Proceedings, vol. 1591, pp. 76–92. CEUR-WS.org (2016)
13. GEMOC Initiative website. http://gemoc.org/index.html. Accessed 02 Apr 2017
14. Giua, A., Seatzu, C.: Petri nets for the control of discrete event systems. Softw. Syst. Model. **14**(2), 693–701 (2015)
15. Hu, H., Liu, Y., Zhou, M.: Maximally permissive distributed control of large scale automated manufacturing systems modeled with Petri nets. IEEE Trans. Control Syst. Technol. **23**(5), 2026–2034 (2015)
16. Jacob, T., Kummer, O., Moldt, D., Ultes-Nitsche, U.: Implementation of workflow systems using reference nets - security and operability aspects. In: Jensen, K. (ed.) Fourth Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools. Department of Computer Science, University of Aarhus, August 2002

17. Jamal, M., Zafar, N.A.: Transformation of activity diagram into coloured Petri nets using weighted directed graph. In: International Conference on Frontiers of Information Technology, FIT 2016, Islamabad, Pakistan, 19–21 December 2016, pp. 181–186. IEEE Computer Society (2016)
18. Jeusfeld, M.A.: SemCheck: checking constraints for multi-perspective modeling languages. Domain-Specific Conceptual Modeling, pp. 31–53. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-39417-6_2
19. Krogstie, J.: Perspectives to Process Modeling. Studies in Computational Intelligence, vol. 444, pp. 1–39. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-28409-0_1
20. Kummer, O.: Referenznetze. Logos Verlag, Berlin (2002)
21. Kummer, O., Wienberg, F., Duvigneau, M., Cabac, L., Haustermann, M., Mosteller, D.: Renew - the Reference Net Workshop, June 2016. http://www.renew.de/. Release 2.5
22. Kummer, O., Wienberg, F., Duvigneau, M., Cabac, L., Haustermann, M., Mosteller, D.: Renew - User Guide (Release 2.5). Faculty of Informatics, Theoretical Foundations Group, University of Hamburg, Hamburg, June 2016
23. de Lara, J., Vangheluwe, H.: AToM^3: a tool for multi-formalism and meta-modelling. In: Kutsche, R.-D., Weber, H. (eds.) FASE 2002. LNCS, vol. 2306, pp. 174–188. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45923-5_12
24. Larsen, V., Ezequiel, M.: BCOol: The Behavioral Coordination Operator Language. Theses, Université de Nice Sophia Antipolis, April 2016
25. Milner, R.: Elements of interaction - turing award lecture. Commun. ACM **36**(1), 78–89 (1993)
26. Möller, P., Haustermann, M., Mosteller, D., Schmitz, D.: Simulating multiple formalisms concurrently based on reference nets. In: Moldt, D., Cabac, L., Rölke, H. (eds.) Proceedings of the Petri Nets and Software Engineering. International Workshop, PNSE 2017, Zaragoza, Spain, 25–26 June 2017. CEUR Workshop Proceedings, vol. 1846, pp. 137–156. CEUR-WS.org (2017)
27. Mosteller, D., Cabac, L., Haustermann, M.: Integrating Petri net semantics in a model-driven approach: the renew meta-modeling and transformation framework. In: Koutny, M., Desel, J., Kleijn, J. (eds.) Transactions on Petri Nets and Other Models of Concurrency XI. LNCS, vol. 9930, pp. 92–113. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53401-4_5
28. Mosteller, D., Haustermann, M., Moldt, D., Schmitz, D.: Graphical simulation feedback in Petri net-based domain-specific languages within a meta-modeling environment. In: Moldt, D., Kindler, E., Rölke, H. (eds.) Proceedings of the Petri Nets and Software Engineering. International Workshop, PNSE 2018, Bratislava, Slovakia, 25–26 June 2018. CEUR Workshop Proceedings, vol. 2138, pp. 57–76. CEUR-WS.org (2018). http://ceur-ws.org/Vol-2138
29. Petri, C.A.: Kommunikation mit Automaten. Dissertation, Schriften des IIM 2, Rheinisch-Westfälisches Institut für Instrumentelle Mathematik an der Universität Bonn, Bonn (1962)
30. Reisig, W.: Embedded system description using Petri nets. In: Kündig, A., Bührer, R.E., Dähler, J. (eds.) Embedded Systems. LNCS, vol. 284, pp. 18–62. Springer, Heidelberg (1987). https://doi.org/10.1007/BFb0016346
31. Valk, R.: Petri nets as token objects. In: Desel, J., Silva, M. (eds.) ICATPN 1998. LNCS, vol. 1420, pp. 1–24. Springer, Heidelberg (1998). https://doi.org/10.1007/3-540-69108-1_1
32. Zeigler, B.P.: Multifacetted Modelling and Discrete Event Simulation. Academic Press Professional Inc., San Diego (1984)

Complexity Aspects of Web Services Composition

Karima Ennaoui(✉), Lhouari Nourine, and Farouk Toumani

LIMOS, CNRS, Universite Clermont Auvergne, Clermont-Ferrand, France
ennaoui@isima.fr

Abstract. The web service composition problem can be stated as follows: given a finite state machine M, representing a service business protocol, and a set of finite state machines $\mathcal{R}$, representing the business protocols of existing services, the question is to check whether there is a simulation relation between M and the shuffle product closure of $\mathcal{R}$.

This paper studies the impact of several parameters on the complexity of this problem. We show that the problem is *Exptime-complete* if we bound either: *(i)* the number of instances of services in $\mathcal{R}$ that can be used in a composition, or *(ii)* the number of instances of services in $\mathcal{R}$ that can be used in parallel, or *(iii)* the number of the so-called hybrid states in the finite state machines of $\mathcal{R}$ by 0, 1 or 2. The problem is still open for 3 hybrid states.

1 Introduction

Web Services [2] is a new computing paradigm that tends to become a technology of choice to facilitate interoperation among autonomous and distributed applications. The UDDI consortium defines Web services as *self-contained, modular business applications that have open, Internet-oriented, standards-based interfaces.* Several models have been proposed in the literature to describe different facets of services. In particular, the importance of specifying external behaviour of services, also called service business protocols, has been highlighted in several research works [3,5,8]. Through literature, different models have been used to represent web service business protocols. The Finite States Machines (FSM) formalism is widely adopted in this context to model *statefull* applications exposed as web services where states represent the different phases that a service may go through while transitions represent "abstract" activities that a service can perform [3,5,8].

We consider in this paper the problem of Web Service Composition (WSC). This problem arises from the situation where none of the existing services can provide a requested functionality. In this case, the idea is to find out, algorithmically, if the target functionality could be composed out of the existing services (components repository). This automatic approach of composition simplifies the development of software by reusing existing components and offers capabilities to customize complex systems built on the fly [11].

© Springer-Verlag GmbH Germany, part of Springer Nature 2018
M. Koutny et al. (Eds.): ToPNoC XIII, LNCS 11090, pp. 116–135, 2018.
https://doi.org/10.1007/978-3-662-58381-4_6

We focus more particularly on a specific instance of WSC, namely the (business) protocol synthesis problem, which can be stated as follows: given a set of business protocols of available services and given a business protocol of a target service, is it possible to synthesize automatically a mediator that *implements* the target service using the existing ones? We find this case interesting for two reasons. First, such description provides developers with all information necessary to write clients that can correctly interact with a given service or with a set of services [4]. Second, business protocols can be described by the means of FSMs and it is much simpler than some richer frameworks where WSC is proven undecidable [6].

[15] shows that in this context, the WSC problem can then be formalized as the problem of deciding whether there exists a simulation relation between the target protocol and the shuffle (or asynchronous product) of the available ones. Another related problem discussed in [15], is the bisimulation relation between FSMs. It is, however, less interesting in the context of WSC, since bisimulation requires that the target service should be able to admit all possible branchings in the composing services, which is too complex and unnecessary in this case.

The results in [15] are however based on the implicit assumption that at most one *instance* of each available service can be used in a composition. This setting has been extended in [11] to the case where the number of instances that can be used in a composition is unbounded. WSC is formalized in this latter case as a simulation problem between an FSM and an infinite state machine, called Product Closure State Machine (PCSM), that is able to compute the shuffle closure of an FSM.

Shuffle product of FSMs (and PCSM) is a subclass of Basic Parallel Processes (BPP), the class of communication free petri nets: every transition has at most one input place. Simulation of FSM by BPP was proven Expspace-hard by Lasota [14] and 2-Exptime-hard in [10].

Complexity analysis of WSC was first considered by Musholl *et al.* [15], under the aforementioned implicit assumption, where it is shown Exptime-Complete. In case of unbounded instances, the WSC problem has been proved decidable with an Ackermanian function as upper bound in [11]. The proof of [11] is based on Dickson's lemma, and hence cannot be exploited to derive tighter upper bounds. An Expspace-hard lower bound is given by Lasota [14].

One of the complexity sources derived from analyzing the algorithm given in [11] is related to the presence of the so-called hybrid states (final states with outgoing transitions and correspond to unbounded places in Petri net terminology) in the input. We finish up in this article the work in [11] and prove that if such hybrid states do not appear in the FSMs simulating the target then the problem is Exptime-complete. Since such hybrid states can appear in a FSM describing a web service's business protocol (example in Fig. 1), then it is also interesting to see the impact of their existence on the WSC problem complexity.

We investigate as well in this paper additional parameters related to bounded web services composition: the number of instances available of each of the sim-

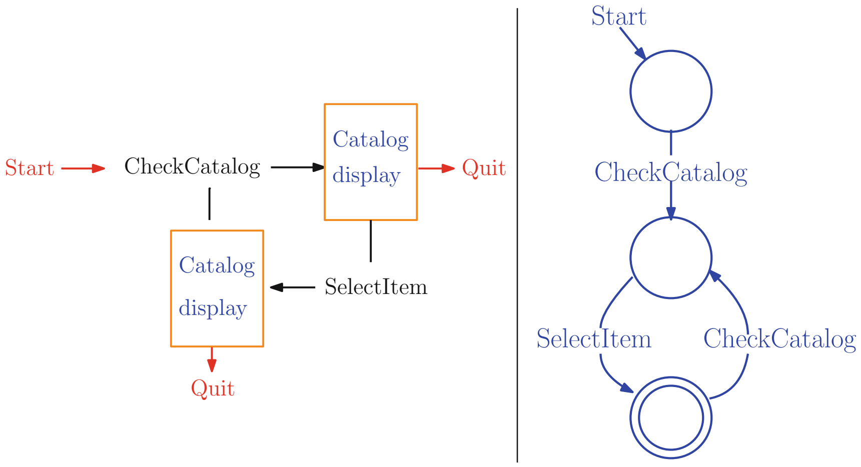


Fig. 1. An informal description of the behavior of a service managing a catalog (left) and its describing FSM (right).

ulating FSMs and the number of these instances allowed, at most, to be used in the simulation simultaneously.

Formally, let us consider as inputs a FSM M (the target protocol) and a set of FSMs $\mathcal{R}$ (the protocols of the available services), we study the complexity of the following problems:

1. $WSC(M, \mathcal{R})$: The problem of composing M using an unbounded number of instances of $\mathcal{R}$.
2. $BC(M, \mathcal{R}, k)$: The problem of composing M using at most k instances of each FSM in $\mathcal{R}$.
3. $PBC(M, \mathcal{R}, k)$: The problem of composing M using simultaneously at most k FSM instances in $\mathcal{R}$ (in parallel).
4. $UCHS(M, \mathcal{R}, k)$: The problem of composing M using an unbounded number of instances of $\mathcal{R}$, where the number of hybrid states in $\mathcal{R}$ is bounded by $k \in \{0, 1, 2\}$.

Table in Fig. 2 displays known and new complexity results regarding the WSC problem.

Paper Organisation. Section 2 recalls some basic definitions needed in this paper. In Sect. 3, we investigate the problem of bounded web services composition and proves that it is Exptime-Complete. Next, we define web service composition with fixed number of parallel instances, and show that it is Exptime-Complete in general and is NP-complete when M is loop free and polynomial for $k = 1$. In Sect. 5, we consider the web service composition when the number of hybrid states is bounded. We show that this problem is Exptime-Complete for $k = 0$, $k = 1$ and $k = 2$. We conclude in Sect. 6.

M	Acyclic FSM	general FSM
$BC(M, \mathcal{R}, 1)$	NP-complete[11]	Exptime-complete [15]
$BC(M, \mathcal{R}, k)$	NP-complete[11]	Exptime-complete
$PBC(M, \mathcal{R}, 1)$	Polynomial	Polynomial
$PBC(M, \mathcal{R}, k)$	NP-complete	Exptime-complete
$WSC(M, \mathcal{R})$	NP-complete [11]	Decidable [11]
$UCHS(M, \mathcal{R}, 0)$	NP-complete[11]	Exptime-complete
$UCHS(M, \mathcal{R}, 1)$	NP-complete[11]	Exptime-complete
$UCHS(M, \mathcal{R}, 2)$	NP-complete[11]	Exptime-complete
$UCHS(M, \mathcal{R}, 3)$	NP-complete[11]	Open

Fig. 2. Complexity results of WSC sub-problems

2 Preliminaries

Service business protocols are formally described in this context as FSMs. We recall below the definition of such machines.

Definition 1 ***(Finite State Machine (FSM)).*** *A State Machine* (**SM**) *M is a tuple $M = (\Sigma_M, Q_M, F_M, q_M^0, \delta_M)$, where: Σ_M is a finite alphabet, Q_M is a set of states, $\delta_M \subseteq Q_M \times \Sigma_M \times Q_M$ is a set of labeled transitions, $F_M \subseteq Q_M$ is a set of final states, and $q_M^0 \in Q_M$ is the initial state. If Q_M is finite then M is called a Finite State Machine (FSM).*

Moreover, a state $q \in Q_M$ is called: **accessible**, if there exists a path from the initial state to q; **co-accessible**, if there exists a path from q to a final state. We consider here only FSMs where all states are both accessible and co-accessible. Hence, we can define the **norm of a state** q as the finite length of the shortest path from q to a final state. **The norm of an FSM** M, noted *norm(M)*, is the maximal norm of its states.

An execution of a FSM M can be seen as a path of a token that moves from a state $q \in Q_M$ to another $p \in Q_M$ linked by a transition $(q, a, p) \in \delta_M$, where $a \in \Sigma$. An execution is valid when the token begins its path in q_M^0 and finishes it in a final state $p_f \in F_M$. Every state that a token can visit during a valid execution, the initial q_0 and last p_f states excluded, is called an intermediate state. We denote by $I(M)$ the set of intermediate states of M. If a state is both final and intermediate, then it is called hybrid. The set of hybrid states is denoted $H(M)$. Finally, if a state p is final but not intermediate, (i.e. $p \in F_M \setminus H(M)$), then p is called terminal, since all executions reaching such state ought to be terminated. Note that every execution can be terminated when the token is in the initial state, i.e. initial state is in F_M.

k-Iterated Product Machine (k-IPM) and Product State Machine (PCSM). We start by defining the asynchronous product and union operations on FSMs:

Definition 2 ***(Asynchronous product and Union of two FSMs).*** *Let $M = (\Sigma_M, Q_M, F_M, q_M^0, \delta_M)$ and $M' = (\Sigma_{M'}, Q_{M'}, F_{M'}, q_{M'}^0, \delta_{M'})$ be two state machines. We have:*

- *The **shuffle or asynchronous product** of M and M', denoted $M \times M'$, is a state machine $(\Sigma_M \cup \Sigma_{M'}, Q_M \times Q_{M'}, F_M \times F_{M'}, (q^0_M, q^0_{M'}), \lambda)$ where the transition function λ is defined as follows: $\lambda = \{((q,q'),a,(q_1,q_1')) : ((q,a,q_1) \in \delta_M$ and $q' = q_1')$ or $((q',a,q_1') \in \delta_{M'}$ and $q = q_1)\}$.*
  For a set of state machines $\{M_1,...,M_k\}$ where $k \geq 3$, we define reciprocally $M_1 \times .... \times M_k$ as the state machine $((M_1 \times ... \times M_{k-1}) \times M_k)$.
- *The **union** of M and M', denoted $M \cup M'$, is the state machine $(\Sigma_M \cup \Sigma_{M'} \cup \{\epsilon\}$, $Q_M \cup Q_{M'} \cup \{q_0\}$, $F_M \cup F_{M'}$, q_0, $\delta_M \cup \delta_{M'} \cup \{(q_0, \epsilon, q^0_M), (q_0, \epsilon, q^0_{M'})\})$.*

For a set of available FSMs $\mathcal{R} = \{M_1, ...M_m\}$, we consider a compact structure that abstracts all possible executions that can be produced using the components of $\mathcal{R}$. First, we begin by the simple case where each M_j can be used only once:

Definition 3 ***(Union of asynchronous products of FSMs set).*** *For a FSMs repository $\mathcal{R} = \{M_1....M_m\}$, we denote the union of all asynchronous products of $\mathcal{R}$'s subsets as: $\odot(\mathcal{R}) = \bigcup_{\{i_1,...,i_j\} \subseteq \{1,...,m\}} (M_{i_1} \times ... \times M_{i_j})$.*

Second, we consider the case where the number of copies of each $M_j \in \mathcal{R}$ is bounded by an integer k:

Definition 4 ***(k-iterated product of FSMs set $\mathcal{R}$).*** *The **k-iterated product of** $\mathcal{R}$ is defined by $\mathcal{R}^{\otimes_k} = \mathcal{R}^{\otimes_{k-1}} \times \odot(\mathcal{R})$ with $\mathcal{R}^{\otimes_1} = \odot(\mathcal{R})$.*

Finally, we consider the general case where the number of instances of each $M_j \in \mathcal{R}$ is unbounded. This corresponds to the product closure of $\mathcal{R}$ [11]:

Definition 5 ***(Product closure of FSMs set).*** *The **product closure** of $\mathcal{R}$, noted $\mathcal{R}^{\otimes}$, is defined as: $\mathcal{R}^{\otimes} = \bigcup_{i=0}^{+\infty} \mathcal{R}^{\otimes_i}$.*

The **P**roduct **C**losure **S**tate **M**achine (**PCSM**) of $\mathcal{R}$, defined in [11] and proven equivalent to $\mathcal{R}^{\otimes}$, is the SM with unbounded number of tokens stacked at the beginning in the initial states in $\mathcal{R}$. Then, the instantaneous description of a PCSM gives the number of tokens (instances) at each state in $\mathcal{R}$ that the PCSM currently underlies. This description is called a configuration of $\mathcal{R}^{\otimes}$ and every component of the configuration is called a witness of its corresponding state. We omit from this description the initial states (source: infinite number of tokens) and terminal states (sink: terminated instances) and represent only intermediate and hybrid states.

A configuration in $\mathcal{R}^{\otimes}$ is called final if all witnesses that correspond to intermediate states (not final) are equal to 0. We define a path of a PCSM as a sequence of transitions between configurations in $\mathcal{R}^{\otimes}$ and a path's length as the number of its transitions. For a configuration $c \in \mathcal{R}^{\otimes}$, $norm(c)$ denotes the length of the shortest path from c to a final configuration.

Example 1. Figure 3 illustrates the execution of the sequence "*abca*" by the PCSM of the FSM set $\{M, M'\}$ in Fig. 3(a). M and M' contain one intermediate state q_1 and two hybrid states q_2 and q_5. Therefore, Fig. 3(b) depicts a part of the PCSM $\{M, M'\}^{\otimes}$ with triplets as configurations where integers witness respectively the number of tokens in q_1, q_2 and q_5. For each configuration c in Fig. 3(b), we associate an instant t (or several instants) during the execution when c describes the PCSM. At the beginning ($t = 0$), $\{M, M'\}^{\otimes}$'s instantaneous description is $(0, 0, 0)$, interpreting an empty stack in every state of M and M', except the initial states q_0 and ${q'}_0$ with an infinite number of tokens (Fig. 3(c)). To execute the transition (q_0, a, q_1), a token is moved from q_0 to q_1 in Fig. 3(d), corresponding to the configuration $(1, 0, 0)$ in instant $t = 1$. In $t = 2$, the executed transition $({q'}_0, b, q_4)$ corresponds to moving a token from the initial state ${q'}_0$ to a terminal one q_4 (Fig. 3(e)). Since the instantaneous description considers neither initial states nor terminal ones, then the configuration stays the same as the previous instant. Notice that this move corresponds to both creating and terminating an instance of the FSM M'. Then, the transition $({q'}_0, c, q_5)$ is executed by moving a token from ${q'}_0$ to the hybrid state q_5. This creates a new instance implying, in this case, an increase in the number of simultaneously used instances in the execution. This is depicted in Fig. 3(f). Finally, a token is moved from the state q_1 to q_2 in Fig. 3(g), in order to execute the transition (q_1, a, q_2). It changes $\{M, M'\}^{\otimes}$'s instantaneous description in $t = 4$ into $(0, 1, 1)$ which is a final configuration (i.e $(0, 1, 1) \in F_C$) since all tokens in the PCSM are in final states (either hybrid or terminal).

Formally, we define the PCSM of $\mathcal{R}$ as the SM $(\Sigma_{\mathcal{R}}, \mathcal{C}_{\mathcal{R}^{\otimes}}, F_{\mathcal{C}}, c_0, \Phi_{\mathcal{R}^{\otimes}})$, where:

1. $\Sigma_{\mathcal{R}} = \bigcup_{M_j \in \mathcal{R}} \Sigma_{M_j}$;
2. $\mathcal{C}_{\mathcal{R}^{\otimes}}$ is the set of states (also called configurations of $\mathcal{R}^{\otimes}$). $\mathcal{C}_{\mathcal{R}^{\otimes}} \subset \mathbb{N}^n$, with: $n = n_I(\mathcal{R}) + n_H(\mathcal{R})$ with: $n_I(\mathcal{R}) = \Sigma_{M_j \in \mathcal{R}} |I(M_j)|$ and $n_H(\mathcal{R}) = \Sigma_{M_j \in \mathcal{R}} |H(M_j)|$. For each configuration c, $c[m]$ (the m^{th} component of c) is called a witness of the unique state $q_m \in Q_{M_j}$. Note that:
   - q_m is an intermediate state, if $1 \leq m \leq n_I(\mathcal{R})$;
   - q_m is a hybrid state, if $n_I(\mathcal{R}) + 1 \leq m \leq n$.

   In an abuse of notation, we use $c[m]$ and $c[q_m]$ interchangeably.
3. $F_{\mathcal{C}}$ is the set of final states. $F_{\mathcal{C}} = \{c \in \mathcal{C}_{\mathcal{R}^{\otimes}} | c[m] = 0,\ \textit{for each:}\ 1 \leq m \leq n_I(\mathcal{R})\}$;
4. $c_0 = \{0\}^n$ is the initial state of $\mathcal{R}^{\otimes}$;
5. $\Phi_{\mathcal{R}^{\otimes}} \subseteq \mathcal{C}_{\mathcal{R}^{\otimes}} \times \Sigma_{\mathcal{R}} \times \mathcal{C}_{\mathcal{R}^{\otimes}}$ is the set of transitions. we have $(c_1, a, c_2) \in \Phi_{\mathcal{R}^{\otimes}}$ if and only if:
   - there exists $(q_0, a, q) \in Q_{M_j}$, such that: q_0 is the initial state of M_j and $c_2[q] = c_1[q] + 1$ and $c_2[p'] = c_1[p']$ for each $p' \neq q$. Or
   - there exists $(p, a, q) \in Q_{M_j}$, such that: p is not the initial state of M_j, q is either a terminal state or the initial state of M_j, $c_2[p] = c_1[p] - 1$ and $c_2[p'] = c_1[p']$ for each $p' \neq p$. Or
   - there exists $(p, a, q) \in Q_{M_j}$, such that: neither p nor q is the initial state of M_j, $c_2[p] = c_1[p] - 1$, $c_2[q] = c_1[q] + 1$ and $c_2[p'] = c_1[p']$ for each $p' \neq p, q$.

We recall below the definition of the simulation preorder between two SMs.

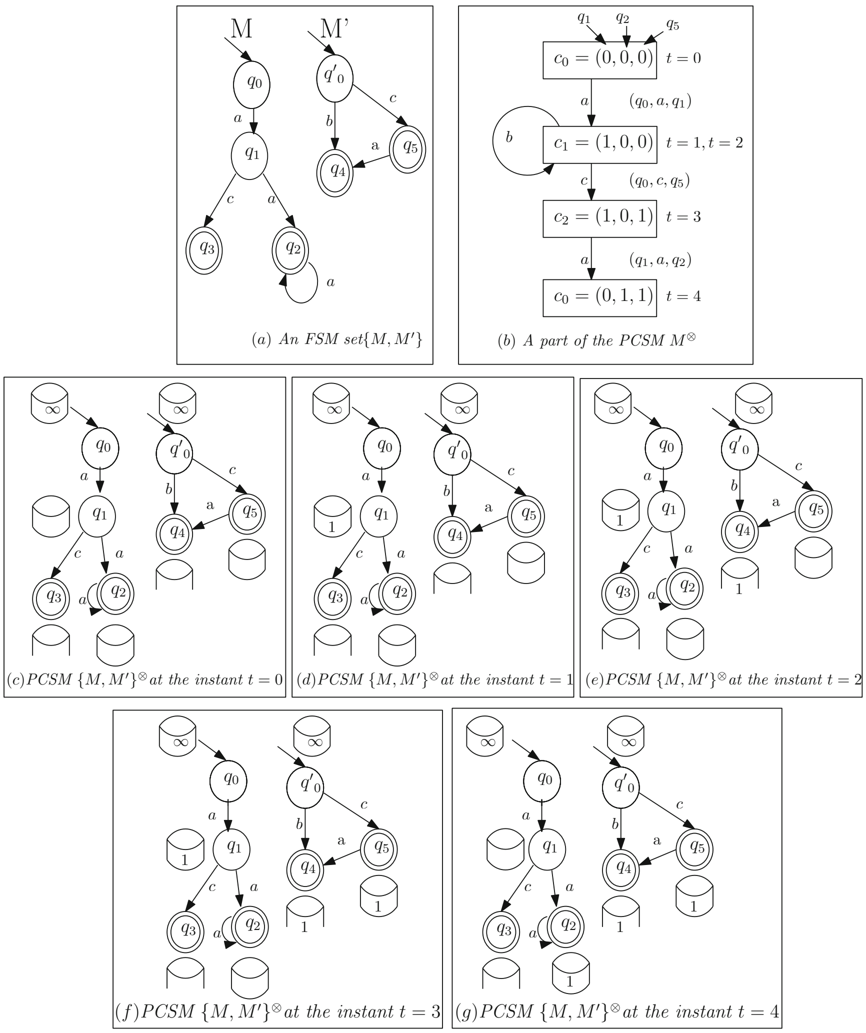

M
M'
(a) An FSM set{M, M'}
(b) A part of the PCSM M^⊗
c0 = (0,0,0) t = 0
c1 = (1,0,0) t = 1, t = 2
c2 = (1,0,1) t = 3
c0 = (0,1,1) t = 4
(q0, a, q1)
(q0, c, q5)
(q1, a, q2)
(c)PCSM {M, M'}^⊗ at the instant t = 0
(d)PCSM {M, M'}^⊗ at the instant t = 1
(e)PCSM {M, M'}^⊗ at the instant t = 2
(f)PCSM {M, M'}^⊗ at the instant t = 3
(g)PCSM {M, M'}^⊗ at the instant t = 4

Fig. 3. An example of execution of a sequence using a PCSM.

Definition 6 *(Simulation).* *Let* $M = (\Sigma_M, Q_M, F_M, q^0_M, \delta_M)$ *and* $N = (\Sigma_N, Q_N, F_N, q^0_N, \delta_N)$ *be two SMs. A state* $p \in Q_M$ *is simulated by a state* $q \in Q_N$*, denoted* $p \ll_{(M,N)} q$ *(*$p \ll q$ *when* M *and* N *are understood from context), if and only if the following two conditions hold:*

1. $\forall a \in \Sigma_M$ *and* $\forall p' \in Q_M$ *such that* $(p, a, p') \in \delta_M$*, there exists* $(q, a, q') \in \delta_N$ *such that* $p' \ll q'$*, and*
2. *if* $p \in F_M$*, then* $q \in F_N$*.*

M is simulated by N, denoted $M \ll N$*, if and only if the initial state of N simulates the initial state of M,* i.e. $q^0_M \ll q^0_N$.

Observe that, by definition, each transition of a PCSM can at most increase or decrease a configuration component by 1. In addition, if a configuration is final then all intermediate states witnesses are equal to 0. Therefore, given a set of FSMs $\mathcal{R}$ and $c \in \mathcal{C}_{\mathcal{R}^{\otimes}}$, we have $\Sigma_{q \in \bigcup_{M_i \in \mathcal{R}} I(M_i)} c[q] \leq norm(c)$. Moreover, since final states can only be simulated by final ones, then for M an FSM and $p \in Q_M$, if $p \ll c$ then $norm(c) \leq norm(p)$. Hence, we are able to derive the following property.

Property 1 **(Intermediate witnesses bound)** [11]**.** For $c \in \mathcal{C}_{\mathcal{R}^{\otimes}}$ and $p \in Q_M$, if $p \ll c$ then $\Sigma_{q \in I(\mathcal{R})} c[q] \leq norm(p)$, where $I(\mathcal{R}) = \bigcup_{M_i \in \mathcal{R}} I(M_i)$.

We denote $\mathcal{C}^M_{\mathcal{R}^{\otimes}} = \{c \in \mathcal{C}_{\mathcal{R}^{\otimes}} | \Sigma_{q \in I(\mathcal{R})} c[q] \leq norm(M)\}$.

In [11], the WSC problem in the unbounded case is reduced to simulation test between an FSM and a PCSM and proven to be decidable. The termination of the algorithm given in [11] is proven using the following property:

Property 2 **(configuration cover)** [11]**.** Let c and c' be two configurations of $\mathcal{R}^{\otimes}$, such that: $c[m] = c'[m], m \in [1, n_I(\mathcal{R})]$ *and* $c[m] \leq c'[m], m \in [n_I(\mathcal{R})+1, n]$. If $q \ll c$, where q is a state of a SM M, then $q \ll c'$.

We say that c' covers c, denoted $c \lhd c'$.

We introduce below the algorithm of [11], focusing the presentation on the structure of its execution tree.

Definition 7 ***(Simulation Tree of an FSM by a PCSM).*** *We call a simulation tree $T_{sim}(M, \mathcal{R}^{\otimes}) = (V, v_0, E)$ with:*

- *$v_0 = (q^0_M, c_0)$ is the root of the tree;*
- *$V \subset Q_M \times \mathcal{C}^M_{\mathcal{R}^{\otimes}}$ is the set of nodes;*
- *If $(q, c) \in V$ and q is final in M then so is c in $\mathcal{R}^{\otimes}$;*
- *$E \subset V \times V$ is the set of the tree's edges. $\forall e = ((p, c), (q, d)) \in E : \exists a \in \Sigma_M$ s.t $(p, a, q) \in \delta_M$ and $(c, a, d) \in \Phi_{\mathcal{R}^{\otimes}}$.*
- *$v = (p, c) \in V$ is a leaf in $T_{sim}(M, \mathcal{R}^{\otimes})$ iff p is terminal in A or there exists an ancestor $(p, c') \in V$ of v in $T_{sim}(M, \mathcal{R}^{\otimes})$ such that $c \lhd c'$.*

Example 2. Figure 4(c) is an example of a simulation tree, verifying if the initial state s_0 of the FSM A (Fig. 4(a)) is simulated by the initial configuration $c_0 = (0, 0, 0)$ of the PCSM $\{M, M'\}^{\otimes}$ (Fig. 4(b)). A branch is terminated with success when a terminal state of A is reached and paired with a final configuration (all intermediate witnesses are null), or when a configuration of $\{M, M'\}^{\otimes}$ that covers one of its predecessors is reached and paired with the same state of A. In this case, the simulation tree proves that $A \ll \{M, M'\}^{\otimes}$.

In the next section, we shall bound the size of this tree in the case of bounded WSC problem (*i.e.*, when the number of web services instances allowed to be used in the simulation is bounded by a parameter k).

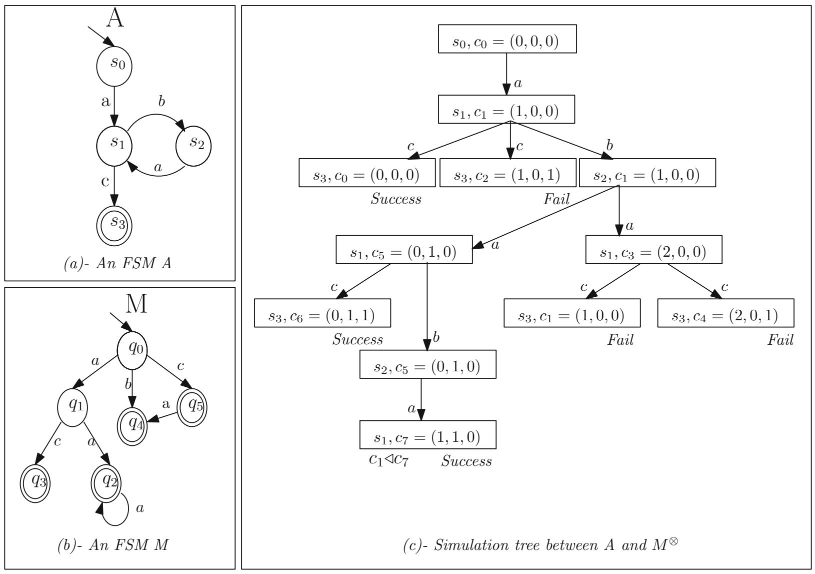


Fig. 4. An example of a simulation tree.

3 Bounded Composition

We call a *bounded* WSC problem, a service composition problem where the number of copies of each web service in the repository $\mathcal{R}$ used to compose the target M is bounded a priori by an integer k. This problem is formally stated as follows.

Problem 1. **Bounded Composition** $BC(M, \mathcal{R}, k)$
Input: $\mathcal{R}$ a set of FSMs; M a target FSM; k an integer.
Question: $M \ll \bigcup_{i=0}^{k} \mathcal{R}^{\otimes_i}$?

The particular case $BC(M, \mathcal{R}, 1)$ has been investigated by Muscholl and Walukiewicz [15] where it is shown to be Exptime-Complete. We shall prove in this section that $BC(M, \mathcal{R}, k)$ is also Exptime-Complete. We point out that the straightforward reduction of $BC(M, \mathcal{R}, k)$ to $BC(M, \mathcal{R}, 1)$, obtained by duplicating k times each service of $\mathcal{R}$, is not polynomial in the input size, since k may be large, and hence cannot be used to achieve our goal.

The parameter k drops the infinite aspect and reduces the search space. In this case, a loop in M can only be simulated by loops in $\mathcal{R}$. For example, one can observe that, in Fig. 5, S_t is not simulated by $\bigcup_{i=0}^{k}\{R_1, R_3\}^{\otimes_i}$ for every $k \in \mathbb{N}$. This is because when we repeat the loop in S_t $(k+1)$ times, there is no corresponding execution in $\bigcup_{i=0}^{k}\{R_1, R_3\}^{\otimes_i}$. However, we have $S_t \ll \bigcup_{i=0}^{k}\{R_1, R_2\}^{\otimes_i}$, for any $k \geq 1$.

In the following, we give an upper bound of the number of states that might appear in $\bigcup_{i=0}^{k} \mathcal{R}^{\otimes_i}$, with $k \in \mathbb{N}$.

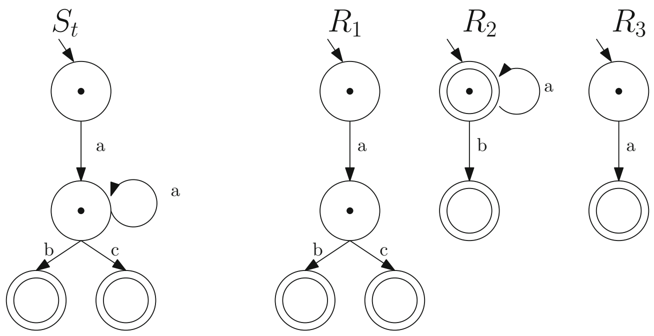


Fig. 5. A yes instance of $BC(M, \mathcal{R}, k)$ with $k = 1$.

Lemma 1. *Let $\mathcal{R}$ be a set of FSM and k is an integer. The number of states in $\bigcup_{i=0}^{k} \mathcal{R}^{\otimes_i}$ is bounded by $O(2^{nlogk})$ where $n = n_I(\mathcal{R}) + n_H(\mathcal{R})$.*

Proof. Notice that $\mathcal{R}^{\otimes} = (\bigcup_{i=0}^{k} \mathcal{R}^{\otimes_i}) \cup (\bigcup_{i=k+1}^{+\infty} \mathcal{R}^{\otimes_i})$.

In fact, the states in $\bigcup_{i=0}^{k} \mathcal{R}^{\otimes_i}$ correspond to the PCSM's configurations subset $\{c \in \mathcal{C}_{\mathcal{R}^{\otimes_k}} \mid 0 \leq c[i] \leq k, i \in [1, n]\}$. Hence, the number of states of $\bigcup_{i=0}^{k} \mathcal{R}^{\otimes_i}$ is bounded by $(k+1) \times \ldots \times (k+1) = 2^{nlog(k+1)}$. □

This lemma reduces the search space to an exponential size and leads to the following theorem.

Theorem 1. *$BC(M, \mathcal{R}, k)$ is Exptime-Complete.*

Proof. `Exptime.` To show that $BC(M, \mathcal{R}, k)$ is Exptime, we bound the size of the simulation tree. A node of the simulation tree corresponds to (q, c) where q is a state of M and c a configuration of $\mathcal{R}^{\otimes_k}$. According to Lemma 1, the number of PCSM's configurations is bounded by k^n. So the number of nodes in the simulation tree is at most $|Q_M| \times k^n = 2^{nlog(k)+log(|Q_M|)}$ and therefore the complexity is in Exptime.

`Exptime-Hardness.` It can be deducted directly from the Exptime-Hardness of the particular case $BC(M, \mathcal{R}, 1)$ [15].

Instead of the total number of instances used in the simulation, what happens if we bound only the number of instances used simultaneously? we raise this question in the next section and prove that the problem stays Exptime-complete.

4 Bounded Parallel Instances

Now we consider a new parameter in service web composition that bounds the number of communications in parallel between the target and the services, i.e. the number of live services executions is bounded, but the number of instances is not. It turns out that the web services composition with unbounded instances and bounded parallel instances is Exptime-Complete.

To show this, we limit the configurations of the PCSM $\mathcal{R}^{\otimes}$ to configurations where the number of waiting instances is bounded by k. Indeed, when we need to use a new instance in $\Phi_{\mathcal{R}^{\otimes}}$, we check if $\sum_{i=1}^{n} c[i] \geq k$. If so, we decrease $c[j]$ for some $j \in [n_I(\mathcal{R})+1, n_I(\mathcal{R})+n_H(\mathcal{R})]$, i.e. we finish an instance that is waiting in a hybrid state. Let us denote by $\mathcal{R}^{\otimes_{k,p}}$ the obtained PCSM.

Problem 2. **Bounded Parallel Instances Composition** ($PBC(M, \mathcal{R}, k)$)
Input: $\mathcal{R}$ a set of FSMs;
M a target FSM.
k an integer, bounding the number of parallel instances of $\mathcal{R}$'s components used simultaneously in the simulation.
Question: $M \ll \mathcal{R}^{\otimes_{k,p}}$?

Note that $PBC(M, \mathcal{R}, k)$ can use an unbounded number of instances but only k instances in parallel.

Theorem 2. *$PBC(M, \mathcal{R}, k)$ is Exptime-complete.*

Proof. First we show that $PBC(M, \mathcal{R}, k)$ is Exptime. Clearly the entry of any configuration is bounded by k (hybrid states are included) and therefore we can check simulation in Exptime, since the depth of the simulation tree is bounded by k^n (see Lemma 1).

To show the Exptime-hardness, it suffices to note that the unbounded composition without hybrid states $UCHS(M, \mathcal{R}, 0)$ is a particular case of $PBC(M, \mathcal{R}, k)$, since we prove later in Theorem 4 that $UCHS(M, \mathcal{R}, 0)$ is Exptime-hard. In fact, the number of tokens in intermediate states of $\mathcal{R}$ is bounded by $norm(M)$ (Property 1). Hence, when $\mathcal{R}$ is hybrid state free, the number of instances that can be used in the simulation is bounded by $norm(M)$. In other words, it corresponds to $PBC(M, \mathcal{R}, norm(M))$. □

For k a constant, we obtain the following.

Corollary 1. *For each fixed k, $PBC(M, \mathcal{R}, k)$ is polynomial.*

Proof. First of all, let us consider for every configuration c of $\mathcal{R}^{\otimes_{k,p}}$, a new component $c[n+1] = k - (\sum_{i=1}^{n} c[i])$, with $n = n_I(\mathcal{R}) + n_H(\mathcal{R})$.

For every configuration c in $\mathcal{R}^{\otimes_{k,p}}$, the non-empty witnesses $\{c[i] > 0, \leq i \leq n+1\}$ correspond to a partition of k elements (instances) into a sequence of j non empty subsets, for $j = |\{c[i] > 0, 1 \leq i \leq n\}| \leq k$. Note that j is in fact inferior to $min(k, n)$, but since k is a constant then it is more interesting to keep it as a lower bound of j.

For every $j \leq k$, the number of labeled partitions of k elements into a sequence of j non empty subsets is $j! \times \left\{ {k \atop j} \right\}$, where $\left\{ {k \atop j} \right\}$ is a Stirling number of the second kind [1]. Hence, the number of configurations in $\mathcal{R}^{\otimes_{k,p}}$ that have j non-empty witnesses is bounded by $C_n^j \times j! \times \left\{ {k \atop j} \right\}$. Notice that $C_n^j = e^{\frac{n \ldots \times (n-j+1)}{j!}}$ is in the order of $O(n^j)$.

We conclude that the number of configurations in $\mathcal{R}^{\otimes_{k,p}}$ is bounded by $\sum_{j=1}^{k} C_n^j \times j! \times \left\{ {k \atop j} \right\} \in O(n^k)$.

Finally, by applying the simulation algorithm in [12], $PBC(M, \mathcal{R}, k)$ can be decided in $O(m_v.m_e)$, where $m_v = |Q_M| + |\mathcal{C}_{\mathcal{R}^\otimes}|$ and $m_e \leq |Q_M|^2 + |\mathcal{C}_{\mathcal{R}^\otimes}|^2$ are respectively the number of edges and transitions in M and $\mathcal{R}^{\otimes_{k,p}}$. □

In the following, we show that $PBC(M, \mathcal{R}, k)$ is NP-Complete for loop-free target FSM. Let μ a sequence of letters (a word) over Σ and M the FSM that recognizes exactly μ. We call $\mu^\otimes$ the language recognized by $M^\otimes$. We consider the following NP-complete Problem [13].

Problem 3. **SHUFFLE PRODUCT**
Input: μ and μ' two words over an alphabet Σ;
Question: $\mu \in \mu'^\otimes$?

Theorem 3. *$PBC(M, \mathcal{R}, k)$ is NP-complete whenever M is loop-free.*

Proof. Clearly $PBC(M, \mathcal{R}, k)$ is in NP since the simulation relation is polynomial in the size of M. To show the NP-hardness, we reduce SHUFFLE PRODUCT to it. Let μ and μ' be an instance of SHUFFLE PRODUCT. We associate an FSM M which recognizes exactly μ and $\mathcal{R} = \{N\}$ where N is the FSM that recognizes exactly μ'. Since M is a chain, then the size of a branch of the simulation tree can not surpass $|\mu|$. Thus, the simulation verification will only explore $\mathcal{R}^{\otimes_{k,p}}$'s executions where the size is bounded by $|\mu| \leq k.|\mu'|$ with $k = \lceil \frac{|\mu|}{|\mu'|} \rceil$ and therefore the number of instances is bounded by k. Hence, $\mu \in \mu'^\otimes$ iff $M \ll \mathcal{R}^{\otimes_{k,p}}$ iff $M \ll \mathcal{R}^\otimes$. We give an example in Fig. 6. □

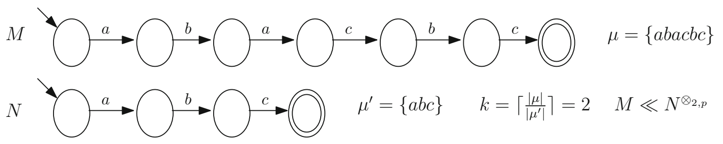

M
a
b
a
c
b
c
$\mu = \{abacbc\}$
N
a
b
c
$\mu' = \{abc\}$
$k = \lceil \frac{|\mu|}{|\mu'|} \rceil = 2$
$M \ll N^{\otimes_{2,p}}$

Fig. 6. An example of **SHUFFLE PRODUCT** problem.

Another factor of complexity of the WSC problem is the number of hybrid states in the available services. We investigate next the effect of this parameter on the complexity of the WSC problem.

5 Bounded Number of Hybrid States

The presence of hybrid states is a source of complexity in a WSC problem. As mentioned before, the size of intermediate states witnesses in configurations of $\mathcal{R}^\otimes$ used to simulate M is bounded by $norm(M)$. We are however unable to provide a similar bound for the number of hybrid states witnesses.

Figure 7 is an example of simulation between an FSM M and a PCSM $\mathcal{R}^{\otimes}$. The FSMs in $\mathcal{R}$ contain two hybrid states (state 1 and 2) and no intermediate state. Hence, a configuration of $\mathcal{R}^{\otimes}$ is a pair of integers witnessing the number of tokens in state 1 and state 2. The example illustrates the different roles that a hybrid state of $\mathcal{R}$ can play to simulate a state of M. Indeed a hybrid state of $\mathcal{R}$, can be used as:

(i) a terminal state, e.g., when testing whether $q_5 \ll (1,1)$, we can consider the second hybrid state of $\mathcal{R}$ as a terminal state and terminate the test, or
(ii) an intermediate state, e.g., when testing whether $q_2 \ll (1,1)$, the second hybrid state of $\mathcal{R}$ here plays the role of intermediate state, or
both a terminal and an intermediate state, e.g., when testing whether $q_1 \ll (1,0)$, a transition of $\Phi_{\mathcal{R}^{\otimes}}$ labeled by $(b,(-1,0))$ only appears in one branch in the simulation tree $\mathcal{T}_{sim}(M,\mathcal{R}^{\otimes})$. Hence, the first hybrid state of $\mathcal{R}^{\otimes}$ is considered intermediate in one branch and terminal in the other, or
a hybrid state, e.g., when it is used to simulate a hybrid state of $H(M)$.

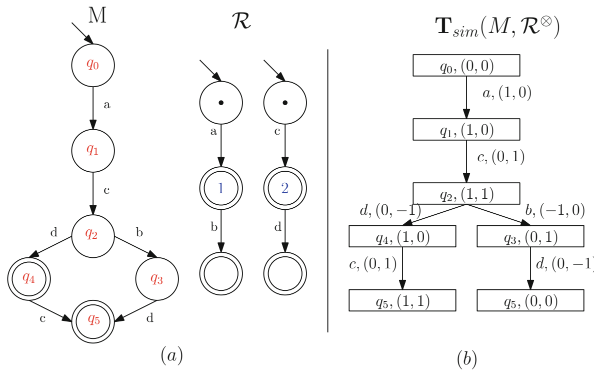


Fig. 7. Example of the simulation tree

We consider in the following the problem defined below.

Problem 4. **Unbounded Composition With limited number of Hybrid States** $UCHS(M,\mathcal{R},k)$
Input: k an integer; $\mathcal{R}$ a set of FSMs, containing at most k hybrid states; M a target FSM.
Question: $M \ll \mathcal{R}^{\otimes}$?

It is worth noting that $UCHS(M,\mathcal{R},k+1)$ is harder then $UCHS(M,\mathcal{R},k)$. In the sequel, we progressively investigate the complexity of $UCHS(M,\mathcal{R},k)$ problem for $k=0$, then for $k=1$ and finally for $k=2$.

5.1 Case of Composition without Hybrid States (i.e. $k = 0$)

In this section, we are interested in the problem $UCHS(M, \mathcal{R}, 0)$. We first give a polynomial transformation, denoted $\mathcal{K}$, which is used to reduce $BC(M, \mathcal{R}, 1)$ to $UCHS(N, \mathcal{R}', 0)$. This transformation provides a mean to bound the number of instances used to prove simulation.

Definition 8. ***Transformation*** $\mathcal{K}$. *For an FSM $M = (\Sigma_M, Q_M, F_M, q_0^M, \delta_M)$ and a set of FSMs $\mathcal{R} = \{M_1, ..., M_m\}$, we define $\mathcal{K}(M, \mathcal{R})$=$(N, \mathcal{R}'$=$\{N_1, .., N_m\})$ where:*

1. *Each N_i is built based on M_i, by adding a letter t_i to its alphabet, a final state f_i and a transition set $\{(q_0^{M_i}, t_i, f_i)\} \cup \{(q, t_i, f_i) | q \in F_{M_i}\}$. All final states of M_i become intermediate in N_i.*
2. *N is defined as:*
   - $\Sigma_N = \Sigma_M \cup \{t_i | 1 \leq i \leq m\}$;
   - $Q_N = Q_M \cup \{r_i | 1 \leq i \leq m\}$;
   - $F_N = \{r_m\}$;
   - $\delta_N = \delta_M \cup \{(q, t_1, r_1) | q \in F_M\} \cup \{(r_i, t_{i+1}, r_{i+1}) | 1 \leq i < m\}$.

Figure 8 illustrates an example of this transformation. We prove later in Proposition 2 that $\mathcal{K}$ defines a polynomial reduction of $BC(M, \mathcal{R}, 1)$ to $UCHS(N, \mathcal{R}', 0)$. In fact, the intuition behind this reduction is based on two points:

- By adding the sequence of letters $t_1, ..., t_m$ at the end of every execution accepted by N and adding t_i at the end of every execution accepted by $N_i \in \mathcal{R}'$, we ensure that even in an unbounded instances simulation, we can not use more than one instance of every N_i in order to simulate N.
- The construction of $\mathcal{R}'$ verifies that every hybrid state in $M_i \in \mathcal{R}$ becomes intermediate in N_i, while keeping its dual role: either terminate the execution by adding the letter t_i to the execution of N_i and reaching the terminal state f_i, or keep the execution in the same way as M_i.

The following propositions show that the transformation $\mathcal{K}$ preserves the simulation preorder.

Proposition 1. *Let M be an FSM, $\mathcal{R} = \{M_1, ..., M_m\}$ be a set of FSMs and $\mathcal{K}(M, \mathcal{R}) = (N, \mathcal{R}' = \{N_1, .., N_m\})$. For p and q two states of respectively M and $\mathcal{R}^{\otimes_1}$, we have: $p \ll_{(M, (\mathcal{R})^{\otimes_1})} q$ iff $p \ll_{(N, (\mathcal{R}')^{\otimes_1})} q$.*

Proof. By construction of $\mathcal{K}(M, \mathcal{R})$, *if* $p \ll_{(M, \mathcal{R}^{\otimes_1})} q$ and p is terminal in M *then* $p \ll_{(N, (\mathcal{R}')^{\otimes_1})} q$.

We suppose next that:

If $(p, a, p') \in \delta_M$, $(q, a, q') \in \delta_{\mathcal{R}^{\otimes_1}}$ and $p' \ll_{(M, \mathcal{R}^{\otimes_1})} q'$, *then* $p' \ll_{(N, (\mathcal{R}')^{\otimes_1})} q'$.

and prove that $p \ll_{(N, (\mathcal{R}')^{\otimes_1})} q$.

For each $(p, a, p') \in \delta_N$, we have:

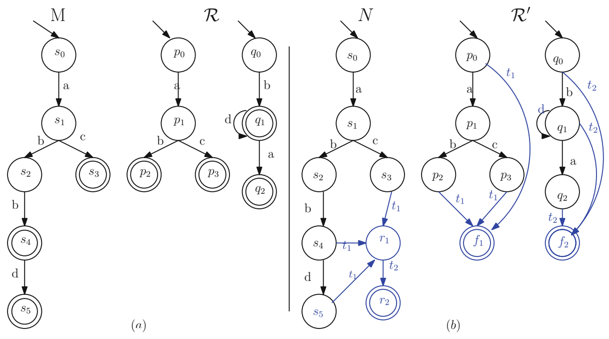


Fig. 8. An example of transformation $\mathcal{K}$

- *if* $a \in \Sigma_M$, *then* there exists $(q, a, q') \in \delta_{\mathcal{R}^{\otimes_1}} \subseteq \delta_{(\mathcal{R}')^{\otimes_1}}$ such that $p' \ll_{(N,(\mathcal{R}')^{\otimes_1})} q'$.
- else $a = t_1$, $p' = r_1$ and q is a product of final states of $\mathcal{R}$. therefore, there exists $(q, t_1, q') \in \delta_{(\mathcal{R}')^{\otimes_1}}$ such that $q' = (f_1, {q'}_{i_1}, ..., {q'}_{i_l})$ where ${q'}_{i_j}$ is final in $\mathcal{R}$ such that $p' \ll_{(N,(\mathcal{R}')^{\otimes_1})} q'$.

We conclude that if $p \ll_{(M,\mathcal{R}^{\otimes_1})} q$ then $p \ll_{(N,(\mathcal{R}')^{\otimes_1})} q$.

Reciprocally, we have $(p, a, p') \in \delta_N$ (respectively $\delta_{(\mathcal{R}')^{\otimes_1}}$) and $a \notin \{t_i | 1 \leq i \leq m\}$ iff $(p, a, p') \in \delta_M$ (respectively $\delta_{\mathcal{R}^{\otimes_1}}$). In addition, the definition of $\mathcal{K}$ ensures that if p is final in M and $p \ll_{(N,(\mathcal{R}')^{\otimes_1})} q$ then q is final in $\mathcal{R}^{\otimes_1}$. Hence if $p \ll_{(N,(\mathcal{R}')^{\otimes_1})} q$ then $p \ll_{(M,\mathcal{R}^{\otimes_1})} q$. □

In particular, we take p as the initial state of M and q the initial state of $\mathcal{R}^{\otimes_1}$. This implies that:

Proposition 2. *Let M be an FSM, $\mathcal{R} = \{M_1, ..., M_m\}$ be a set of FSMs and $\mathcal{K}(M, \mathcal{R}) = (N, \mathcal{R}' = \{N_1, .., N_m\})$. We have: $M \ll \mathcal{R}^{\otimes_1}$ iff $N \ll (\mathcal{R}')^{\otimes}$.*

Hence, $\mathcal{K}$ is a polynomial reduction of $BC(M, \mathcal{R}, 1)$ problem to the UCHS problem. This enables to derive the following result.

Theorem 4. *$UCHS(M, \mathcal{R}, 0)$ problem is Exptime-complete.*

Proof. According to Proposition 2, the $\mathcal{K}$ transformation reduces $BC(M, \mathcal{R}, 1)$ to $UCHS(M, \mathcal{R}, 0)$ in polynomial time. Thus $UCHS(M, \mathcal{R}, 0)$ is Exptime-hard. Since it is also proven Exptime in [11], then $UCHS(M, \mathcal{R}, 0)$ is Exptime-complete. □

5.2 Case of Composition with One Hybrid State

We consider the problem $UCHS(M, \mathcal{R}, 1)$ where M is an FSM and $\mathcal{R}$ a set of FSMs containing at most one hybrid state ($n_H(\mathcal{R}) \leq 1$). We denote $k_0 = |Q_M|.2^{n_I(\mathcal{R}).log(norm(M))}$. Two nodes (q, c) and (q', c') in a simulation tree are called comparable if $q = q'$ and either $c \lhd c'$ or $c' \lhd c$. The nodes (q, c) and (q', c') are said incomparable otherwise.

Property 3. Let $\mathcal{R}$ be a set of FSMs containing at most one hybrid state. Two configurations of $\mathcal{R}^{\otimes}$ are comparable by the cover relation, if and only if they have exactly the same intermediate witnesses.

Proof. According to Property 2, for c, c' two configurations in $\mathcal{R}^{\otimes}$ we have $c \triangleleft c'$ *iff*:

1. c and c' have the same intermediate witnesses; and
2. for every hybrid witness $c[h]$, we have: $c[h] \leq c'[h]$.

In the current case, we consider that $\mathcal{R}$ has at most one hybrid witness. Hence, for any pair of configurations of $\mathcal{R}^{\otimes}$, condition 2 is verified.

We conclude that for every two configurations c, c' in $\mathcal{R}^{\otimes}$, $c \triangleleft c'$ *iff* c and c' have the same intermediate witnesses. □

Property 4. Let S be a set of nodes of $\mathcal{T}_{sim}(M, \mathcal{R}^{\otimes})$ that are pairwise incomparable, *then* $|S| \leq k_0$.

Proof. In configurations considered in $\mathcal{T}_{sim}(M, \mathcal{R}^{\otimes})$, intermediate witnesses are bounded by $norm(M)$ (Property 1). Therefore and according to Property 3, the number of incomparable configurations considered in $\mathcal{T}_{sim}(M, \mathcal{R}^{\otimes})$ is at most $2^{n_I(\mathcal{R}).log(norm(M))}$. Since $S \subset Q_M \times \mathcal{C}_{\mathcal{R}^{\otimes}}$, then $|S| \leq k_0$. □

Proposition 3. *If* $n_H(\mathcal{R}) = 1$*, then for each* $(q, c) \in \mathcal{T}_{sim}(M, \mathcal{R}^{\otimes})$*, the witness* $c[h]$ *of the unique hybrid state in* $\mathcal{R}$ *is bounded by* $O({k_0}^2)$*.*

Proof. Let P be a path in $\mathcal{T}_{sim}(M, \mathcal{R}^{\otimes})$ and $S = (v_i = (q_i, c_i))_{n \in \mathbb{N}}$ be the subsequence of nodes in P satisfying the following properties:

- The node v_i is comparable to one of its predecessors $v = (q_i, c)$ in P, i.e. v appears before v_i in P; and
- For each $v_i, v_j \in S$, v_i and v_j are incomparable.

If $S = \emptyset$, then all nodes of P are not comparable. The size of P is then bounded by k_0, therefore, $c[n_I(\mathcal{R}) + 1] \leq k_0$ for each (q,c) in P.

Now suppose that $S = (v_1 ... v_k)$, $k \in \mathbb{N}$. We prove by induction on the size of S, i.e. for each $n \in [1, k]$, $c_n[h] \leq nk_0$.

For $n = 1$, all predecessors of v_1 in P are pairwise incomparable. Hence, $c_1[h] \leq k_0$ (Property 4).

For $1 < n < k$, we suppose that $c_n[h] \leq nk_0$. Each node $v = (q, c)$ strictly between v_n and v_{n+1} in P, v is not in S, therefore either:

1. v is comparable to a node v_i with $i \in [1, n]$. In this case, $c[h] < c_i[h] \leq n.k_0$ (otherwise $c_i \triangleleft c$, thus by definition of $\mathcal{T}_{sim}(M, \mathcal{R}^{\otimes})$, v should be a leaf).
2. v is incomparable to all its predecessors. The number of such nodes in P is strictly bounded by $k_0 - 1$ (because they should all be incomparable to each other and to v_n and v_{n+1}). And since transitions displacements is in $\{-1, 0, 1\}^h$, then we have $c[h] < n.k_0 + k_0 - 1$.

We conclude from above that for every $v = (q, c)$ between v_n and v_{n+1} in P, $c[h] \leq nk_0 + k_0 - 1$ (supposing w.l.o.g that $k_0 \geq 1$). Thus, $c_{n+1}[h] \leq nk_0 + k_0 = (n+1)k_0$.

Once we reach the last node v_k in S, all its possible successors in P are either: comparable to a node $v_i \in S$ with $c[h] < c_i[h]$, or incomparable to any of its predecessors in P or it is the leaf of P.

Finally, since $k < k_0$ (because S is a sequence of incomparable nodes), we conclude that each node of P is in $Q_M \times ([1, norm(A)]^{n_I(\mathcal{R})} \times [1, {k_0}^2 + k_0])$. □

Lemma 2. *$UCHS(M, \mathcal{R}, 1)$ is in Exptime.*

Proof. To show that $UCHS(M, \mathcal{R}, 1)$ is Exptime, we bound the size of the simulation tree. A node of the simulation tree corresponds to (q, c) where q is a state of M and c a configuration of $\mathcal{R}^{\otimes}$ that verifies, according to Proposition 3, the following:

- $c[h] \leq {k_0}^2$ where $c[h]$ is the witness of the unique hybrid state in $\mathcal{R}$;
- $c[i] \leq norm(M)$ where $c[i]$ is a witness of an intermediate state in $\mathcal{R}$.

Hence, the number of nodes in the simulation tree is bounded by

$$\underbrace{|Q_M|.norm(M)^{n_I(\mathcal{R})}}_{k_0}.\ (k_0^2 + k_0) = O(k_0^3)$$

And since deciding simulation only requires to visit each node once, then the complexity is in Exptime. □

To prove the Exptime-hardness of the problem, we recall that $UCHS(M, \mathcal{R}, 0)$ is Exptime-hard (Theorem 4) and that $UCHS(M, \mathcal{R}, 1)$ is harder than $UCHS(M, \mathcal{R}, 0)$.

Theorem 5. *$UCHS(M, \mathcal{R}, 1)$ is Exptime-complete.*

5.3 Case of Composition with Two Hybrid States

In this section, we consider the problem of unbounded composition of web services with at most 2 hybrid states in $\mathcal{R}$, i.e. $UCHS(M, \mathcal{R}, 2)$.

Our approach is based on reducing the simulation problem to the Z-reachability issue [7,9].

Interestingly, the simulation verification has been reduced in [11] to a two players game in a directed graph $(V_{att}, V_{def}, \delta, v_0)$, such that $V = V_{att} \cup V_{def}$ is the set of vertices with $V_{att} \subseteq Q_M \times Q_N$ and $V_{def} \subseteq Q_M \times Q_N \times \Sigma_M$, $\delta \subseteq (V_{att} \times V_{def}) \cup (V_{def} \times V_{att})$ is the edge set verifying:

- for $(q, p) \in V_{att}$ and $(q, a, q') \in \delta_M$, we have $((q, p), (q', p, a)) \in \delta$; and
- for $(q, p, a) \in V_{def}$ and $(p, a, p') \in \delta_N$, we have $((q, p, a), (q, p')) \in \delta$.

The game is played by an attacker and a defender. It starts by putting a token in $v_0 = (q^0_M, q^0_N) \in V_{att}$, then the players move it along the edges of the graph. If the token is on a vertex $v \in V_{att}$ then the attacker moves it, otherwise it is the defender's turn.

A strategy of a player $x \in \{a, d\}$ is a function $S : V^*.V_x \mapsto V$, where $V^*.V_x$ denotes all sequences of vertices in V that end with a vertex in V_x and $S(v_0, ..., v_k) = v_{k+1}$ implies that $(v_k, v_{k+1}) \in \delta$. In each different play, a player x adapts a strategy that decides his moves.

The defender wins every infinite play. Otherwise, the first player who cannot move loses. M is simulated by N **iff** the defender has a winning strategy regardless of his opponent's strategy.

The Z-reachability game, on the other hand, is played on a finite weighted graph $(V_\Box, V_\Diamond, E, v_0, e_0, k)$ by 2 players $\Box$ and $\Diamond$. A play begins by placing a token in $v_0 \in V_\Box$, then the players move it along the graph's edges $E \subseteq V \times V \times \{-1, 0, 1\}^k$, with $V = V_\Box \cup V_\Diamond$ and $k \in \mathbb{N}$. If the token is in vertex $v \in V_\Box$ then $\Box$ moves it, otherwise his opponent does. The play is winning for $\Diamond$ if the components of the sum of the weights of the edges traversed plus $e_0 \in \mathbb{N}^k$ are strictly above $(0, .., 0) \in \mathbb{N}^k$ during the whole play, otherwise $\Box$ wins. if the play is finite, then the first player who cannot move loses. We define here a strategy of a player $x \in \{\Box, \Diamond\}$ like in the simulation game. A player wins the Z-reachability game if he has a strategy that ensures winning, whatever his opponent's strategy is. Chaloupka proves in [9] that a 2-dimensional Z-Reachability problem (for k = 2) can be solved in $O(|V|^{17})$.

Considering an instance of the problem $UCHS(M, \mathcal{R}, k)$, we build next an equivalent k-dimensional Z-Reachability game of an exponential size.

Theorem 6. *There exist an algorithm that can solve $UCHS(M, \mathcal{R}, 2)$ in $O((norm(M)^n \times |Q_M| \times |\Sigma_M|)^{17})$, with n is the number of states in $\mathcal{R}$.*

Proof. Considering the simulation game $(V_{att}, V_{def}, \delta, v_0)$ associated to M and $\mathcal{R}^\otimes$, note that the only known upper bound of $|V_{att} \cup V_{def}|$ is Ackermanian. However the set $\mathcal{C}_I = \{(c[1], ..., c[n_I(\mathcal{R})]) | c \in \mathcal{C}^M_{\mathcal{R}^\otimes}\}$ has an exponential size (Property 1).

Hence, we consider the weighted graph $(V_\Box, V_\Diamond, E, w_0, e_0, k)$ with: $V_\Box \subseteq Q_M \times \mathcal{C}_I$; $V_\Diamond \subseteq Q_M \times \mathcal{C}_I \times \Sigma_M$; $E \subseteq V \times V \times \{-1, 0, 1\}^k$, with $V = V_\Box \cup V_\Diamond$, $k = n_H(\mathcal{R})$ and:

- for $(q, c) \in V_\Box$ and $(q, a, q') \in \delta_M$, we have $((q, p), (q', p, a), (0, ..., 0)) \in \delta$;
- for $(q, c, a) \in V_\Diamond$ and $(d, a, d') \in \Phi_{\mathcal{R}^\otimes}$ with c and d have the same intermediate components $(c[i] = d[i], i \in [1, n_I(\mathcal{R})])$, we have $((q, c, a), (q, c')) \in \delta$ with $c'[i] = d'[i], i \in [1, n_I(\mathcal{R})]$.
  and $w_0 = (q^0_M, (0, ..., 0)) \in V_\Box$ and $e_0 = (1, .., 1) \in \mathbb{N}^k$.

We consider two mappings:

Let $f : V_{def} \cup V_{att} \mapsto V$ be defined for $q, c, a \in Q_M, \mathcal{C}^M_{\mathcal{R}^\otimes}, \Sigma_M$ as: $f(q, c) = (q, c')$ and $f(q, c, a) = (q, c', a)$ with $c'[i] = c[i]$ for $i \in [1, n_I(\mathcal{R})]$.

Let $g : V^* \mapsto V_d \cup V_{att}$ be defined for $q, c, a \in Q_M, \mathcal{C}^M_{\mathcal{R}^\otimes}, \Sigma_M$ and $w_0, ..., w_l \in V$ as: $g(w_0, ..., w_l, w_{l+1} = (q, c)) = (q, c')$ or $g(w_0, ..., w_l, w_{l+1} = (q, c, a)) = (q, c', a)$

and for each $i \in [1, n_I(\mathcal{R})]$ and $j \in [1, n_H(\mathcal{R})]$, $c'[i] = c[i]$ and $c'[n_I(\mathcal{R}) + j]$ is equal to the j^{th} component of the sum of weights of the edges traversed in the path $\{w_0, ..., w_{l+1}\}$.

Let $S_\Diamond$ be a winning strategy of $\Diamond$ in the Z-Reachability game $(V_\Box, V_\Diamond, E, v_0, e_0, k)$. We build next a winning strategy S_d for the defender in the simulation game $(V_{att}, V_{def},\ \delta, v_0)$. Let $v_0, ..., v_l \in V_{att} \cup V_{def}$ be a path in the simulation game and $w_i = f(v_i)$ for each $i \in [1, l]$ and $w_{l+1} = S_\Diamond(w_0, ..., w_l)$. We take $S_d(v_0, ..., v_l) = g(w_0, .., w_{l+1})$ because by construction we have $(v_l, g(w_0, .., w_{l+1})) \in \delta$. Hence, if $S_\Diamond$ is the winner, then so is the defender.

Reciprocally, we take S_d a winning strategy of the defender in the simulation game and we build $S_\Diamond$, a winning strategy of $\Diamond$ in the Z-Reachability game. Let $w_0, ..., w_l \in V$ be a path in the Z-Reachability with the sum of weights of the edges traversed in the path $\{w_0, ..., w_l\}$ is superior to $(0, ..., 0)$. Considering $v_i = g(w_0, ..., w_i)$ for each $i \in [1, l]$ and $v_{l+1} = S_d(v_0, ..., v_l)$, we take $S_\Diamond(w_0, ..., w_l) = f(v_{l+1})$.

Hence we conclude that there is simulation between M and $\mathcal{R}^\otimes$ **iff** $\Diamond$ wins the Z-Reachability game. Since for k=2, this is decided in $O(|V|^{17})$ [9] and $|V| \leq norm(M)^{n_I(\mathcal{R})} \times |Q_M| \times |\Sigma_M|$, we conclude the result. □

We conclude in the next corollary the Exptime-completeness of $UCHS(M, \mathcal{R}, 2)$.

Corollary 2. *$UCHS(M, \mathcal{R}, 2)$ is Exptime-complete.*

Proof. First, $UCHS(M, \mathcal{R}, 2)$ is Exptime according to Theorem 6. Second, it is harder then $UCHS(M, \mathcal{R}, 0)$ which is Exptime-hard (Theorem 4). Hence, $UCHS(M, \mathcal{R}, 2)$ is Exptime-complete. □

6 Conclusion

In this paper we have considered two parameters that are source of complexity of the web services composition problem. We have shown that among the considered problems, several instances remain Exptime-complete when a parameter (number of hybrid states or parallel instances) is bounded. It remains an open question to identify the complexity of $UCHS(M, \mathcal{R}, 3)$. [7] proves in the context of Z-Reachability that the problem is k-Exptime. This complexity is quite far from the known lower bound, i.e. Expspace hardness.

References

1. Abramowitz, M., Stegun, I.A.: Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables. Dover, ninth dover printing, tenth gpo printing edition (1964)
2. Alonso, G., Casati, F., Kuno, H., Machiraju, V.: Web Services: Concepts, Architectures and Applications. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-662-10876-5

3. Benatallah, B., Casati, F., Toumani, F.: Web service conversation modeling: a cornerstone for e-business automation. IEEE Internet Comput. **08**(1), 46–54 (2004)
4. Benatallah, B., Casati, F., Toumani, F.: Representing, analysing and managing web service protocols. Data Knowl. Eng. **58**(3), 327–357 (2006)
5. Berardi, D., Calvanese, D., De Giacomo, G., Hull, R., Mecella, M.: Automatic composition of transition-based semantic web services with messaging. In: VLDB, pp. 613–624 (2005)
6. Berardi, D., Calvanese, D., De Giacomo, G., Hull, R., Mecella, M.: Automatic composition of web services in colombo (2005)
7. Brázdil, T., Jančar, P., Kučera, A.: Reachability games on extended vector addition systems with states. In: Abramsky, S., Gavoille, C., Kirchner, C., Meyer auf der Heide, F., Spirakis, P.G. (eds.) ICALP 2010. LNCS, vol. 6199, pp. 478–489. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14162-1_40
8. Bultan, T., Fu, X., Hull, R., Su, J.: Conversation specification: a new approach to design and analysis of e-service composition. In: WWW 2003. ACM (2003)
9. Chaloupka, J.: Z-reachability problem for games on 2-dimensional vector addition systems with states is in P. In: Kučera, A., Potapov, I. (eds.) RP 2010. LNCS, vol. 6227, pp. 104–119. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15349-5_7
10. Courtois, J.-B., Schmitz, S.: Alternating vector addition systems with states. In: Csuhaj-Varjú, E., Dietzfelbinger, M., Ésik, Z. (eds.) MFCS 2014. LNCS, vol. 8634, pp. 220–231. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-44522-8_19
11. Ragab Hassen, R., Nourine, L., Toumani, F.: Protocol-based web service composition. In: Bouguettaya, A., Krueger, I., Margaria, T. (eds.) ICSOC 2008. LNCS, vol. 5364, pp. 38–53. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-89652-4_7
12. Henzinger, M.R., Henzinger, T.A., Kopke, P.W.: Computing simulations on finite and infinite graphs. In: Proceedings of 36th Annual Symposium on Foundations of Computer Science, pp. 453–462. IEEE (1995)
13. Jedrzejowicz, J., Szepietowski, A.: Shuffle languages are in p. Theor. Comput. Sci. **250**(1–2), 31–53 (2001)
14. Lasota, S.: Expspace lower bounds for the simulation preorder between a communication-free petri net and a finite-state system. Inf. Process. Lett. **109**(15), 850–855 (2009)
15. Muscholl, A., Walukiewicz, I.: A lower bound on web services composition. Logical Methods Comput. Sci. **4**(2), 1–14 (2008)

GPU Computations and Memory Access Model Based on Petri Nets

Anna Gogolińska[✉], Łukasz Mikulski, and Marcin Piątkowski

Faculty of Mathematics and Computer Science, Nicolaus Copernicus University, Toruń, Toruń, Poland
{anna.gogolinska,lukasz.mikulski,marcin.piatkowski}@mat.umk.pl

Abstract. In modern systems CPUs as well as GPUs are equipped with multi-level memory architectures, where different levels of the hierarchy vary in latency and capacity. Therefore, various memory access models were studied. Such a model can be seen as an interface abstracting the user from the physical architecture details. In this paper we present a general and uniform GPU computation and memory access model based on bounded inhibitor Petri nets (PNs). Its effectiveness is demonstrated by comparing its throughputs to practical computational experiments performed with the usage of Nvidia GPU with CUDA architecture.

Our PN model is consistent with the workflow of multithreaded GPU streaming multiprocessors. It models a selection and execution of instructions for each warp. The three types of instructions included in the model are: the arithmetic operation, the access to the shared memory and the access to the global memory. For a given algorithm the model allows to check how efficient the parallelization is, and whether a different organization of threads will improve performance.

The accuracy of our model was tested with different kernels. As the preliminary experiments we used the matrix multiplication program and stability example created by Nvidia, and as the main experiment a binary version of the least significant digit radix sort algorithm. We created three implementations of the algorithm using CUDA architecture, differing in the usage of shared and global memory as well as organization of calculations. For each implementation the PN model was used and the results of experiments are presented in the work.

Keywords: Petri nets · CUDA architecture · GPU · Memory model

1 Introduction

The inter-process communication over a common part of the memory shared by processes is a usual performance bottleneck in multiprocessor environments. In modern systems CPUs as well as GPUs are equipped with multi-level memory

This research has been supported by the Polish National Science Center through grant No. 2013/09/D/ST6/03928.

architectures, where different levels of the hierarchy vary in latency and capacity. By considering different local views of the processes on the common part of the memory one can try to improve the processor utilization. Therefore, various memory access models were studied, see for instance [8,12,17]. Such a model can be seen as an interface abstracting the user from the physical architecture details. It allows to specify, without a reference to processors, the local views that are possible in concurrent task executions and maintain its consistency.

Another important issue is the task distribution between threads and CPU/GPU cores and the instruction scheduling, which can have a significant impact on the efficiency. Consider for instance running the three threads on a single processor depicted in Fig. 1. Each of them performs a list of arithmetic operations interleaved by memory reads/writes consisting of a short preprocessing and then a longer period of waiting for the memory access (which is a usual situation in parallel computing). In the initial part of the computation each thread realizes its preprocessing for the memory access and then starts to wait for the access itself. This causes the processor idle period when all threads are waiting (marked as I). After that, the arithmetic operations of all threads are executed simultaneously. They can be scheduled in such a way that one thread waits for the memory access, while other threads perform their computations and there is no idle period (marked as II). Thanks to that, the waiting period of a thread can be hidden behind the active computations of other threads.

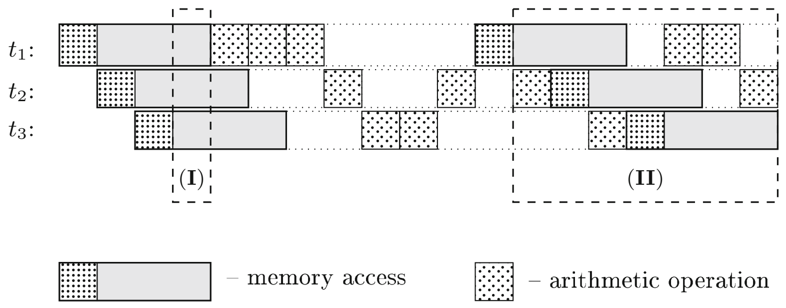


Fig. 1. The example run of the threads t_1, t_2 and t_3 on a single processor. The area marked with (I) represents the idle period when no computation is done, and the area marked with (II) represents the period when the waiting for the memory access is hidden behind arithmetic operations.

The main contribution of this paper is a general and uniform GPU computation and memory access model based on bounded Petri nets [16] together with the application simulating its execution. For a given algorithm pseudocode (or the source code) one can use our model to count the number of operations performed (taking into account their simultaneous execution). The main advantage of the model over the basic arithmetic counting and other models described below is the possibility of the reorganization of the source code fragments, as well as the potential ability to predict a duration of GPU calculations and to handle the aforementioned computation scheduling. Such an approach might be used to improve the algorithm code organization and to partition the computation tasks between the threads to maximally increase the efficiency.

The effectiveness of our model is demonstrated by comparing its throughputs to practical computational experiments performed with the usage of Nvidia GPU. We study the impact on the complexity of various parameters such as the number of concurrent processes, and the level of memory used (shared memory vs. global memory). The successful application of our model is discussed in details, as a proof of concept, on a single example of digit radix sorting algorithm. We do not want to discuss methods of parallelization, nor the most efficient way of using different types of the memory. It is beyond the scope of this paper. Those problems are very complex, and cannot be explained in such a short publication. More about those topics can be found for instance in [6,23].

Our PN model is not the only GPU efficiency model. There are quite a few other GPU performance models, which can be divided into two groups (according to [13]):

(1) Calibrated performance models that make specific predictions and include many lower levels of details. They usually contain many specialized parameters, some of which may be difficult to obtain or calculate. The first example is a model presented in [7]. It is the first analytical model of the GPU efficiency. The most important values used there are MWP (number of memory requestes that can be executed concurrently) and CWP (number of warps, which can be computed while one warp is waiting for memory values). The model consists of 14 equations, which contains 21 parameters. Other example is a model presented in [22]. The authors created not only performance GPU model but also power consumption model. In the performance part they estimate execution time for individual GPU architecture components (e.g. shared, global memory) separately. This way they easily identify potential performance bottlenecks. The model has a form of equations and contains 32 parameters. In [26] authors used stochastic Petri nets to analyse the execution time of MapReduce model on GPU clusters. Their results are very promising, however they are highly specific and limited only to the considered model.

(2) Asymptotic models for algorithm analysis at a high level of abstraction that capture only the essential features of GPU architecture. One of the models from this group is the model described in [15]. He used elements of PRAM, BSP and QRQW approaches. The model calculates the number of cycles required for the whole kernel, taking into account the number of blocks, warps and threads, the maximal number of cycles required by a single thread to perform calculations and the number of threads which can be executed in parallel. However the model is quite simple and some important elements are neglected (like hiding memory latency in computations). The other asymptotic model is Thread Multi-Core Memory (TMM) model presented in [12]. Basing on the TMM, GPUs can be presented as abstract core groups, each containing a number of cores and fast local memory. The large and slow global memory is shared by all cores. The

running time of the algorithm is calculated basing on suitable equation with basic GPU parameters. In [14] authors used colored Petri nets to combine both approaches to GPU computations modelling. They approach is different then ours, e.g. they represented warps as tokens. Our model belongs to the asymptotic group.

One of the most popular GPU efficiency models is the roofline model introduced in [25]. It can be used to obtain performance estimates of GPU computations, and requires two parameters: the number of operations performed by a kernel and the number of bytes transferred from/to the memory, which can be used to calculate the arithmetic intensity I. In the naive roofline model I can be handily presented as a point in two-dimensional space restricted by two ceilings lines: the memory bandwidth and the processor's peak efficiency. The resulting performance is a bound under which the arithmetic intensity appeared: the memory bandwidth bound or the peak performance bound. In the extended versions of the model, additional ceilings can be added. They are related to software prefetching or task level parallelism. The roofline model can determine the type of kernel limitation and show, how optimal the program is. However, such a simple arithmetic operation cannot precisely represent the complex processes of kernel execution, like for example hiding the waiting period in calculations (see Fig. 1). Similar problem occurs in other purely arithmetical models. More complex tools are necessary for such a purpose.

Our model is one of the first utilizing Petri nets. Other Petri nets based models [14,26] where developed in parallel.

The paper is organized as follows. In the next section we describe Graphical Processing Units and CUDA Toolkit, focusing in particular on memory types (and organization). In Sect. 3 we recall some standard notions and notations related to Petri nets. Then we introduce our memory access and computation model followed by the description of radix sort algorithm. We also present the results of experiments conducted on the base of our implementations of the radix sort algorithm. We conclude the paper and give some directions of further research in the last section.

2 CUDA

An intensive development of Graphical Processing Units (GPU in short) resulted in construction of high performance computational devices, which besides the graphical display management, allow also the execution of parallel general purpose algorithms (not necessarily related to computer graphics). As opposed to CPU consisting of a few cores optimized for sequential serial processing, GPU contains thousands of smaller, more efficient cores designed for handling multiple tasks simultaneously. The most popular ones are GPU's produced by Nvidia Corporation, supplied with Nvidia CUDA Toolkit (see [2]).

A program running in heterogeneous environment equipped with GPU can be split into so-called *host* parts, which are executed by CPU, and so-called *kernel* parts, which are executed by GPU. The host part specify the kernel execution context and manages the data transfer between the host and the GPU memory. The kernel functions create a big number of threads allowing a highly parallel computation (where each thread runs the same kernel code). GPU threads, as opposed to CPU threads, are much lighter, hence their creation and controlling requires less CPU cycles.

All threads executed on GPU are organized into several equally-sized thread blocks, which in turn are organized into a grid.

A thread block is the set of concurrently executed threads. Such an execution and the thread cooperation can be coordinated by a barrier synchronization. Moreover, the data can be exchanged between threads using a shared memory. The size of a block is limited by the capacity of resources accessible on a single processor core. On currently available GPU's a single block can contain up to 1024 threads. Each thread block within a grid is uniquely identified by its block ID, and each thread within a block – by its thread ID.

A grid is an array of thread blocks executing the same kernel. The number of blocks in a grid is specified by the amount of data to be processed and the number of available processors. The exchange of data between threads within a grid requires the usage of the global memory. Moreover, while all threads within a single block run simultaneously, different blocks can be executed in any order.

The physical Nvidia GPU architecture consists of several multithreaded streaming multiprocessors (SM in short). Thread blocks are distributed between SM's in such a way that all threads within a single block are concurrently executed on the same SM (different blocks may, but not necessarily have to, be executed on the same SM). A streaming multiprocessor organizes threads into so-called *warps* consisting of 32 threads each. The partition is done according to increasing thread ID. Each SM works utilizing SIMT (single-instruction, multiple-threads) architecture, which means that all threads within a single warp execute one common instruction at a time. Any divergence (e.g. caused by a data processed) leads to a serial execution of a single computation path until possible convergence to the same execution path.

A single SM serves multiple warps. It is equipped with a number of warp schedulers and instruction dispatch units (currently 2 or 4 depending on the device used). The scheduler selects a warp, which is ready to be executed, and issues it to the physical cores of GPU. If the currently active warp needs to wait for the memory read/write operation, it is replaced by another ready warp. While the replaced warp is waiting for the memory access, other warps perform their computations, therefore the SM is busy as often as possible. The waiting period of a single warp is hidden behind the computations of the others (if there are only enough active warps available) and is not seen outside the SM. The simulation of such a behavior is the main part of our model.

The above mentioned heterogeneous environment is equipped with the *host memory* managed by CPU and the GPU *device memory*. The latter is significantly more complex than the former. Due to a necessary compromise between the data transfer/access speed and the possible capacity, the GPU device memory consists of various types of data storage, such as global, constant, local and shared memory.

The global memory is the largest and at the same time the slowest type of GPU memory (with hundreds of cycles latency). Together with the constant memory it is the only type of GPU memory, which can be accessed by the host. It is available for reading and writing for all running threads, however the data exchange and result sharing are possible only after a kernel-wide global synchronization.

The content of the global memory is accessed in blocks of size 32, 64 or 128 bytes (depending on the device used). Every time an element within a block is accessed, the whole block has to be transferred. Therefore, the concurrent (among the threads within a single warp) global memory read and write operations are grouped into transactions, the number of which depends on the cache lines required to serve all threads within a warp. However, if different threads in a warp refer to different memory blocks, all such blocks have to be transferred to cache sequentially. Such a situation cause the necessity of repeated global memory accesses.

The shared memory is a fast memory physically placed inside a multiprocessor. It consists of blocks, each of which is available for all threads within a single thread block. Moreover, each such block is divided into several so-called memory banks. The access of different threads to different banks is realized simultaneously, while the access of different threads to the same bank is realized sequentially. Such a situation is called the *bank conflict*. The shared memory can be used for data exchange between threads within the same thread block after block-wide thread synchronization.

The local memory of a single thread consists of a number of registers, which are the fastest type of memory available (with almost negligible access time). It is used to store local thread variables. Due to large number of threads the capacity of each thread local memory is strongly limited.

To complete the picture, we have to mention also the constant and texture memory – dedicated parts of the GPU device memory (usually buffered). Both of them are optimized for access speed within the device, but are available for threads in read-only mode. Neither of them is considered in our model.

3 Petri Nets

The set of non-negative integers is denoted by $\mathbb{N}$. Given a set X, the cardinality (number of elements) of X is denoted by $|X|$, the powerset (set of all subsets) by 2^X – the cardinality of the powerset is $2^{|X|}$. Multisets over X are members of $\mathbb{N}^X$, i.e., functions from X into $\mathbb{N}$. For convenience and readability, if the set X is finite, multisets in $\mathbb{N}^X$ will be represented by integer vectors of dimension $|X|$ (assuming a fixed ordering of the set X). The addition and the partial order $\leq$ on $\mathbb{N}^X$ are understood componentwise, while $<$ means $\leq$ and $\neq$.

Let us now recall basic definitions and facts concerning inhibitor Petri nets [1, 19] and coloured Petri nets [10].

Definition 1. *An inhibitor*[1] *place/transition net (p/t-net) is a quintuple* $S = (P, T, W, I, M_0)$*, where:*

- P *and* T *are finite disjoint sets, of places and transitions (actions), respectively;*
- $W : P \times T \cup T \times P \to \mathbb{N}$ *is an arc weight function;*
- $I \subseteq P \times T$ *is an inhibition relation;*
- $M_0 \in \mathbb{N}^P$ *is a multiset of places, named the initial marking.*

For all $a \in T$ we use the following denotations:

$^{\bullet}a = \{p \in P \mid W(p, a) > 0\}$ – the set of *entries* to a,
$a^{\bullet} = \{p \in P \mid W(a, p) > 0\}$ – the set of *exits* from a,
$^{\circ}a = \{p \in P \mid (p, a) \in I\}$ – the set of *inhibitor places* for a,
$W(P, a) \in \mathbb{N}^P$, where $W(P, a)(p) = W(p, a)$,
$W(a, P) \in \mathbb{N}^P$, where $W(a, P)(p) = W(a, p)$.

Petri nets admit a natural representation as bipartite graphs, in which places are indicated by circles, and transitions by boxes. Arcs with classical arrow heads represent the weight function, while arcs with small circles as arrowheads represent inhibition relation.

The set of all finite strings of transitions is denoted by T^*, the empty string is denoted by ε, the length of $w \in T^*$ is denoted by $|w|$, the number of occurrences of a transition a in a string w is denoted by $|w|_a$.

Multisets of places are called markings. In the context of p/t-nets, they are typically represented by nonnegative integer vectors of dimension $|P|$, assuming that P is totally ordered. Markings are depicted by tokens inside the circles, the capacity of places is unlimited. However, Petri nets used in our model are bounded (which means that there exist a common bound for all the numbers of tokens appearing during the computation in a single place).

A transition $a \in T$ is enabled at a marking M whenever $W(P, a) \le M$ (all its entries are marked) and $\forall_{p \in {}^{\circ}a}\ M(p) = 0$ (all inhibitor places are empty). If a is enabled at M, then it can be executed. A marking M is called a *dead marking* if no transition is enabled at M (which means that $\forall_{a \in T} \exists_{p \in P} \big(W(p, a) > M(p) \lor ((p, a) \in I \land M(p) > 0)\big)$). The execution of an enabled transition a is not forced and changes the current marking M to the new marking $M' = M - W(P, a) + W(a, P)$ (tokens are removed from entries, then put to exits). We shall denote Ma for the predicate "a is enabled at M" and MaM' for the predicate "a is enabled at M and M' is the resulting marking".

In this paper however, we use the maximal concurrent semantics and in every marking we execute one of the maximal sets of enabled transitions (i.e. a step, which is maximally concurrent at this marking). Formally, a set of transitions $A \subseteq T$ is called *step* and is enabled if $\big(\sum_{a \in A} W(P, a)\big) \le M$ and

[1] Note that in the case of bounded nets the use of inhibitors is not necessary, one can provide an equivalent (with more complex structure) net without inhibitors.

$\forall_{p \in (\bigcup_{a \in A} {}^{\circ}a)}\ M(p) = 0$. The execution of a step A changes the current marking M to the new marking

$$M' = \Big(M - \sum_{a \in A} W(P,a)\Big) + \sum_{a \in A} W(a,P).$$

We say that a step is *maximally concurrent at marking* M if A is enabled at M and $\forall_{a \notin A} A \cup \{a\}$ is not enabled at M.

The notions of enabledness and execution can be extended, in a natural way, to strings of steps (*computations*): the empty string ε is enabled at any marking, a string $w = Av$ is enabled at a marking M whenever MAM' and v is enabled at M'. The predicates MA, Mw, MAM' and MwM' are defined like for single transitions.

Another system model used in this paper are coloured Petri nets defined as

Definition 2 ([10]). *A (non-hierarchical)* coloured Petri net *is a nine-tuple* $CPN = (P, T, A, \Sigma, V, C, G, E, I)$*, where:*

- P *and* T *are finite, disjoint sets of* places *and* transitions *(similar to the case of inhibitor nets);*
- $A \subseteq P \times T \cup T \times P$ *is a set of* directed arcs*;*
- Σ *is a finite set of non-empty* colour sets*;*
- V *is a finite set of* typed variables *such that* $Type(V) \in \Sigma$ *for all variables* $v \in V$*;*
- $C : P \to \Sigma$ *is a* colour set function *that assigns a colour set to each place;*
- $G : T \to EXPR_V$ *is a* guard function *that assigns a guard to each transition* t *such that* $Type[G(t)] = Bool$*;*
- $E : A \to EXPR_V$ *is an* arc expression function *that assigns an arc expression to each arc* a *such that* $Type[E(a)] = \mathbb{N}^{C(p)}$*, where* p *is the place connected to the arc* a*;*
- $I : P \to EXPR_\emptyset$ *is an* initialisation function *that assigns an initialisation expression to take each place* p *such that* $Type[I(p)] = \mathbb{N}^{C(p)}$*.*

Note that, according to the utilised CPN Tools [24], $EXPR$ is the set of *net inscriptions* (over a set of variables V or over an empty set, i.e. using only constant values) provided by CPN ML. Moreover, by $Type[e]$ we denote the type of values obtained by evaluation expression e. The set of *free variables* in an expression e is denoted by $Var[e]$, while the type a variable v – by $Type[v]$. The setting of the particular value to free variable v is called a *binding* $b(v)$, we require that $b(v) \in Type[v]$ and denote with the use of $\langle\rangle$ filled by the list of valuations and written next to the element to whom it relates. The set of bindings of a transition t is denoted by $B(t)$. The *binding element* is a transition t together with a valuation $b(t)$ of all the free variables related to t. We denote it by (t, b), where $t \in T$ and $b \in B(t)$.

A *marking* M in coloured Petri nets is a function which assigns to each $p \in P$ a multiset of tokens $M(p) \in \mathbb{N}^{C(p)}$. An initial marking is denoted by M_0 and defined for each $p \in P$ as follows: $M_0(p) = I(p)\langle\rangle$.

A binding element (t, b) is enabled at a marking M if $G(t)\langle b\rangle$ is true and at each place $p \in P$ there is enough tokens at M to fulfil the evaluation of the arc expression function $E(p, t)\langle b\rangle$. The resulting marking is obtained by removing from $M(p)$ the tokens given by $E(p, t)\langle b\rangle$ and adding those given by $E(t, p)\langle b\rangle$ for each $p \in P$.

4 Memory Access and Computations Model

The Petri net model of GPU calculations is consistent with the workflow of multithreaded streaming multiprocessors (SMs). The model represents the way one SM operates. It models each warp assigned to the SM, selection of the next instruction for the SM, accesses to the global and shared memory, and arithmetic operations. It does not represent threads hierarchy (blocks, grid), repeated accesses to the global memory nor bank conflicts. The model allows simultaneous accesses to the global memory, but the number of warps, which can use the global memory, at the same time, is limited by the number of SM warp schedulers (2 or 4). Each element of the model was created basing on [3,4].

The size of the considered Petri net depends on the number of warps. The two elements: the place p_0 – *SM* and the transition t_0 – *waiting* are the constant part of the model, other are generated for every warp. The place p_0 represents the streaming multiprocessor and its initial marking should correspond to the number of warp schedulers. For the modern graphical cards it should be 2 or 4. This place is connected by a loop with the transition t_0. The transition t_0 can be executed only when the warp schedulers cannot schedule any instruction, i.e. there is no instruction ready to be executed. The *waiting* transition is added to the model to gain control over how many steps of the calculations on the SM is idle. The minimization of that number is crucial for optimization of GPU programming. Besides those two elements, the PN model contains 18 places and 17 transitions for every warp required by the analysed algorithm. That part is called a warp part of PN model (WPNM). The detailed description of the most important places and transitions of WPNM is presented in Table 1.

The WPNM together with *SM* place and *waiting* transition are depicted in Fig. 2. The place p_1 represents the activation of the warp. It is marked when the warp is active. The place p_2 is marked when the warp finishes the execution of its previous instruction. The warp can be scheduled for the execution only when the places p_2 and p_4 are marked. A token in p_4 means that the next instruction is selected and ready. The places from p_3 to p_{10} and the transitions t_2, t_3, t_4 (the frame I part in Fig. 2) are responsible for controlling and selecting the instructions. The three types of instructions are allowed in the model: an arithmetic calculation, an access to the shared memory and an access to the global memory. The initial marking of the place p_5 corresponds to the number x_a of arithmetic operations required by the analyzed algorithm. Similarly, the initial markings of the places p_7 and p_9 represent respectively the numbers x_s and x_g of read/write operations from/to the shared and global memory. If the place p_3 is marked and at least one of the places: p_5, p_7, p_9 is not empty, the next instruction can be selected. The selection is random (if more than one of the places: p_5, p_7, p_9 is marked then a token is taken from the randomly chosen one – the distribution of the probability is uniform). When the instruction is chosen, according to its type (arithmetic calculation, shared memory access, global memory access) the marking of the corresponding place is decreased and a token is added to p_4. Moreover, (according to the instruction type) one of the places: p_6, p_8 or p_{10} is marked. Now the selected instruction is ready to be executed and the warp can be processed by SM. The execution of the instruction is represented by the three parts of the net marked in frame II, frame III and frame IV (see Fig. 2). They correspond to the type of the selected instruction: frame II for an arithmetic calculation, frame III – an access to the shared memory and frame IV – an access to the global memory. The arithmetic calculation is simply represented by one place and two transitions. When the calculation is done, tokens are put in places: p_2 and p_3 (which means that the instruction is finished and the next one can be selected), and in the place p_0 (which means that SM is ready to execute the next instruction). The same situation is obtained when the access to the memory (shared or global) is finished, but those parts of the model contain more places and transitions. Those additional elements are used to model the memory access latencies. The shared memory latency and the global memory latency are the parameters of the model and are denoted by l_1 for the global memory and by l_2 for the shared memory. The transition t_9 (t_{12} respectively) may be executed only after l_1 (l_2 respectively) executions of t_8 (t_{17} respectively). For testing, their default values were 20 for l_1 and 2 for l_2, which is consistent with [4].

The transition t_{16} is connected by inhibitor arcs with the places p_5, p_7 and p_9 (the frame I in Fig. 2) and can be executed only when those places are empty, i.e. there is no instruction left for execution. The transition is also connected by a regular arc with p_2, Moreover, t_{16} is the only one able to take the token from the place p_1 and its execution is equivalent to the termination of a given warp.

As it was mentioned above, the WPNM (places from p_1 to p_{18} and transitions from t_1 to t_{17}) is generated for a single warp. In the case of multiple warps a separate WPNM should be generated for each of them. To distinguish between distinct WPNMs one can either increase the numeration of places and transitions accordingly or assign to them two-part labels consisting of the original place/transition number together with the warp id. It should be noticed that each place corresponding to p_2 and representing the readiness of the given warp should be connected by an inhibitor arc with the transition t_0. Moreover, each transition corresponding to t_1 should be connected with the place p_0 (SM). The same goes for transitions corresponding to t_5, t_{14} and t_{15}. Note that the control is returned by t_5 not t_{11} in the case of part responsible for the access to the global memory. Thanks to that, SM can process the next ready warp while the current one is waiting for the memory access.

Our PN model of GPU computation and memory access may be adapted for any algorithm. Instantiations of the model for different kernels may differ in the number of warps and the marking of places responsible for instructions

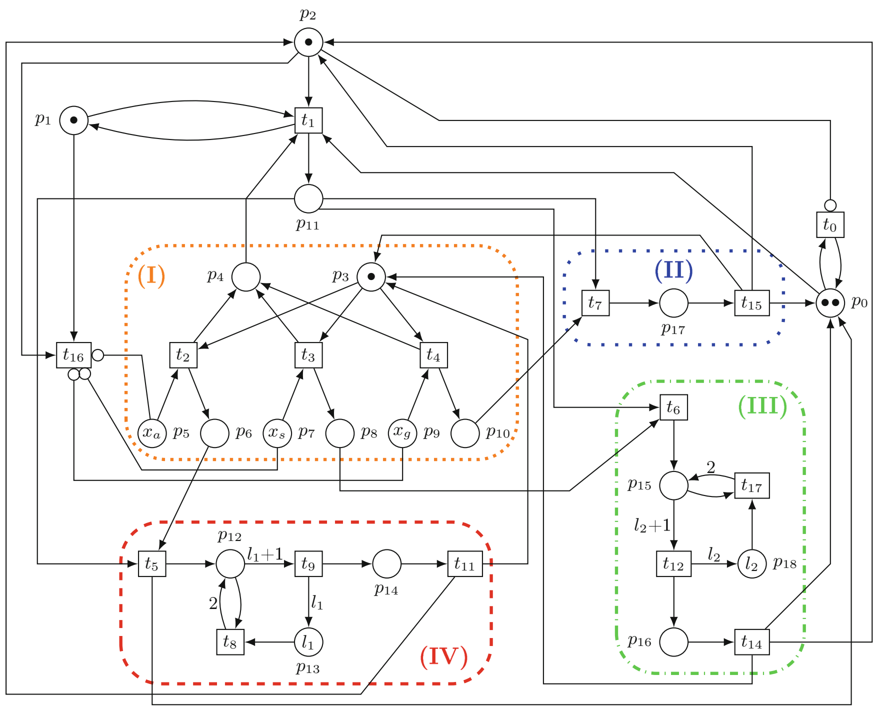


Fig. 2. The example of a warp PN model (WPNM). In frame I part the next instruction is selected. Sections: frame II, frame III and frame IV represent the execution of different types of instructions: arithmetic calculation, access to the shared memory and access to the global memory (respectively).

counting (i.e. p_5, p_7 and p_9). For the chosen number of required warps, a new model containing sufficient number of WPNMs can be generated. The other possibility is the generation of one big model with the maximal possible number of WPNMs (i.e. 64 for modern graphical cards [4]), and then the necessary number of places representing active warps should be marked. The number of WPNMs can be calculated basing on the number of threads and blocks, which are parameters of the kernel. Notice that the model provides no controlling mechanism for the maximum number of WPNMs. It is up to the user to be aware that the maximal number of WPNMs is limited to 64 in modern GPUs. The number of arithmetic operations and accesses to the shared and global memory need to be calculated for the considered algorithm. Those numbers should be used as the initial marking of places p_5, p_7 and p_9. If the model is constructed according to the description above, it is ready to be used out of the box. The usage of the model involves the execution of computations according to the maximal concurrent semantics (i.e. concurrent execution of all transitions that are enabled) starting from the initial marking until reaching the dead marking. The latter is obtained only when all places corresponding to p_1 (for each WPNM)

Table 1. The description of the places and transitions depicted in the Fig. 2.

Place/transition name and description			
p_0	Streaming Multiprocessor (SM)	t_0	Waiting
p_1	Active warp	t_1	The warp is executed on SM
p_2	The previous instruction is finished and the warp is ready for the next one	t_2	Selection of a calculation for the next instruction
p_3	Check the next instruction	t_3	Selection of an access to the shared memory for the next instruction
p_4	Next instruction is ready	t_4	Selection of an access to the global memory for the next instruction
p_5	Arithmetic operations	t_5	Execution of the access to the global memory
p_6	Instruction – arithmetic calculation	t_6	Execution of the access to the shared memory
p_7	Access to the shared memory	t_7	Execution of arithmetic operation
p_8	Instruction – shared memory access	t_8	Waiting for the global memory access
p_9	Access to the global memory	t_9	Access to the global memory
p_{10}	Instruction – global memory access	t_{11}	Access to the global memory is finished
		t_{12}	Access to the shared memory
		t_{14}	Access to the shared memory is finished
		t_{15}	Calculation is finished
		t_{16}	Termination of the warp
		t_{17}	Waiting for the shared memory access

become empty, which is equivalent to the termination of all warps. The number of steps of the computation is returned by the model and corresponds to the GPU execution time.

The maximal concurrent semantics requires the execution of all enabled transitions. However, some of the enabled transitions may be in a conflict (i.e. execution of one transition disables another transition). In our implementation of the model, in every step, a permutation of the enabled transitions is randomly generated (using uniform distribution). Transitions are executed according to an order determined by the generated permutation. If two (or more) transitions are in a conflict, the transition appearing earlier in the considered order is executed.

The initial tests of the PN model were performed using the matrix multiplication kernel from [4] and the *stability* example from [23]. The PN models were generated for both kernels. The matrix multiplication program was executed many times with different sizes of matrices. Similarly, the *stability* example was executed with different values of *Time Step* and *Final Time* parameters. For the same data, the computations of the PN model were executed. The execution times of kernels and the numbers of steps of PN computations were compared. The results were consistent in both cases.

4.1 The Version of the Model Based on Colored Petri Nets

In the model presented above, the bounded inhibitor Petri nets were used. This initial model was rewritten with the usage of the colored PNs, which leads to its simplification. The new version of the model is presented in Fig. 3. It is suitable to present arbitrary number of warps. Repetitions of WPNMs are not longer necessary, because every color of a place contains at least one number, labeled W, which describes the warp index. That is why the colored PN model for the whole system is very similar to the model of a single WPNM. The parts corresponding to frames II, III and IV in Fig. 2 can be easily identified. Only small differences in those parts appear in models of memory access latencies. In the inhibitor version, one place and one transition are necessary to model the latency. Using the colored PN semantics it is sufficient to use a single transition for this purpose.

The more significant difference between the inhibitor and colored model is the part corresponding to selection of the next instruction (part I in Fig. 2). In the colored version, this part is strongly reduced. The numbers of instructions, which have to be executed, together with their types and the index of a warp are described by tokens color in place p_7. The type of selected instruction (and the warp id) is transferred to p_8, and then it can be recognized by transitions t_5, t_6 or t_7.

Apart from those differences the colored PN model is in one to one correspondence with the inhibitor model, which was confirmed by tests (see Sect. 5 for details). Example tests results are presented in Fig. 7. Those tests where performed using randomly generated parameters without maximal concurrent semantics. The tests show that there is a strong correlation between both models.

One can also consider the timed PNs, however in our opinion they are not suitable for our model. In this type of PNs time intervals are associated with transitions. Namely, a transition a can be executed only if the net's clock value belongs to interval defined for a. In the model we are interested in the total time of the PN simulation. It is hard to predict when a particular transition may be executed, hence we do not have the sufficient data to establish approximate time intervals of transitions. We do not want to restrain time of transitions executions, we want to execute them freely and then check the total time of computation.

5 Experimental Results

In the main experiment we used a binary version of the least significant digit radix sort algorithm [20,21]. The idea of this method is to sort a list of positive n-bit integers using their binary representation. We make n runs rearranging the list in such a manner that in i-th run all the integers having 0 on i-th bit are arranged in the first part of the array, while those having 1 – in the second part. An important requirement is to preserve the order of elements which do not differ on the processed bit. In other words, the sorting subroutine need to be stable. As a side effect the whole sorting procedure is also stable (Fig. 4).

In the parallel version we made n runs (one for every bit), each run consisting of three phases. At the beginning of each run, we partition the dataset equally between m nodes. During the first phase, j-th node counts $zeros[i, j]$ – the number of elements containing 0 on i-th bit (consequently, we know $ones[i, j]$

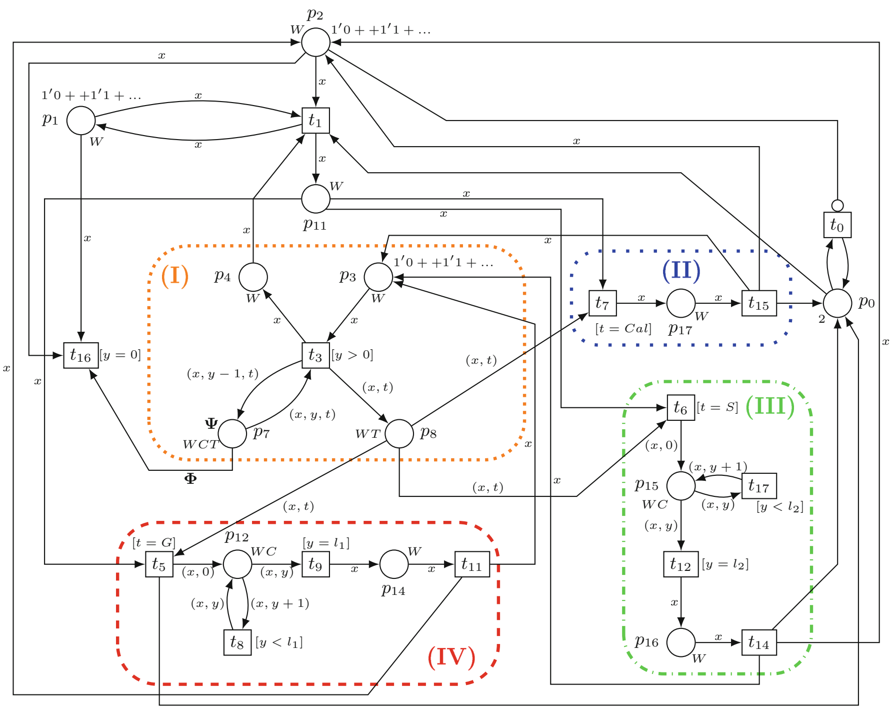


Fig. 3. The CPN model. Frames like in Fig. 2. The special symbols meaning: $\mathbf{\Phi} = 1'(x, y, G) + +1'(x, y, S) + +1'(x, y, Cal)$, $\mathbf{\Psi}$ is an initial marking of p_7 which describes numbers of arithmetic operations, accesses to the global and shared memory for each warp. Places p_1, p_2 and p_3 contain one token for each warp in the initial marking. Colours definitions: $C = int\ with\ 0..1000000$, $W = int\ with\ 0..100000$, $Type = with\ G|S|Cal$, $WC = product\ W * C$, $WT = product\ W * Type$, $WCT = product\ W * C * Type$.

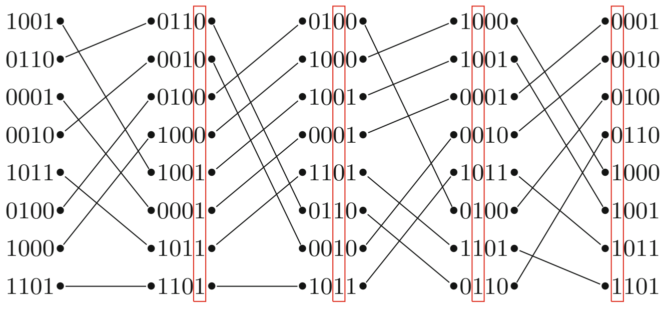


Fig. 4. Example of the use of radix sort procedure for four-bit integers. In consecutive columns we present the lists after each run of sorting subroutine. The rectangles emphasize columns with freshly sorted bits.

– the number of elements with 1 on i-th bit for this node). In the next phase, we need to compute the positions for the set of elements assigned to each node. Namely, j-th node should place all the elements having 0 on i-th bit between $\sum_{k<j} zeros[i,k]$ and $\left(\sum_{k\leq j} zeros[i,k]\right) - 1$, while those with 1 on i-th in the range

$$\Big[\sum_{k\leq m} zeros[i,k] + \sum_{k<j} ones[i,k], \sum_{k\leq m} zeros[i,k] + \big(\sum_{k\leq j} ones[i,k]\big) - 1\Big].$$

The last, third phase, is the rearrangement of the list of integers. Each node traverses the assigned part of the data splitting it into two parts (containing only 0 on i-th bit and only 1 on i-th bit) with the use of the positions computed in the second phase and in a stable manner. Since the output space for the nodes is partitioned into disjoint blocks, this phase may be realized using either shared or global memory.

We consider three CUDA implementations of the algorithm described above. In all versions, the array of integers A to be sorted is stored in the global memory. During each kernel execution, one block of threads (with different number of threads) is created. Each thread has its own part of the array A assigned. Its size is the parameter and is denoted by $memsize$. The product: $threadsNumber * memsize$ should be equal to the size of A. At the beginning of each run, every thread copies the assigned part of the array A from the global memory to its local registers, then calculates number of 0 and 1 bits. At the end of the run, the content of the global memory array is rearranged – each thread moves the elements from its part of A.

Each of the three implementations of the radix sort algorithm was tested on a randomly generated array of 65536 integers, with five combinations of $threadsNumber$ and $memsize$ parameters. For the given implementation and the value of parameters, the numbers of arithmetic operations and accesses to the global and shared memory were calculated and used in the PN model (as the initial marking of the places p_5, p_7 and p_9). As an arithmetic operation we count every assignment, addition, subtraction, multiplication, division, relational operation, logical operation and array subscript (arrays in registers). Every access (read or write) to data stored in the global or shared memory is counted as single memory operation.

The numbers of steps of the model calculations were compared to the execution times of kernels. The tests were performed on NVIDIA GeForce GTX 960M graphical card with CUDA Toolkit 8.0. The execution times of kernels are averages of one hundred runs. The results for the PN model were calculated as averages of ten computations of the models.

In the first implementation only the global memory is used. The second phase of the algorithm is performed by thread with id *0*. The results of the tests are depicted in Fig. 5 – the dotted line.

In the second implementation the realization of the second phase is organized in a more efficient way. Instead of computing all sums incrementally, we compute all partial sums (for indexes between $p \cdot 2^q$ and $(p+1) \cdot 2^q$, where p and q are

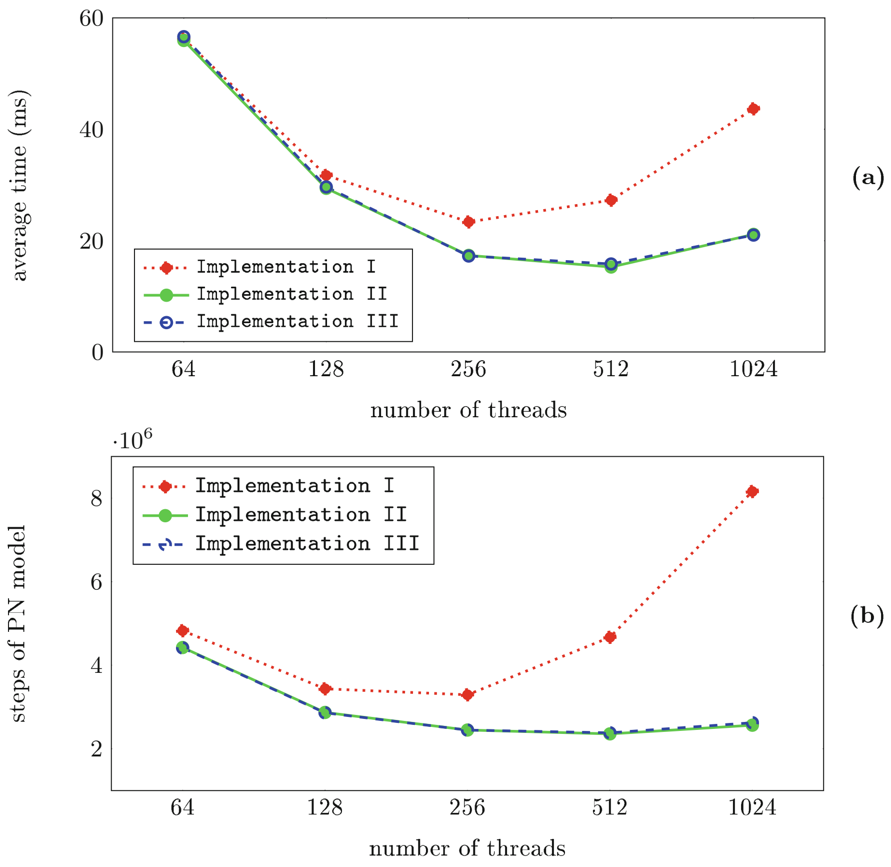


Fig. 5. The results for the radix sort algorithm tests with 65536 element arrays: (a) execution times of kernels (ms), (b) steps of the PN model.

suitable non-negative integers) and use them as input components for other sums.

Having $m = 2^r$ nodes we can compute $zeros[i, j]$ and $ones[i, j]$ for all $j \leq m$ in $r+1$ cycles with full system load (using all nodes in every cycle). To do it, we compute in *c-th* cycle specific partial sums of lengths between 2^{c-1} and $2^c - 1$, an example for $r = 2$ is depicted on Fig. 6. In this sample case $z[x..y]$ denotes $\sum_{x \leq t \leq y} zeros[i, t]$, while $o[x..y]$ – $\sum_{x \leq t \leq y} ones[i, t]$, each row corresponds to a single element in table *zeros* or *ones*, while in subsequent columns the values of partial sums stored in those elements are given. Each arc between $x - th$ and $y - th$ row denotes the addition of value kept in $x - th$ element to the value kept in $y - th$ element, the result is stored in $y - th$ element.

More specifically, in the first cycle we compute

$$zeros[i, 2k] + zeros[i, 2k + 1] \quad \text{and} \quad ones[i, 2k] + ones[i, 2k + 1]$$

placing results in $zeros[i, 2k+1]$ and $ones[i, 2k+1]$ respectively. The number of all operations made in this cycle (the number of arcs between first and second column in example) is $m/2 + m/2 = m$, hence we can utilize all available nodes to do it at once.

In subsequent cycles we compute longer partial sums using already precomputed ones. This way in $c-th$ cycle we compute

$$\sum_{0 \leq t \leq u} zeros[i, 2^c k + t] \quad \text{and} \quad \sum_{0 \leq t \leq u} ones[i, 2^c k + t],$$

where $2^{c-1} \leq u < 2^c - 1$, while $0 \leq k < 2^{r-c} - 1$, and store the results in $zeros[i, 2^c k + t]$ and $ones[i, 2^c k + t]$, respectively. Since after the previous cycle all values $\sum_{0 \leq t \leq u} zeros[i, 2^c k + t]$ and $\sum_{0 \leq t \leq u} ones[i, 2^c k + t]$ are stored in memory, where $2^{c-2} \leq u < 2^{c-1}$, while $0 \leq k < 2^{r-c+1}$, we need to add only two elements for each longer partial sum computed in $c-th$ cycle. Note that the number of such operations equals the size of the range of u multiplied by the size of the range of k and doubled (we need to compute both ones and zeros), i.e.

$$|\{u, 2^{c-1} \leq u < 2^c\}| \cdot |\{k, 0 \leq k < 2^{r-c}\}| \cdot 2 = 2^{c-1} \cdot 2^{r-c} \cdot 2 = 2^r = m.$$

Finally, in the last cycle we add the computed so far $\sum_{k \leq m} zeros[i, k]$ to all $\sum_{t \leq k} ones[i, t]$ for each $k \leq m$. The execution times of kernels and the results from the PN model are depicted in Fig. 5 – the solid line (Fig. 6).

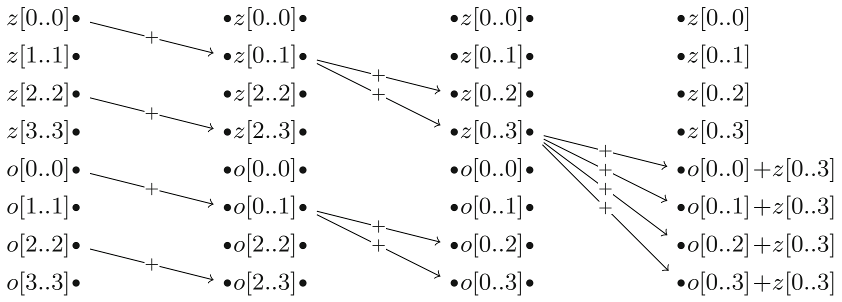


Fig. 6. The organization of the second phase of the algorithm for $m = 4$ (as $m = 2^r$, $r = 2$) nodes in $r + 1 = 3$ cycles.

In the third version of the implementation the arrays *zeros* and *ones* are stored in the shared memory instead of global. The results of those tests are also presented in Fig. 5 – the dashed line.

The results of the PN model calculations for the first implementation (Fig. 5(a)) clearly show that this parallelization is not very efficient as compared to the others, especially for larger numbers of threads. This is confirmed by the execution times of kernels (Fig. 5(b)). One can easily observe that in both plots the number of PN model steps and execution times of kernels for this implementation initially decrease with the increasing number of threads, however for more than 256 threads both of them increase. A similar situation can be also noticed for other implementations, but here the growth is more significant. It can also be observed that for the largest amounts of threads the number of PN model steps increases faster than execution times of kernels. In this case the general direction of changes predicted by our model is consistent with the kernels executions (despite the lack of the exact match of the plots). The model predicted correctly the most efficient choice of the number of threads, which in this case is 256.

The predictions of the PN model for the 2nd and 3rd implementations are more consistent with the execution times of kernels. In both cases differences between implementations are very small. It is probably caused by a relatively small number of shared memory operations in comparison to accesses to the global memory. For the larger number of threads, the increase in execution times of kernels is more significant than in the results from the PN model. The reliable explanation of such a difference is an overhead for communication between threads. It is clear that such overhead will not be observed in the PN model. However, the general characterization of the results is the same both for the model and the kernels. The PN model predicted correctly also the most effective choice of the number of threads for both implementations, which is 512 threads.

5.1 Tests of the CPN Model

The CPN model was created using CPN Tools software [18,24]. The same tool was used to perform the simulation of the model. Unfortunately CPN Tools uses only interleaving semantics and does not support the maximal concurrency [24]. Hence, the tests for the inhibitor model were also repeated using interleaving semantics. The results obtained for the 3rd implementation (number of warps, arithmetic operations, accesses to the shared and global memory) are presented in Fig. 7a. One can easily observe that those results are not the same, as it should

Table 2. Statistics of transitions executions for the 3rd implementation and two warps for both versions of the model.

CPN model		Inhibitor model - warp 1		Inhibitor model - warp 2	
t0	1178789	t0	734992		
t1	131072	t1	65536	t18	65536
t3	131072	t4	65536	t21	65536
t5	131072	t5	65536	t22	65536
t8	1310720	t8	655360	t25	655360
t9	131072	t9	65536	t26	65536
t11	131072	t11	65536	t28	65536
t16	2	t16	1	t33	1

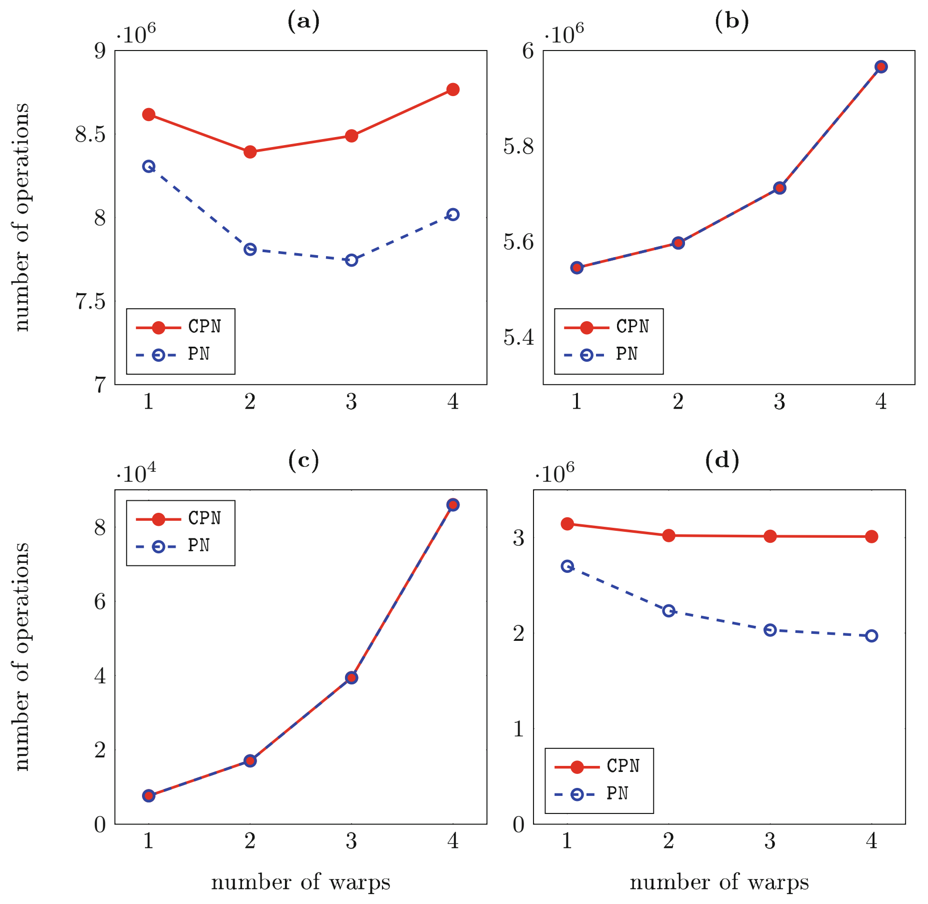


Fig. 7. Comparisons of the CPN model and the inhibitor model. (a) Results for the 3rd implementation - all operations. (b) Results only for a selected type of operations: (b) arithmetic calculations, (c) accesses to the shared memory. (d) accesses to the global memory.

be expected. That is the reason why we needed a more detailed analysis. Separate tests for arithmetic operations, accesses to the global memory and accesses to the shared memory were performed. The results are presented in Fig. 7. It is easy to notice that the result for arithmetic operations and accesses to the shared memory are the same for both versions of the model, however differences can be observed in accesses to the global memory. To find a reason of those differences we analyzed this case in more details. The number of transitions executions for two warps are presented in Table 2.

According to Table 2, transition t_0 was executed more frequently in the CPN model than in the inhibitor model. The numbers of executions of other transitions are coherent. Notice that a single transition in the CPN model (except for

t_0) corresponds to a single transition in each of the two WPNM in the inhibitor model. The number of t_0 executions in both cases differ due to different probability distribution used. When there are only global memory access instructions, very often both warps are waiting for such an access. Then, in the CPN model two transitions will be enabled: t_0 corresponding to the waiting and t_8 corresponding to the global memory latency. Without the maximal concurrency semantics only one of them might be executed, and the probability of choosing t_0 is $\frac{1}{2}$. However, at the same time in the inhibitor model three transitions are enabled: t_0 corresponding to the waiting, t_8 corresponding to the global memory latency for the first warp and t_{25} corresponding to the global memory latency for the second warp. Each of them will be executed would the probability $\frac{1}{3}$. The numbers from the Table 2 corresponded to those observations. Therefore, in the CPN model t_0 is executed more frequently when the interleaving semantics is used, because in conflict situations it is chosen with the higher probability. For more warps in the inhibitor model the probability distributions differ even more. When the maximal concurrency semantics is used the described problem do not occur, because then in every step all the enabled transitions are executed as a single step.

6 Conclusions and Future Work

The purpose of our PN based GPU computations and memory access model is to help in the analysis and optimization of parallel algorithms, which are designed to be implemented on CUDA graphical cards. We do not require the source code to be given as an input, however the algorithm description should be detailed enough to estimate the number of arithmetic operations and accesses to the global and shared memory. Note that the other tools (e.g. the roofline model) also require those information. The expected speedup of the computation from a parallelization can be predicted and compared to other algorithms, even without using any physical GPU device. The model can help to predict which algorithm is the fastest, how much its modifications can affect the speedup of the computation and whether they are significant enough to include them in the source code.

Any inaccurate results demonstrated by our model might be interpreted as a premise that the algorithm should be improved. As an example recall the presented results for the radix sort algorithm. The predictions of the PN model for the first implementation were not satisfying, and the model clearly showed the possibility of reducing the computation time with different organization of the second phase of the algorithm. On the other hand, using the shared memory in this case was not so beneficial. That was confirmed by the GPU kernels execution times.

Another important advantage of the PN model is the possibility to check how different values of parameters and the level of parallelization can affect the final efficiency. As it can be observed in Fig. 5, it is not only a theoretical discussion, the problem is substantial and can result in very different execution times of kernels. With the appropriate number of threads, the GPU calculations were even three times faster. It should also be noticed that for all three implementations presented above, different numbers of threads were the most efficient, hence the selection of the one, universal number of threads is not possible. Our model easily allows to check different number of threads (warps) and its predictions seem to be very accurate. Various values of parameters may also result in different numbers of arithmetic operations and accesses to the memory, and it can be also easily introduced and analyzed by the model.

Our model can freely swap instructions of various types. It allows to check whether different order of the instructions may improve the algorithm efficiency, for example by allowing to hide thread waiting periods in calculations. If the results of the PN model are significantly better than the results from GPU kernels execution, that possibility should be considered. Naturally, the swap of the instructions is not always possible because of the nature of calculations. One of the most important improvements of the model would be the introduction of a partial order over the set of instructions. This can be achieved by defining dependence of instructions basing on the access to the same variable (see [9]). The partial order would make predictions of the model more accurate.

The PN results are expressed as the number of steps performed, while for the GPU computations we use the execution times of kernels in milliseconds. To compare them directly, the special coefficient is required to align one result with the other. However, different nature of various algorithms, as well as the lack of research on atomic and comparable in terms of time consumption operations makes the issue of finding the universal coefficient a very hard task.

Although designing an efficient sorting algorithm was not the aim of this paper, we described the process of improving the parallel version of the considered radix sort algorithm. Nevertheless, it is worth to note that providing its further improvements is possible. The radix sort is quite popular, both in the most significant digit version (MSD), normally together with merge sort subroutine [5], and the least significant digit version (LSD), as suggested in [11].

Note that for different GPU devices the execution time of a fixed kernel may differ, while the number of steps performed by the model remains the same. It would be useful to prepare a set of benchmarks, which for a given device compute the universal scaling coefficient for this device and the model.

References

1. Chiola, G., Donatelli, S., Franceschinis, G.: Priorities, inhibitor arcs and concurrency in P/T nets. In: Proceedings of ICATPN, vol. 91, pp. 182–205 (1991)
2. Nvidia Corporation. CUDA. http://www.nvidia.com/object/cuda_home_new.html
3. Nvidia Corporation. CUDA. Best practice guide version 8.0.61 (2017)
4. Corporation, Nvidia: CUDA. C programming guide version 7.5 (2017)
5. Duvanenko, V.J.: Algorithm improvement through performance measurement (2009). http://www.drdobbs.com/architecture-and-design/algorithm-improvement-through-performanc/220000504
6. Grama, A.: Introduction to Parallel Computing. Pearson Education, London (2003)

7. Hong, S., Kim, H.: An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In: ACM SIGARCH Computer Architecture News, vol. 37, pp. 152–163. ACM (2009)
8. Hwang, K., Jotwani, N.: Advanced Computer Architecture, 3rd edn. McGraw-Hill Education, New York (2011)
9. Janicki, R., Koutny, M.: Structure of concurrency. Theoret. Comput. Sci. **112**(1), 5–52 (1993)
10. Jensen, K., Kristensen, L.M.: Coloured Petri Nets - Modelling and Validation of Concurrent Systems. Springer, Heidelberg (2009). https://doi.org/10.1007/b95112
11. Luebke, D., Owens, J., Roberts, M., Lee, C.-H.: Intro to Parallel Programming - Online Course. Nvidia Corporation, Santa Clara
12. Ma, L., Agrawal, K., Chamberlain, R.D.: A memory access model for highly-threaded many-core architectures. Future Gen. Comput. Syst. **30**, 202–215 (2014)
13. Ma, L., Chamberlain, R.D., Agrawal, K.: Performance modeling for highly-threaded many-core GPUs. In: 2014 IEEE 25th International Conference on Application-Specific Systems, Architectures and Processors (ASAP), pp. 84–91. IEEE (2014)
14. Madougou, S., Varbanescu, A.L., de Laat, C.: Using colored petri nets for GPGPU performance modeling. In: Proceedings of the ACM International Conference on Computing Frontiers, pp. 240–249. ACM (2016)
15. Mukherjee, R.: A performance prediction model for the CUDA GPGPU platform. Ph.D. thesis, International Institute of Information Technology Hyderabad, India (2010)
16. Murata, T.: Petri nets: properties, analysis and applications. Proc. IEEE **77**(4), 541–580 (1989)
17. Patterson, D.A.: Computer Architecture: A Quantitative Approach. Elsevier, Amsterdam (2011)
18. Ratzer, A.V., et al.: 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). https://doi.org/10.1007/3-540-44919-1_28
19. Reisig, W.: Petri Nets: An Introduction, vol. 4. Springer, Heidelberg (2012)
20. Satish, N., Harris, M., Garland, M.: Designing efficient sorting algorithms for many-core GPUs. In: IEEE International Symposium on Parallel & Distributed Processing, IPDPS 2009, pp. 1–10. IEEE (2009)
21. Seward, H.E.: Information sorting in the application of electronic digital computers to business operations, Master Thesis, MIT (1954)
22. Shuaiwen, S., Chunyi, S., Barry, R., Kirk, C.: A simplified and accurate model of power-performance efficiency on emergent GPU architecure. In: 2013 IEEE 27th International Symposium on Parallel & Distributed Processing (IPDPS), pp. 673–686 (2013)
23. Storti, D., Yurtoglu, M.: CUDA for Engineers: An Introduction to High-performance Parallel Computing. Addison-Wesley Professional, Boston (2015)
24. Westergaard, M., (Eric) Verbeek, H.M.W.: CPN Tools official webpage. Eindhoven University of Technology
25. Williams, S., Waterman, A., Patterson, D.: Roofline: an insightful visual performance model for multicore architectures. Commun. ACM **52**(4), 65–76 (2009)
26. Cheng, S.-T., Hung, Y.: Estimation of job execution time in mapreduce framework over GPU clusters. In: The Fifth International Conference on Performance, Safety and Robustness in Complex Systems and Applications, PESARO 2015 (2015)

Model-Based Testing of the Gorums Framework for Fault-Tolerant Distributed Systems

Rui Wang[1](✉), Lars Michael Kristensen[1], Hein Meling[2], and Volker Stolz[1]

[1] Department of Computing, Mathematics, and Physics,
Western Norway University of Applied Sciences, Bergen, Norway
{rwa,lmkr,vsto}@hvl.no

[2] Department of Electrical Engineering and Computer Science,
University of Stavanger, Stavanger, Norway
hein.meling@uis.no

Abstract. Data replication is a central mechanism for the engineering of fault-tolerant distributed systems, and is used in the realization of most cloud computing services. This paper explores the use of Coloured Petri Nets (CPNs) for model-based testing of quorum-based distributed systems. We have developed an approach to model-based testing of fault-tolerant services implemented using the Go language and the Gorums framework. We show how a CPN model can be used to obtain both unit test cases for the quorum logic functions, and system level test cases consisting of quorum calls. The CPN model is also used to obtain the test oracles against which the result of running a test case can be compared. We demonstrate the application of our approach by considering an implementation of a distributed storage service on which we obtain 100% code coverage for the quorum functions, 96.7% statement coverage on the quorum calls, and 52.3% coverage on the Gorums framework. We demonstrate similar encouraging results also on a more complex Gorums-based implementation of the Paxos consensus protocol.

1 Introduction

Distributed systems serve millions of users in many important applications and domains. However, such complex systems are known to be difficult to implement correctly because they must cope with challenges such as concurrency and failures [12]. Thus, when designing and implementing distributed systems, it is important to ensure correctness and fault-tolerance. Distributed systems can rely on a quorum system to achieve fault-tolerance, yet it remains challenging to implement fault-tolerance correctly. Therefore, the use of testing techniques is essential to detect bugs and to improve the correctness of such systems.

One promising testing approach is *model-based testing* (MBT) [23]. MBT is a paradigm based on using models of a system under test (SUT) and its environment to generate test cases for the system. The goal of MBT is validation and

M. Koutny et al. (Eds.): ToPNoC XIII, LNCS 11090, pp. 158–180, 2018.
https://doi.org/10.1007/978-3-662-58381-4_8

error-detection by finding observable differences between the behavior of the implementation and the intended behavior of the SUT. A test case consists of test input and expected output and can be executed on the SUT. Typically, MBT involves: (a) build models of the SUT from informal requirements; (b) define test selection criteria for guiding the generation of test cases and the corresponding test oracle representing the ground-truth; (c) generate and run test cases; (d) compare the output from test case execution with the expected result from the test oracle. The component that performs (c) and (d) is known as a *test adaptor* and uses a *test oracle* to determine whether a test has passed or failed.

In this paper, we investigate the use of Coloured Petri Nets (CPNs) [11] for model-based testing applied to quorum-based distributed systems [24]. Quorum systems are fundamental to building fault-tolerant distributed systems, and recently the Gorums framework [17] has been developed to ease the implementation of quorum-based distributed systems. The Gorums framework constitutes a distributed middleware that hides the complexity in implementing the communication, synchronization, message processing, and error handling between the protocol entities. The widespread use of the Gorums framework will depend on the correctness of its implementation in Go. This motivates our goal of systematically testing the Gorums middleware implementation and provides an MBT approach that can be used to also systematically test applications that rely on the Gorums framework implementation.

The contribution of this paper is to propose an MBT approach using CPNs for quorum-based distributed applications implemented by the Gorums framework. To illustrate the application of our approach, we show in detail how it can be used on a Gorums-based implementation of a single-writer, multi-reader distributed storage. The distributed storage system is implemented with a read and a write *quorum call*, which clients can use to access the distributed storage. The distributed storage may return multiple replies to a quorum call. To simplify client access to the storage, Gorums uses a user-defined *quorum function* to coalesce the replies into a single reply that can then be returned to the client. For this particular storage system, we use a majority quorum. By developing a CPN model of such a distributed storage, we are able to generate test cases consisting of read and write quorum calls that test the Gorums framework implementation. For evaluation, we report on results obtained on the distributed storage system, and present results obtained on a more complex example in the form of the Paxos consensus protocol [16].

CPNs has been widely used for modeling and verifying models of distributed systems spanning domains such as workflow systems, communication protocols, and distributed algorithms [14]. Recently, work has also been done on automated code generation allowing an implementation of the modeled systems to be obtained [15]. Comprehensive testing of an implementation is, however, an equally important task in the engineering of distributed systems, independently of how the implementation has been obtained. This also applies in the case of automated code generation, as it is seldom the case that the correctness of the model-to-text transformations and their implementation can be formally proved. We have chosen CPNs as the foundation of our MBT approach as it has a strong track record for modeling distributed systems, and enables compact modeling of data and data manipulation which is required for message modeling, quorum functions modeling, and concrete test case generation. Furthermore, CPNs has the ability to create parametric models, perform model validation prior to test case generation, and it has mature tool support for both simulation and state space exploration, which is important in order to implement our approach and conduct practical experiments.

The rest of this paper is organized as follows. Section 2 introduces quorum-based distributed systems and the Gorums framework, and Sect. 3 describes the Gorums-based distributed storage which constitutes our system under test. Section 4 presents the constructed CPN model for test case generation, and Sect. 5 shows how state-spaces can be used to obtain test cases and test oracles. In Sect. 6 we present the Go implementation of our test adapter and how it is connected to the Gorums implementation of the distributed storage in order to execute the test cases. In Sect. 7 we report on experimental results. Section 8 presents related work, and in Sect. 9 we sum up conclusions and present directions for future work. The reader is assumed to be familiar with the basic concepts of high-level Petri Nets. This paper is an extended and revised version of an earlier workshop paper [25].

2 Quorum-Based Distributed Systems and Gorums

Distributed algorithms are commonly used to implement replicated services, and they rely on a quorum system [24] to achieve fault tolerance. That is, to access the replicated state, a process only needs to contact a quorum, e.g. a majority of the processes. In this way, a system can provide service despite the failure of individual processes. However, communicating with and handling replies from sets of processes often complicate the protocol implementations. The Gorums [17] framework has been developed to alleviate the development effort for building advanced distributed algorithms, such as Paxos [16] and distributed storage [2].

The Gorums framework reduces the complexity of implementing quorum-based distributed systems by providing two core abstractions: (a) a flexible and simple quorum call abstraction, which is used to communicate with a set of processes and to collect their responses, and (b) a quorum function abstraction which is used to process responses. These abstractions help to simplify the main control flow of protocol implementations. Figure 1 illustrates the interplay between the main abstractions provided by Gorums. Gorums consists of a runtime library and code generator that extends the gRPC [8] remote procedure

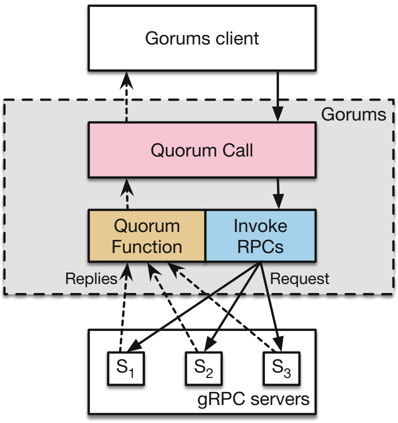


Fig. 1. Gorums architecture.

call library. Gorums allows clients to invoke a quorum call, i.e. a set of RPCs, on a group of servers, and to collect their replies. The replies are processed by a quorum function to determine if a quorum has been obtained. The quorum function is invoked every time a new reply is received at the client, to evaluate whether or not the received set of replies constitutes a quorum. With Gorums, developers can specify several RPC service methods using `protobuf` [9], and from this specification, Gorums's code generator will produce code to facilitate quorum calls and collection of replies. However, each RPC/quorum call method must provide a user-defined quorum function that Gorums will invoke to determine if a quorum has been obtained for that specific quorum call. In addition, the quorum function also provides a single reply value, based on a coalescing of the received reply values from the different server replicas. This coalesced reply value is then returned to the client as the result of its quorum call. That is, the invoking client does not see the individual replies.

The contribution of this paper is to provide an MBT approach for generating test cases to validate the correctness of the Gorums framework implementation itself. This comes in addition to test cases for quorum function and quorum call implementations for a specific use of the framework such as for implementing a distributed storage. The quorum functions for a specific protocol implementation must follow a well-defined interface generated by Gorums. These only require a set of reply values as input and a return of a single reply value together with a boolean quorum decision. Hence, quorum functions can easily be tested using unit tests. However, some quorum functions involve complex logic, and their input and output domains may be large, and so generating test cases from a model provides significant benefit to verify correctness. A quorum call is implemented by a set of RPCs, invoked at different servers, and so different interleavings must be considered due to invocations by different clients. Hence, using MBT we can produce sequences of interleavings aimed at finding bugs in the server-side implementations of the RPC methods and also in the Gorums runtime system.

3 System Under Test: Gorums and Distributed Storage

We have implemented a distributed storage system, with a single writer and multiple readers. The storage system is replicated for fault-tolerance, and is implemented using Gorums. To test this storage implementation, we have designed a corresponding CPN model that we use to generate test cases (see Sect. 4). In this section, we describe the different components of the distributed storage and how it has been implemented using Gorums.

As with any RPC library, Gorums requires that the server implements the methods specified in the service interface. For our distributed storage, we have implemented two server-side methods: Read() and Write(). These can be invoked as quorum calls from storage clients, to read/write the state of the storage. In our current implementation, we allow only a single write quorum call to be invoked, but any number of read quorum calls can be invoked by the client to read the state of the storage.

A client reading from the storage may observe different replies returned by the different server replicas. The reason for this is that the read may be interleaved with one or more writes generated by the client. To allow a reader to pick the correct reply value to return from a quorum call, each server maintains a timestamp that is incremented for each new Write(). That is, the reader will always return the value associated with the reply with the highest timestamp. Thus, to implement the reader using Gorums, we can simply implement a user-defined ReadQF quorum function for the Read() quorum call as shown in Algorithm 1. As this code illustrates, a set of replies from the different servers are coalesced into a single reply that can then be returned from the quorum call. The reply of the quorum function is determined by the reply from the server(s) having the highest timestamp.

The user-defined quorum functions are implemented as methods on an object of type QuorumSpec, named *qs* in Algorithm 1. This object holds information about the quorum size, such as ReadQSize, and other parameters used by the quorum functions. This *qs* object must satisfy an interface generated by Gorums's code generator. In Algorithm 1, ReadQSize is used to determine if sufficient replies have been received to return the server reply with the highest timestamp.

Algorithm 1. Read quorum function

```
1: func (qs QuorumSpec) ReadQF(replies []ReadReply)
2:     if len(replies) < qs.ReadQSize then                  ▷ read quorum size
3:         return nil, false                  ▷ no quorum yet, await more replies
4:     highest := ⊥                          ▷ reply with highest timestamp seen
5:     for r := range replies do
6:         if r.Timestamp ≥ highest.Timestamp then
7:             highest := r
8:     return highest, true                                      ▷ found quorum
```

4 CPN Testing Model for the Distributed Storage

In this section, we describe the CPN model of our test framework developed in order to generate test cases for the Gorums framework and the distributed storage implementation presented in Sect. 3. We model the entire system, parametrized by the number of clients and servers. Some key features of the model are the use of colored tokens for distinguishing multiple incoming and outgoing messages, and the quorum specification based on the numbers of replies received so far.

Figure 2 shows the top-most module of the CPN model. The substitution transition Clients represents the clients (users) of the distributed storage system while Servers represent the servers. The places ClientToServer and ServerToClient are used for modeling the message channels for communication between the clients and the servers. The CPN model has been constructed in a folded manner so that the number of servers is a parameter that can be configured without making changes to the net-structure. Below we provide more details on selected modules of the CPN model. The complete CPN model including all color sets, variable declarations, and function definitions is available from [5].

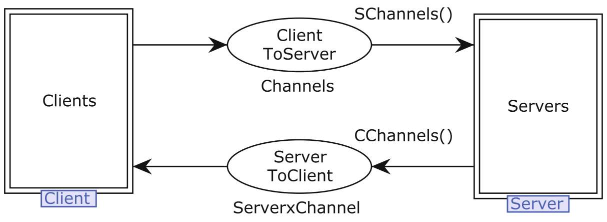


Fig. 2. Top-level module of the CPN model.

Figure 3 shows the client submodule of the Clients substitution transition in Fig. 2. The substitution transition QuorumCalls is used to model the behavior of applications running on the clients, which makes the read and write quorum calls. In particular, the submodules of QuorumCalls serve as test driver modules used to generate system tests for the distributed storage and the Gorums framework. The content of QuorumCalls depends on the specific test scenarios to be investigated for the system under test, and we give a concrete example of a test driver module in Sect. 6. The substitution transitions Read and Write represent the quorum calls provided by the distributed storage. The invocation of quorum calls is done by placing tokens on the Read and Write places. The port places ServerToClient and ClientToServer are linked to the identically named socket places in Fig. 2.

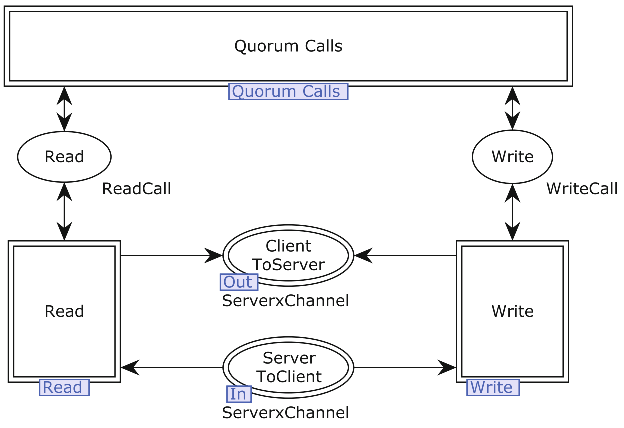


Fig. 3. The Clients module.

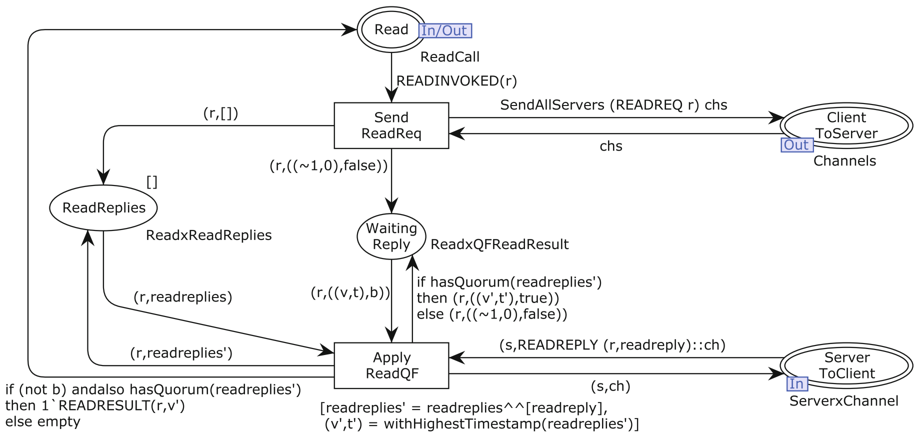


Fig. 4. The Read module.

Figure 4 shows the submodule of the Read substitution transition which provides an abstract implementation of the Read() quorum call. The main purpose of the Read module is to generate test cases for the ReadQF quorum function. A read quorum call is triggered by the presence of a token with the color `READINVOKED(r)`, where `r` identifies the call and is used to match replies from servers to the call. The execution of a read quorum call starts by sending a read request to each of the servers. This is modeled by the transition SendReadReq and the expression on the arc to place ClientToServer, which will add tokens representing read requests being sent to the servers. In addition, a list-token is put on place ReadReplies, which is used to collect the replies received from the servers. The call then enters a WaitingReply state and waits for replies coming back from the servers. When a read's reply comes back, represented by a token on place ServerToClient, then transition ApplyReadQF will be enabled. This transition takes the current list of `readreplies` and appends the received `readreply` to form `readreplies'`. The quorum function is then invoked, as represented by the arc expressions to WaitingReply and Read. If enough replies have been received, then a read result is returned to the Read place containing the value with the highest timestamp. As we will see later, we use occurrences of the ApplyReadQF transition for generating test cases for the ReadQF quorum function. In addition, we record the result computed by the CPN model as the test oracle and compare it to the result of our SUT's implementation of the ReadQF quorum function. The submodule for the Write() quorum call is similar. It has a transition ApplyWriteQF, which we use as a basis for generating test cases and obtain a test oracle for the WriteQF quorum function.

Figure 5 shows the server submodule of the Servers substitution transition in Fig. 2. The replicated state of each server is modeled by the place State. The two substitution transitions are used for modeling the handling of write requests and read requests on the server side.

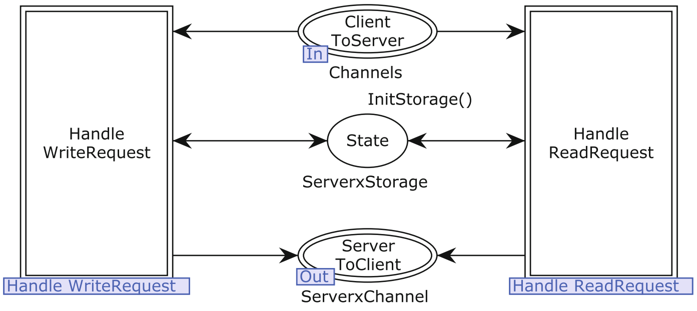


Fig. 5. The Server module.

Figure 6 shows the submodule of the substitution HandleWriteRequest modeling the processing of a write request from a client. The incoming write request will be presented as a value in the list-token on place ClientToServer and contains a value `v'` to be written in the distributed storage together with a timestamp `t'`. The server compares the timestamp of the incoming write request with the timestamp `t` for the currently stored value `v`. If the timestamp of the incoming write request is larger, then the new value is stored on the server, and a write acknowledgment is sent back in a write reply to the client. Otherwise, the stored value remains unchanged and a negative write acknowledgment is sent to the client in the write reply. Handling of an incoming request requires that the server is `running` (as opposed to `failed`) as modeled by the double arc connecting ServerStatus and HandleWriteRequest. The handling of read requests is modeled in a similar manner, except that no comparison is needed, and the server simply returns the currently stored value together with its timestamps.

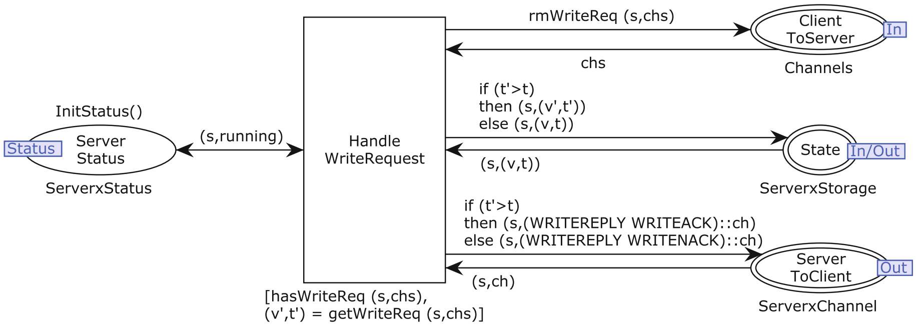


Fig. 6. The HandleWriteRequest module.

5 Test Case Generation

The generation of test cases for Gorums and the distributed storage system is based on the analysis of executions of the CPN model. Test cases can be generated for both the quorum functions and the quorum calls.

The test cases generated for the quorum functions are unit tests, whereas the test cases generated for quorum calls are system tests consisting of concurrent and interleaved invocations of read and write quorum calls. The latter tests both the implementation of the quorum calls and the Gorums framework implementation. In addition to the test cases, we also generate a *test oracle* for each test case to determine whether the test has passed.

5.1 Unit Tests for Quorum Functions

Test cases for the ReadQF quorum function can be obtained by considering occurrences of the ApplyReadQF transition (Fig. 4). When this transition occurs, the variable `readreplies'` is bound to the list of all replies that have been received from the servers so far, and which the quorum function is invoked on. In addition, we can use the implementation of the quorum function in the CPN model as the test oracle. This means that the expected result of invoking the quorum function can be obtained by considering the value of the token put back on place WaitingReply. The value of this token contains the result of invoking the quorum function in its second component. Generally, occurrences of ApplyReadQF can be detected using either state spaces or simulations:

State-space based detection. We explore the full state space of the CPN model searching for arcs corresponding to the ApplyReadQF transition. Whenever an occurrence is encountered we emit a test case together with the expected result. In this case, we obtain test cases for all the possible ways in which the quorum function can be invoked in the CPN model.

Simulation-based detection. We run a simulation of the CPN model and use the monitoring facilities of the CPN Tools [4] simulator to detect occurrences of the ApplyReadQF transition and emit the corresponding test cases. The advantage of this approach over the state-space based approach is scalability, while the disadvantage is potentially reduced test coverage.

Test cases are generated based on detecting transition occurrences. This is done in a uniform way for both detection approaches. Specifically, we rely on a *detection function*, which must evaluate to true whenever a specific transition occurrence is detected. When this happens, a *generator function* is invoked to generate the actual test case. The state space for the CPN testing model of the distributed storage service is relatively small and we can obtain all test cases based on state space-based detection. The Paxos consensus protocol considered in Sect. 7 is more complex, and hence we rely on simulation-based detection for its test case generation.

Listing 1 shows an example of how our test cases are represented using XML. The test case for the ReadQF quorum function corresponds to two replies (one with value 0 and timestamp 0, and one with value 42 and timestamp 1). With three servers, this constitutes a quorum, and the value returned from the quorum function is therefore expected to be 42 with the timestamp of 1.

```xml
<Test TestName="ReadQFTest">
  <TestCase CaseID="1">
    <TestValues>
      <Content>
        <Value>0</Value>
        <Timestamp>0</Timestamp>
      </Content>
      <Content>
        <Value>42</Value>
        <Timestamp>1</Timestamp>
      </Content>
    </TestValues>
    <ExpectResults>
      <Value>42</Value>
      <Timestamp>1</Timestamp>
    </ExpectResults>
    <ExpectQuorum>true</ExpectQuorum>
  </TestCase>
</Test>
```

Listing 1. Example of generated test cases for read quorum function.

5.2 System Tests of Quorum Calls

The generation of test cases and expected results is based on the submodule of the QuorumCalls substitution transition (see Fig. 3). This module acts as a test driver for the system by specifying scenarios for read and write quorum calls to the underlying quorum system. By varying this module, it is possible to generate different scenarios of read and write quorum calls.

Figure 7 shows an example of a test driver in which the client executes one read and one write quorum call as modeled by the transition InvokeRDWR. Upon completion of these two calls, there are server failures and a new read and a write call is invoked (modeled by the transition InvokeRDWRFailures). The server failures are modeled by changing the color of the server-tokens on place ServerStatus which is used with the place ServerStatus on the HandleWriteRequest (see Fig. 6). Each quorum call has a unique identifier (1, 2, 3, and 4) for identifying the call. Each write call also has a value (in this case 42 and 7) to be written to the distributed storage.

To make test case generation independent to the particular test driver module, we exploit that the read and write quorum calls, made during an execution of the CPN model, can be observed as tokens on the Read and Write socket places (see Fig. 3). When there is a `READINVOKED(i)` token on place READ for some integer `i`, it means that a read quorum call identified by `i` has been invoked. When the read quorum call has terminated, there will be a token with the color `READRESULT(i,v)` present on the place Read, where `v` is the value read by the call. The invocation and termination of write quorum calls can be detected in a similar manner by considering the tokens with the colors `WRITEINVOKED(i,v)`

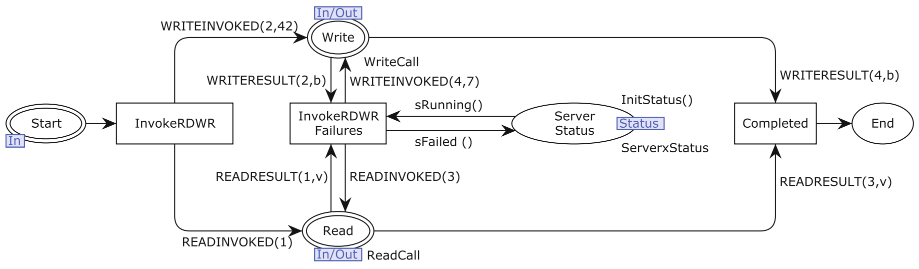


Fig. 7. The QuorumCalls module.

and WRITERESULT(i,b) on the place Write (Fig. 3), where the boolean value b denotes whether the value v was written or not.

Based on this, we can generate test cases in XML format specifying both the concurrent and sequential execution of read and write calls. Listing 2 (discussed further below) shows an example where first a read and a write are initiated and upon completion of these two calls, a new read call is initiated.

```
<Test TestName="SystemTest">
  <TestCase CaseID="WRprRDsqRD">
    <Routine RoutineID="A" OperationName="Write">
      <OperationValues>
        <Value>7</Value>
      </OperationValues>
      <Routine RoutineID="B" OperationName="Read">
        <OperationValues>
           <Value>7</Value>
           <Value></Value>
        </OperationValues>
      </Routine>
   </Routine>
   <Routine RoutineID="A" OperationName="Read">
     <OperationValues>
       <Value>7</Value>
     </OperationValues>
   </Routine>
 </TestCase>
</Test>
```

Listing 2. Example of a generated test cases for the concurrent and sequential execution of read and write calls.

We handle concurrent executions by nesting the read and write Routine tag as illustrated in Listing 2, while non-nested Routine tags are considered sequential. For write calls, we use the value tag to specify the value to be written, and for read calls we use the value tag to describe permissible values for the test case (see next section) returned by read calls. The absence of a value between value tags indicates that the result could be null—corresponding to the case where no value have yet been written into the storage.

It should be noted if the CPN model specifies that a read and write call may execute concurrently (independently), but happened to be executed in sequence in a concrete execution of the CPN model (e.g., first the read executes and completes and then the write executes an completes), then that will be specified as a sequential test case in the XML format. This is not a problem as the CPN model captures all the possible executions and hence there will be another execution of the CPN model in which the read and the write are running concurrently.

5.3 Test Oracle for System Tests

Checking that the result of an execution with read and write quorum calls is as expected is more complex than for quorum functions. This is because the result of concurrently executing read and write calls depends on the order in which messages are sent and received. Figure 8 shows an example test case in which there are two routines (threads of execution) that concurrently execute read and write quorum calls. When $Write_1$ and $Read_a$ are initialized and executed concurrently, the returned result of $Read_a$ could be the old value in the servers before $Write_1$ writes a new value to servers, or the returned result of $Read_a$ could be the value already written by $Write_1$. The same situation applies to $Write_2$ and $Read_c$. Since they are executed concurrently, the returned value of $Read_c$ could be the value written by $Write_1$ or $Write_2$.

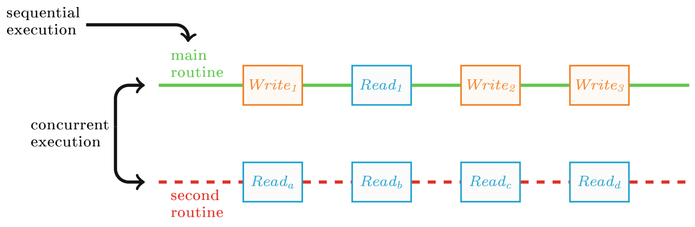


Fig. 8. An example of concurrent and sequential execution of quorum calls.

This means that if we execute (simulate) the CPN model with a test case containing concurrent read and write quorum calls, then the result returned upon completion of the calls may be different if we execute the same test case against the Go implementation. The reason is that we cannot control in what order the messages are sent and delivered by the underlying gRPC library, i.e., due to non-determinism in the execution. When we apply a state-space based approach for extracting the test cases, e.g., for the quorum function, then we can compute all the possible legal outcomes of a quorum call since the state space captures all interleaved executions. In contrast, we cannot obtain all legal values when extracting test cases from a single execution of the CPN model.

The first step towards constructing a general test oracle is to characterize the permissible values of a read quorum call. These are:

1. the initial value of the storage in case no writes were invoked before the read was invoked, or;

2. the value of the most recent write invoked but not terminated prior to the read call (if any) or;
3. the value of the most recent write that has terminated prior to invocation of the read or;
4. the value of a write that was invoked between the invocation and completion of the read.

The above can be formally captured in the stateful automaton shown in Fig. 9 (left), which can be used to monitor the global correctness of the distributed storage. The four events are shorthands for the abstract tokens per client-request observed in the model, e.g., `READINVOKED(i)` is abbreviated RI_i.

The set S is used to collect the set of permissible values for a read call. On a read call RI_i, any pending write $WI(c)$ observed since the last write-return $WI(c)$ is a potential read-result. We abuse notation from alternating automata with parametrized propositions [22] to capture that on a read invocation, we remain in the initial state and collect further input for a new instance of the monitor with the same current state (indicated by the dashed line) for subsequent read-invocations. We explain Fig. 9 (right) in the next section.

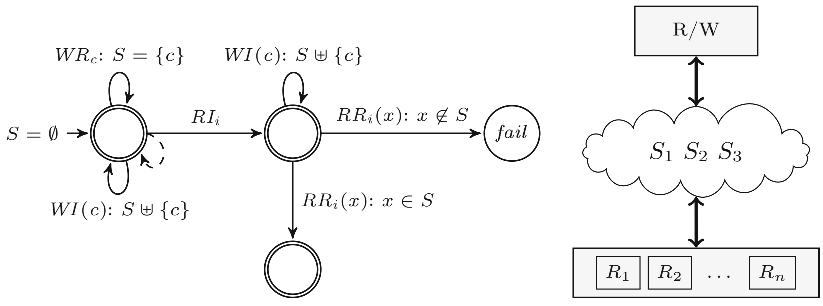


Fig. 9. Read-write automaton (left) and monitor deployment (right).

6 Test Case Execution

We have developed the QuoMBT test framework in order to perform model-based testing of quorum-based systems implemented using the Gorums framework. Also, we have implemented a client application and a distributed storage system which together with Gorums constitute the SUT. Figure 10 gives an overview of the testing framework comprised of CPN Tools and a test adapter. CPN Tools is used for modeling and generation of test cases and oracles (see Sects. 4 and 5). The generated

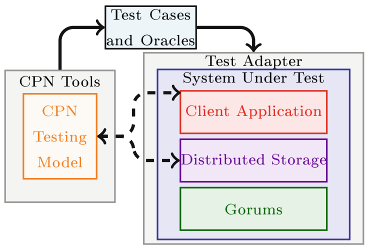


Fig. 10. QuoMBT testing framework.

test cases and oracles are written into XML files by CPN Tools, and then read by the test adapter. The reader of the test adapter feeds the test cases into the client application (test cases for quorum calls) and the distributed storage (test cases for quorum functions) implemented by the Gorums framework. Each test case is executed with the provided test values as inputs. The tester included in the test adapter compares the test oracle's output against the output of each test case in order to determine whether the test fails or succeeds. The test adapter is implemented in the Go programming language.

The reader in the test adapter can read XML files for unit tests of read and write quorum functions, and for system level tests involving quorum calls. The implementation of the reader uses Go's *encoding/xml* package, which makes it easy to define mappings between Go structs and XML elements. In order to map XML content into Go structs, each field of the Go struct has an associated XML tag, which is used by Go's XML decoder to identify the field to populate with content from the XML. We could have generated Go-based table-driven tests, which is already supported by the Go standard library. However, we chose to use an XML-based format for the generated test cases to enable reuse of the test generator across programming languages.

We have implemented the tester in the test adapter using the *testing* package provided by the Go standard library. This tester can start the implemented client application and execute generated test cases for the SUT. Go's testing infrastructure allows us to simply run the `go test` command to execute our generated tests, which will provide pass/fail information for each test case. In addition, this test infrastructure can also provide code coverage. When testing the distributed storage, we distinguish between quorum functions and quorum calls, because quorum functions are defined by developers when implementing their specific abstractions, whereas quorum calls are provided generally by the Gorums library. This separation also provides a modular approach to testing.

Our test adapter implements a Go-based tester for testing quorum functions, i.e., performing the unit tests. We simply iterate through the test cases obtained from the reader, invoking the ReadQF and WriteQF functions with the test values, and compare the results against the test oracles. The unit tests for read and write quorum functions can be performed without running any servers.

The system level tests require a set of running servers to test the complete system, including parts of the Gorums framework. When doing the system level tests involving quorum calls, the servers shown in Fig. 1 must be started first. Then, the test adapter starts a client so that it can execute the quorum calls. The test value, obtained from XML files, for each write quorum call is written to servers by calling the write quorum call, and for each read quorum call, the value returned by the servers will be captured by the tester to compare against the test oracle. For each write quorum call, the tests simply check if it returns an acknowledgment from servers.

The non-trivial part of the system test case execution is the concurrent and sequential executions of read and write quorum calls. For the detailed implementation of the storage involving quorum calls under test, the testing function for quorum calls run through each test case read from the reader. For the run of each test case, the write and read quorum calls can be executed both sequentially and concurrently depending on the test driver used. For the sequential executions, the decision to execute write or read calls is made according to their sequences in the XML files generated by CPN Tools. For the concurrent executions of write and read quorum calls, the test execution makes use of go-routines provided by the Go programming language. Therefore, within each run of test cases, a write or read quorum call is executed based on their sequence in the XML files. Meanwhile, there may be other read calls that can be executed concurrently with the running write or read quorum call and this is then done in a separate go-routine. After executing each test case, the returned values of quorum calls are collected.

In order to obtain a test oracle for quorum calls which can be used in both state space-based and simulation-based test case generation, we use the automaton in Fig. 9 (left) to perform run-time verification of the Go implementation when executed on the system test cases derived from the CPN model. Specifically, our test adapter implements a *run-time monitor* corresponding to the automaton in order to keep track of the invoked and terminated write calls and thereby determining whether a value returned from a read call is permissible. Our test framework currently runs the client (the single writer and multiple readers) within a single Go process. This allows us to directly call into the monitor *before* the client sends the fan-out messages to servers, and *after* the quorum function returns the resulting quorum value, to check the result of the read request for plausibility against the permitted values specified above. This corresponds to monitoring *all* calls and returns in a particular deployment, i.e., correlating read calls and returns of the client in the system against those of the writer in the shaded area of Fig. 9 (right).

7 Experimental Evaluation

We now consider experimental evaluation of our model-based testing approach based on CPNs. In Sect. 7.1, we present in detail the results obtained for the distributed storage system. In Sect. 7.2 we summarise experimental results for an additional case study in which we have applied our approach to the Paxos consensus protocol for data replication. The main purpose of the Paxos case study is to demonstrate the generality of our approach and to show that it can be applied also to more complex examples of Gorums-based distribution systems. The library which we have developed for CPN Tools as part of this work to support test case generation is available via [19].

7.1 Results on Distributed Storage

To perform an evaluation of our model-based test case generation, we consider the code coverage obtained using different test drivers for concurrent and sequential execution of quorum calls in the client application. Our experimental evaluation comprises both successful scenarios and scenarios involving server failures and programming errors. The toolchain of the Go language includes a code coverage tool which we have used to measure statement coverage.

Table 1 summarizes the experimental results obtained using different test drivers in which there are not server failures included. We consider the following test drivers: one read call (RD), one write call (WR), a read call followed by a write call (RD; WR), a write call followed by a read call (WR; RD), a read and a write call executed concurrently (WR||RD), a read and a write call executed concurrently and followed by a read call ((WR||RD); RD).

Table 1. Experimental results for distributed storage – successful scenarios.

Test driver		Test case generation					Test case execution (coverage in percentage)				
							System			Unit	
ID	Name	Nodes	Arcs	Time (seconds)	QC	QF	Gorums library	QCs		QFs	
								RD	WR	RD	WR
S1	RD	39	72	<1	1	3	24.6	84.4	0	100	0
S2	WR	39	72	<1	1	3	24.6	0	84.4	0	100
S3	RD; WR	254	543	<1	1	7	39.1	84.4	84.4	100	100
S4	WR; RD	254	543	<1	1	12	40.8	84.4	84.4	100	100
S5	WR||RD	1,549	4,379	1	6	17	40.8	84.4	84.4	100	100
S6	(WR||RD); RD	3,035	7,867	2	6	17	40.8	84.4	84.4	100	100

The table shows the number of nodes/arcs in the state space of the CPN model with the given test driver, the state space and test case generation time in seconds, the number of test cases generated for quorum calls (QC), the number of test cases generated for quorum functions (QF). For the test case execution, we show the code coverage (in percentage) that was obtained for the system level and unit tests. The results for successful execution scenarios show that the statement coverage for read (RD-QF) and write (WR-QF) quorum functions is 100% for both system and unit tests, as long as both read and write calls are involved. The statement coverage for read (RD-QC) and write (WR-QC) quorum calls is up to 84.4%. For the Gorums library as a whole, the statement coverage reaches 40.8%. It is worthwhile noting that the sizes of the state spaces considered are small. This is due to the fact that the CPN testing model describes the quorum-based system at a high level of abstraction which in turn is what makes the approach feasible.

The test cases considered above validates that the implementation of the distributed storage and the Gorums framework works correctly when there are no server failures. To further increase the code coverage and further evaluate our approach, we additionally evaluated the following aspects:

Programming errors. Gorums requires the developer to implement the quorum functions for the specific quorum-based system. To evaluate our ability to detect programming errors in these function, we injected programming errors in the quorum functions for the distributed storage (see Algorithm 1) such that the requirement for having a quorum was incorrectly implemented.

Server and communication failures. Quorum-based systems are designed to tolerate server failures. To test the Gorums framework under such conditions, we consider the S6 driver from Table 1 and created a scenario in which first S6 is executed, then there is one or more server failures, and then S6 is repeated. A related scenario is that the client attempts to make quorum-calls before the servers have started.

Rerunning the test cases from Table 1 in the presence of programming errors resulted in the test cases not passing. This demonstrates our ability to detect programming errors in the quorum functions.

The server failures scenarios are handled in the test adapter by a component that can terminate any number of the servers when executing such test cases. Our test case execution showed that the distributed storage and the Gorums framework in a configuration with three servers are able to handle up to one server failure (as expected). The size of the state spaces generated for these scenarios ranged between 1,500 states (all servers failed) and 3,000 (no server failure). The total number of test cases ranged from 9 to 16.

The results for scenarios involving failures and programming errors show that the statement coverage for read (RD-QF) and write (WR-QF) quorum functions is still 100% for both system and unit tests. The statement coverage for read (RD-QC) and write (WR-QC) quorum calls is increased to 96.7%. For the Gorums library as a whole, the statement coverage is also increased to 52.3%. The reason for the lower coverage of the Gorums library is that it contains code generated by Gorums's code generator, and among them, various auxiliary functions that are never used by our current implementation. The total number of lines of code for the system under test is approximately 2100 lines, which include generated code by Gorums's code generator (around 1800 lines), server code (around 120 lines), client code (around 80 lines) and the code for quorum functions (around 60 lines).

As part of analyzing the results of the code coverage and experimenting with the test case generation, we also discovered a code path that was not covered. So we added an additional test that would cover this particular path. This involved passing **nil** as an argument to either the read (RD-QC) or write (WR-QC) quorum calls. The code path in question had recently been introduced to support a new feature in Gorums, but when the code path was exercised without activating its intended feature, the test case revealed that this code path had a bug causing the test client to panic. The bug has since been reported to the Gorums developers, and a fix has been implemented.

7.2 Results on the Paxos Consensus Protocol

To show that our approach is more generally applicable, we report on one additional case study which we have conducted with our model-based testing approach for CPNs and the support provided by QuoMBT. The example is an implementation of the Paxos protocol using the Gorums framework. Paxos is a fault-tolerant consensus protocol that makes it possible to construct a replicated service using a group of server replicas. Paxos is considerably more complex than the distributed storage system, and each Paxos node (server replica) implements a proposer, an accepter, and a learner subsystem in addition to software components for failure and leader detection. Furthermore, three quorum calls (prepare, accept, and commit) are used in the implementation of the protocol. Due to the complexity of the Paxos protocol we have used simulation-based test case generation using up to 10 simulation runs to extract test cases.

Table 2 summarizes the experimental results obtained. The table shows the statement coverage obtained for the different subsystems of our Paxos implementation. Note that the Unit tests are for the quorum functions and hence not applicable for the other subsystems. The two numbers written below System tests and Unit tests gives the total number of test cases generated for 3 and 5 replica configurations, respectively. The test case generation for each configuration considered took less than 10 seconds, and the execution of each test case took less than one minute.

Table 2. Experimental results for test case generation and execution.

Subsystem	Component	System tests	Unit tests
		15/38	74/424
Gorums library		51.8%	-
Paxos core	Proposer	97.4%	-
	Acceptor	100.0%	-
	Failure detector	75.0%	-
	Leader detector	91.4%	-
	Replica	91.4%	-
Quorum calls	Prepare	83.9%	-
	Accept	83.9%	-
	Commit	83.9%	-
Quorum functions	Prepare	100.0%	90.0%
	Accept	100.0%	85.7%

The results show that the statement coverage of unit tests for Prepare and Accept quorum functions are up to 90% and 85.7%, respectively. For the system tests, the statement coverage for Prepare, Accept and Commit quorum calls reaches 83.9%, respectively; the results of statement coverage for Prepare and Accept quorum functions are up to 100%; for the Paxos implementation (Paxos core in the table), the Proposer module's statement coverage reaches 97.4%; the statement coverage of the Acceptor module is up to 100%; the statement coverages of the Failure Detector and Leader Detector modules reach 75.0% and 91.4%, respectively; the statement coverage of the Paxos replica module reaches 91.4%; for the Gorums library as a whole, the highest statement coverage reaches 51.8%.

Similar to the distributed storage system, we obtain a high statement coverage for the Paxos implementation. In addition, using the generated test cases we identified several programming errors in the Paxos implementation which could then be fixed. To construct the CPN testing model for the Paxos protocol, we reused the modeling patterns for quorum functions and quorum calls developed for the distributed storage system. Furthermore, the test case generation algorithms were directly used without change to generate test cases for the Paxos protocol. The parts that had to be developed specifically for the Paxos protocol was the observation and detection functions, and parts of the formatting function used to generate the XML test case representation. Finally, parts of the test adapter had to be implemented to match the quorum calls and quorum functions that are specific for the Paxos implementation. This shows that significant parts of our approach can be used for other Gorum-based protocol implementations.

8 Related Work

Model-based testing is a large research area, and MBT approaches and tools have been developed based on a variety of modeling formalisms, including flowcharts, decision tables, finite-state machines, Petri Nets, state-charts, object-oriented models, and BPMN [13]. Saifan and Dingel's survey [20] provides a detailed description of how model-based testing is effective in testing different aspects of distributed systems, and it classifies model-based testing based on different criteria and compares several model-based testing tools for distributed systems based on this classification. The comparison, however, does not identify work that can be applied to systems that rely on a quorum system to achieve fault-tolerance.

The Gorums framework has only recently been developed, and hence there does not yet exist work that have considered model-based testing of this framework. Chubby [3] was one of the first implementations of Paxos that were deployed in a production environment, and thus were extensively tested. They highlight that at the time (2007), it was unrealistic to prove correct a real system of that size. Thus to achieve robustness, they adopted meticulous software engineering practices, and tested random sequences of network outages, message delays, timeouts, and process crashes. Using our CPN model and our generated tests, we aim to test many of the same attributes in a more systematic manner. Xiangdong et al. [10] applied a CPN-based simulation method on a quorum-based distributed storage system called Cassandra [1]. Cassandra is highly configurable, and the focus in their work was to find appropriate parameter settings to achieve the best performance. To this end, they developed a CPN-based simulator specifically for Cassandra, which allow tuning various system parameters such as cluster size, timeouts and read/write ratios, for their CPN models. In our work, we focus on using the CPN test models for generating test cases to perform both unit and system tests to the implementations of distributed systems.

The Integration and System Test Automation (ITSA) tool follows a CPN-based approach to MBT [28]. The tool can generate test code for a variety of languages including Java, C/C++, *C#* and HTML. Compared to the ITSA tool, our MBT approach is not tied to a particular programming language, since the test cases can be generated as an XML format, which can be read by any programming language. The ITSA tool also uses the state space of the testing model to generate and select test cases. To obtain concrete test cases with input data, the tool relies on a separate model-to-implementation mapping. In contrast, we obtain the input data for the quorum functions and calls directly from the data modeling contained in the CPN testing model. As a case study, the ISTA tool has been applied to an online shopping system. However, their approach does not appear to be suitable for testing complex distributed system protocols, since they do not handle concurrency and failures, which is at the core of our work in this paper.

Faria et al. [6] use timed event-driven CPNs to generate test code. They do not use CPNs as a direct interface to the user, but generate them from UML sequence diagrams. Their tool suite has a different focus in that they instrument a running system to observe the messages specified in the sequence diagrams. The toolset can only perform JUnit tests on Java-based applications, and it has not been used for unit and system testing of distributed systems. Liu et al. [18] has also proposed a CPN-based test generation approach. The approach requires defining a conformance testing-oriented CPN (CT-CPN) model and a PN-ioco relation specifying how an implementation conforms to its specifications. Their test case generation algorithm for the CT-CPN model is simulation-based. In our approach on the other hand, we can directly generate test cases using both state space-based and simulation-based test case generation for an existing implementation of the system under test. A model-based test generation technique based on CPNs is used by Daohua, Eckehard and Jan [27] to verify a module of a satellite-based train control system. They use CPN Tools to generate the reachability graph of the test model, and use state space analysis with CPN Tools to extract the expected output of each test case from the path of the reachability graph. However, their technique does not support simulation-based test case generation, which is of utmost importance for scalability.

Moreover, Watanabe and Kudoh [26] propose two different CPN-based test suite generation methods for concurrent systems. However, their methods do not directly address a particular way to derive a CPN testing model for a distributed system, nor do they analyze achieved code coverage. Wei et al. [29] describe two algorithms for generating test cases and test sequences from a CPN model. In their method, a CPN model of the system under test is created. This model is then used as input to their APCO algorithm to generate an initial set of test cases which can be converted to test sequences using their SPS algorithm. Then, the set of original test cases and the test sequences can be exported as XML formatted files. They demonstrated their MBT approach for testing a radio module in a centralized railway control system. In contrast to our approach, Wei et al. do not consider test scenarios with failures, do not handle concurrency, and their approach has not been used to validate distributed systems. Finally, we discuss Farooq, Lam and Li's test sequence generation technique [7]. They derive a CPN model from a UML Activity Diagram, and use the derived model to generate test sequences. They demonstrate their approach on a fictional enterprise commerce system, describing the process of purchasing products online. In our MBT approach, we have designed a testing framework, consisting of the constructed CPN test models, test case generation algorithms and a test adapter, in order to enable the execution of the generated tests on real distributed systems.

9 Conclusions and Future Work

The main contribution of our work is an MBT approach that can be used for testing quorum-based distributed systems implemented using Gorums. Our approach includes modeling patterns, test case generation algorithms, and a test case execution infrastructure. As case studies, we have applied our approach to a distributed storage system and a Paxos implementation to illustrate and evaluate its applicability. The results are promising in that we have obtained high code coverage by considering both common case execution scenarios and failure scenarios. Furthermore, the results have been obtained with relatively simple test drivers and a small number of test cases. We have shown that in addition to obtaining results on code coverage, our generated unit and system tests are able to detect programming errors.

An important attribute of our approach is that the CPN testing model has been constructed such that it can serve as a basis for model-based testing of *other* quorum-based systems. This has been demonstrated by the application of our approach to the more complex Paxos consensus protocol. In particular, it is only the modeling of the quorum calls on the client and server side that are system dependent. To experiment with different quorum functions for a given quorum system, it is only the implementation of the quorum functions in Standard ML that needs to be changed. The state space and simulation-based test case generation approaches are independent of the particular quorum system under test. Our current solution uses the CPN model to generate test cases and record the correct response from the quorum function. The global monitor presented in Sect. 5 independently specifies safe behavior in the form of correct read calls.

The work presented in this paper opens up several directions of future work. We have obtained good coverage results on the quorum functions and quorum calls with the current testing model by considering both successful execution scenarios and scenarios involving server failures and programming errors. However, in order to increase coverage and consider more of Gorums's code paths, we need to test the quorum calls under additional failures scenarios and adverse conditions, such as network errors. This will require extensions to the model e.g. for generating timeouts, which in turn must be recorded in the test cases. This in turn will require extensions to the test adapter such that the environment can replay these events during test case execution.

Model-based testing can be used to test a system either by connecting a model (acting as a test driver) directly to an instance of the running system, or, as we do in this paper, generate test cases offline and execute these test cases against the system. The main challenge related to this, is how to handle non-determinism during test case execution. In our current approach, we have addressed this by using monitors known from the field of run-time verification. Instead of the automaton, a different formal specification logic for (distributed) systems could have been used, e.g. Scheffel and Schmitz's distributed temporal logic [21]. Their three-valued logic would allow us to adequately capture that the monitor has neither detected successful nor failed completion.

To further evaluate the generality of our modeling and test case generation approach, we need to apply it to additional quorum-based systems. For example, we can extend our current distributed storage to support multi-writer storages with multiple clients. This will challenge the limits of state space-based generation of test cases as was also demonstrated with the Paxos protocol. A future direction is to also extend our approach to be applicable also to non quorum-based distributed systems. In particular, it becomes important to investigate in more detail the test coverage that can be obtained with simulation versus the test case coverage that can be obtained with state spaces. We anticipate that this will motivate work into techniques for on-the-fly test case generation and test case selection during state space exploration.

References

1. Apache Software Foundation. Apache Cassandra. http://cassandra.apache.org
2. Attiya, H., Bar-Noy, A., Dolev, D.: Sharing memory robustly in message-passing systems. J. ACM **42**(1), 124–142 (1995)
3. Chandra, T.D., Griesemer, R., Redstone, J.: Paxos made live: an engineering perspective. In: Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing, PODC 2007, pp. 398–407. ACM (2007)
4. CPN Tools. CPN Tools. http://www.cpntools.org
5. CPN Testing Model for Gorum-based Distributed Storage, July 2018. http://home.hib.no/ansatte/lmkr/DistributedStorage.xml
6. Faria, J.P., Paiva, A.C.R.: A toolset for conformance testing against UML sequence diagrams based on event-driven colored Petri nets. Int. J. Softw. Tools Technol. Transfer **18**(3), 285–304 (2016)
7. Farooq, U., Lam, C.P., Li, H.: Towards automated test sequence generation. In: Australian Conference on Software Engineering (ASWEC 2008), pp. 441–450 (2008)
8. Google Inc. gRPC Remote Procedure Calls. http://www.grpc.io
9. Google Inc., Protocol Buffers. http://developers.google.com/protocol-buffers
10. Huang, X., Wang, J., Qiao, J., Zheng, L., Zhang, J., Wong, R.K.: Performance and replica consistency simulation for quorum-based NoSQL system cassandra. In: van der Aalst, W., Best, E. (eds.) PETRI NETS 2017. LNCS, vol. 10258, pp. 78–98. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-57861-3_6
11. Jensen, K., Kristensen, L.M.: Coloured Petri nets: a graphical language for modelling and validation of concurrent systems. Commun. ACM **58**(6), 61–70 (2015)

12. Jepsen. Distributed Systems Safety Analysis. http://jepsen.io
13. Jorgensen, P.: The Craft of Model-Based Testing. CRC Press, Boca Raton (2017)
14. Kristensen, L.M., Simonsen, K.I.F.: Applications of coloured Petri nets for functional validation of protocol designs. In: Jensen, K., van der Aalst, W.M.P., Balbo, G., Koutny, M., Wolf, K. (eds.) Transactions on Petri Nets and Other Models of Concurrency VII. LNCS, vol. 7480, pp. 56–115. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38143-0_3
15. Kristensen, L.M., Veiset, V.: Transforming CPN models into code for TinyOS: a case study of the RPL protocol. In: Kordon, F., Moldt, D. (eds.) PETRI NETS 2016. LNCS, vol. 9698, pp. 135–154. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-39086-4_10
16. Lamport, L.: The Part-time parliament. ACM Trans. Comput. Syst. **16**(2), 133–169 (1998)
17. Lea, T.E., Jehl, L., Meling, H.: Towards new abstractions for implementing quorum-based systems. In: Proceedings of 37th IEEE International Conference on Distributed Computing Systems (ICDCS), pp. 2380–2385 (2017)
18. Liu, J., Ye, X., Li, J.: Colored Petri nets model based conformance test generation. In: IEEE Symposium on Computers and Communications (ISCC), pp. 967–970. IEEE (2011)
19. MBT/CPN. Repository, July 2018. https://github.com/selabhvl/mbtcpn.git
20. Saifan, A., Dingel, J.: Model-based testing of distributed systems. Technical report 548, School of Computing, Queen's University, Canada (2008)
21. Scheffel, T., Schmitz, M.: Three-valued asynchronous distributed runtime verification. In: Twelfth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE), pp. 52–61. IEEE (2014)
22. Stolz, V.: Temporal assertions with parametrized propositions. J. Logic Comput. **20**(3), 743–757 (2010)
23. Utting, M., Pretschner, A., Legeard, B.: A taxonomy of model-based testing approaches. Softw. Test. Verif. Reliab. **22**, 297–312 (2012)
24. Vukolic, M.: Quorum Systems: With Applications to Storage and Consensus. Morgan and Claypool, San Rafael (2012)
25. Wang, R., Kristensen, L.M., Meling, H., Stolz, V.: Application of model-based testing on a quorum-based distributed storage. In: Proceedings of PNSE 2017. CEUR Workshop Proceedings, vol. 1846, pp. 177–196 (2017)
26. Watanabe, H., Kudoh, T.: Test suite generation methods for concurrent systems based on coloured Petri nets. In: Software Engineering Conference, pp. 242–251. IEEE (1995)
27. Wu, D., Schnieder, E., Krause, J.: Model-based test generation techniques verifying the on-board module of a satellite-based train control system model. In: 2013 IEEE International Conference on Intelligent Rail Transportation Proceedings, pp. 274–279, August 2013
28. Xu, D.: A tool for automated test code generation from high-level Petri nets. In: Kristensen, L.M., Petrucci, L. (eds.) PETRI NETS 2011. LNCS, vol. 6709, pp. 308–317. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21834-7_17
29. Zheng, W., Liang, C., Wang, R., Kong, W.: Automated test approach based on all paths covered optimal algorithm and sequence priority selected algorithm. IEEE Trans. Intell. Transp. Syst. **15**(6), 2551–2560 (2014)

MCC'2017 – The Seventh Model Checking Contest

Fabrice Kordon[1(✉)], Hubert Garavel[2,3], Lom Messan Hillah[1,4], Emmanuel Paviot-Adet[1,5], Loïg Jezequel[6], Francis Hulin-Hubard[7], Elvio Amparore[8], Marco Beccuti[8], Bernard Berthomieu[9], Hugues Evrard[10], Peter G. Jensen[11], Didier Le Botlan[9], Torsten Liebke[12], Jeroen Meijer[13], Jiří Srba[11,14], Yann Thierry-Mieg[1], Jaco van de Pol[13], and Karsten Wolf[12]

[1] Sorbonne Université, CNRS, LIP6, 75005 Paris, France
`Fabrice.Kordon@lip6.fr`
[2] Univ. Grenoble Alpes, Inria, CNRS, LIG, 38000 Grenoble, France
[3] Universität des Saarlandes, Saarbrücken, Germany
[4] Université Paris Nanterre, 92001 Nanterre, France
[5] Université Paris Descartes, 75005 Paris, France
[6] Université de Nantes, LS2N, UMR CNRS 6597, 44321 Nantes, France
[7] CNRS, LSV, Ecole Normale Supérieure Paris-Saclay, 94235 Cachan, France
[8] Università degli Studi di Torino, Turin, Italy
[9] LAAS-CNRS, Université de Toulouse, CNRS, INSA, Toulouse, France
[10] Imperial College London, London, UK
[11] Department of Computer Science, Aalborg University, Aalborg, Denmark
[12] Institut für Informatik, Universität Rostock, 18051 Rostock, Germany
[13] Department of Computer Science, University of Twente, Enschede, The Netherlands
[14] FI MU, Brno, Czech Republic

Abstract. Created in 2011, the Model Checking Contest (MCC) is an annual competition dedicated to provide a fair evaluation of software tools that verify concurrent systems using state-space exploration techniques and model checking. This article presents the principles and results of the 2017 edition of the MCC, which took place along with the Petri Net and ACSD joint conferences in Zaragoza, Spain.

1 Goals and Scope of the Model Checking Contest

The Model Checking Contest (MCC) is part of the growing trend of scientific contests, among which one can also mention: the SAT[1] and the SMT[2] competitions, the Hardware Model Checking Competition[3], the Rigorous Examination of Reactive Systems challenge[4], the Timing Analysis Contest[5], and the Compe-

[1] http://www.satcompetition.org.
[2] http://www.smtcomp.org.
[3] http://fmv.jku.at/hwmcc/.
[4] http://rers-challenge.org.
[5] http://www.tauworkshop.com/.

M. Koutny et al. (Eds.): ToPNoC XIII, LNCS 11090, pp. 181–209, 2018.
https://doi.org/10.1007/978-3-662-58381-4_9

tition on Software Verification[6]. The overall goal of these contests is to identify the theoretical approaches that are the most fruitful in practice when applied to a variety of examples, and figure out which types of systems are best handled by each approach. These contests also favor the emergence of systematic, rigorous, and reproducible ways to assess the capabilities of verification tools on complex (realistic and synthesized) benchmarks.

The primary goal of the MCC is to evaluate model-checking tools that analyze formal description of concurrent systems, i.e., systems in which several processes run simultaneously, communicating and synchronizing together. Examples of such systems include hardware, software, communication protocols, and biological models. The Model Checking Contest has been actively growing since its first edition in 2011, attracting key people sharing a formal methods background, but with diverse knowledge.

The community gathered around the MCC is actively involved in key activities that contribute to its growth year after year: contributing models to the benchmark, submitting tools to the competition, improving the automated generation of temporal logic formulas, maintaining the repository of models, contributing the infrastructure for running and evaluating the competing tools, improving the performance measurement and assessment tools, and publishing the results.

So far, all editions of the MCC have been using Petri nets to describe the analyzed systems. However, the contest is also open to tools not primarily based on Petri nets. Indeed, we have observed, over several editions, participating tools interfacing a native generic engine with the input format of the MCC.

The present paper reports about the seventh edition, which was organized in Zaragoza as a satellite event of the 38th International Conference on Application and Theory of Petri Nets and Concurrency and the 17th International Conference on Application of Concurrency to System Design. The goals of the MCC were first depicted in [31]; for this first edition of the MCC, experiments were conducted on a small number of models, and tools were merely asked to build the reachability graph, perform deadlock detection, and evaluate simple reachability formulas. Over the years, new models and new classes of problems have been added, and enhancements of the evaluation procedure have been introduced, which have been described in [32], which reports about the fifth edition of the MCC held in 2015. Compared to this latter presentation, the present paper:

– highlights the new benchmarks used for the recent editions of the contest, highlighting the variety of the contributed models;
– reports about improvements in the formulas generation workflow and updates in restructuring and simplifying the categories of verification tasks in the competition;
– discusses the impact of the growing number of runs (as a result of the growing number of models and submitted tools) on the evolution of the execution infrastructure and in-house assessment tool Bench$K$it;

[6] https://sv-comp.sosy-lab.org/.

– provides an in-depth analysis and discussion of the results and involved techniques used in the competing tools, with a comparison between symbolic approaches (i.e., those based on decision diagrams) and explicit ones (i.e., those based on the enumeration of explicit states); and
– presents all the tools that won a gold or silver medal during the 2017 edition of the MCC; for each winning tool, the underlying algorithmic techniques are explained, and the lessons learned from the contest are discussed.

The remainder of this paper is organized as follows. Section 2 presents the models submitted to the 2017 edition of the MCC, highlighting an interesting collection of models (originating from novel distributed algorithms) that have been contributed to the MCC, and describes the way temporal-logic formulas have been produced by the MCC team. Then, the monitoring environment and the experimental conditions are sketched in Sect. 3. Section 4 focuses on the results of the 2017 edition and the presentation of the tools that won a gold or silver medal in at least one examination. Finally, Sect. 5 reflects on the overall experience gained in organizing the MCC over the first seven editions, and discusses future work.

2 Models and Formulas

All the tools participating in a given edition of the MCC are evaluated on the same benchmark suite, which is incrementally updated every year. The yearly edition of the MCC starts with a *call for models* inviting the scientific community at large (i.e., beyond the developers of the participating tools) to propose novel benchmarks that will be used for the MCC. These benchmarks consist of *models* and *formulas*.

2.1 Models

Each *model* corresponds to a particular academic or industrial problem, e.g., a distributed algorithm, a hardware protocol in a circuit, a biological process, etc. A model may be parameterized by one or more parameters representing quantities, such as the number of agents in a concurrent system, the number of messages exchanged and the like. To each parameterized model are associated as many *instances* (typically, between 2 and 25) as there are different combinations of parameter values; each non-parameterized model has a single associated instance.

Each instance is a Petri net encoded in the PNML [26] file format. Each model, its instances, and their structural and behavioural properties are described in a synthetic PDF document called *model form* – see [32, Sect. 2] for details about model forms and their preparation. There are two kinds of instances: colored Petri nets and place-transition nets (noted P/T nets). Among the latter, one identifies the particular class of NUPNs (*Nested-Unit Petri Nets*) [17] [32, Sect. 2], a structured form of Petri nets that preserve locality and hierarchy information by recursively expressing a net in terms of parallel and sequential

Table 1. Accumulation of models and instances over the years

Year	2011	2012	2013	2014	2015	2016	2017
New models	7	12	9	15	13	11	10
All models	7	19	28	43	56	67	77
New instances, among which:	95	101	70	138	121	139	153
– *new colored nets*	43	37	24	33	27	9	16
– *new P/T nets*	52	64	46	105	94	130	137
– *new NUPNs (among P/T nets)*	0	0	1	5	15	62	64
All instances	95	196	266	404	525	664	817

compositions. It is worth mentioning that, beyond the Model Checking Contest, the NUPN model has also been adopted for the parallel problems of the RERS (*Rigorous Examination of Reactive Systems*) competition[7].

In 2017, following the MCC call for models, ten new models (totalling 153 instances) have been proposed, namely: *BART* (a sample speed controller, by Fabrice Kordon), *ClientsAndServers* (an architecture with clients, servers, managers, and resources, by Claude Girault), *CloudReconfiguration* (a dynamic reconfiguration protocol for cloud applications, by Rim Abid, Gwen Salaün, and Noël de Palma), *DLCround* (various distributed implementations of the musical chairs game, by Hugues Evrard), *FlexibleBarrier* (a novel barrier algorithm for multitasking on GPUs, by H. Evrard), *HexagonalGrid* (a packet-switching network whose ports are situated on the sides of an hexagon, by Tatiana Shmeleva), *JoinFreeModules* (a model of a schedulability problem, by Thomas Hujsa), *NeighborGrid* (a canvas of cellular automata, by Dmitry Zaitsev), *Referendum* (a simple referendum system, by F. Kordon), and *RobotManipulation* (concurrent processes that handle robots, by F. Kordon).

The new models submitted each year are called *surprise models* because they are not known in advance by the tool developers participating in the MCC, contrary to the models submitted during the former years, which are thus called *known models*. The surprise models are merged with the known ones to form a growing collection[8] (continuously expanded since 2011), which gathers systems from diverse academic and industrial fields: software, hardware, networking, biology, etc. Table 1 gives an account of this collection, which currently has 77 models and 817 instances; colored Petri nets, P/T nets, and NUPNs represent respectively 23%, 77%, and 18% of this collection.

2.2 Featured Model Contribution

The collection of MCC benchmarks is a perennial result, which will remain available to the scientific community after the MCC contests have stopped. The usefulness of this collection is already witnessed by nearly fifty scientific publications[9].

[7] See http://www.rers-challenge.org.
[8] The collection of benchmarks is available from http://mcc.lip6.fr/models.php.
[9] The current list of publications is available from http://mcc.lip6.fr/bibliography.php.

The contributors who submit new models play an essential role in every MCC edition. However, these contributors receive much less visibility than the tool developers participating in the MCC, as the latter can be rewarded by podiums and medals. To address this bias, it was decided to put a special focus on the contributor who submitted the most models since the first MCC edition. So far, the most active contributors are Fabrice Kordon (15 models), Lom Messan Hillah (6 models), Hugues Evrard (5 models), Monika Heiner (5 models), and Niels Lohmann (5 models). Given that many of these contributors are already authors of the present article, either as MCC organizers or tool developers, we chose to put the focus on Hugues Evrard by inviting him to present the five models he contributed to the MCC.

These five models are: *MultiwaySync* (2014, prepared with Frédéric Lang), *Raft* (2015), *DLCshifumi* (2016, also with F. Lang), *DLCround* (2017), and *FlexibleBarrier* (2017). These models share two common points: (i) they all describe involved aspects of the implementation of synchronisation among concurrent processes in modern distributed systems, and (ii) these models have not been expressed directly using Petri nets, but rather generated automatically from formal specifications written in a higher-level concurrent language. Precisely, these models have been written in LNT [19], a concurrent language designed as a modern replacement for LOTOS (ISO/IEC international standard 8807). The LNT specifications are first translated to LOTOS using the LNT2LOTOS translator, which is part of the CADP toolbox [18], and then to Petri nets (actually, NUPNs) using the CÆSAR compiler, also available as part of CADP.

MultiwaySync [15] is a distributed synchronisation protocol that implements multiway rendezvous, the generalization of Hoare's rendezvous [25] to more than two concurrent processes. Multiway rendezvous (see [20] for an overview) is used by most process calculi to perform synchronisation and communication between an arbitrary number of concurrent processes. It is more complex than ordinary synchronisation barriers, as each process can be willing to participate in several synchronisations at the same time, but can only engage in a single one. *MultiwaySync* implements multiway synchronisation on top of the lower-level, asynchronous message-passing primitives provided by usual networks; the design of *MultiwaySync* revealed subtle bugs in former protocols implementing multiway synchronisation.

Raft was obtained by formally specifying in LNT the Raft algorithm [37] that enables concurrent processes to reach consensus, even in the presence of failures; this is crucial for implementing fault-tolerant services replicated on many servers, some of which may crash or be unreliable. Raft was designed to replace Paxos, the standard consensus algorithm [33], which is notoriously complex and difficult to implement correctly; the rapid adoption of Raft by several companies (Facebook, Hashicorp, CoreOS) illustrates the need for a "simpler Paxos".

DLCshifumi and *DLCround* model distributed implementations of two well-known games: shifumi (rock-paper-scissors) and musical chairs. These two models, contrary to the three other ones, have not been written by hand in LNT, but produced automatically using the DLC tool [14,16]. DLC (Distributed LNT

Compiler) generates, from an LNT specification, a distributed implementation running on a set of machines communicating through TCP sockets and synchronizing using the aforementioned *MultiwaySync* protocol. For verification purpose, DLC can also generate a formal model of such a distributed implementation, where the multiway synchronisation protocol used in the runtime is made explicit. This formal model is itself expressed in LNT, and this is the way *DLCshifumi* and *DLCround* have been produced, before being translated to NUPNs.

Finally, *FlexibleBarrier* describes a special barrier used in the context of cooperative kernels, so as to enable multitasking on GPUs [39]. In this programming model, processes can offer to be killed or forked by the scheduler at precise points of their execution; hence, the number of alive processes varies dynamically and the flexible barrier must interact with the scheduler to know how many processes have to be synchronised.

2.3 Formulas

Tools competing in the MCC are evaluated over five categories of verification tasks: state-space generation, upper-bounds computation (this category was introduced in 2016), reachability analysis, CTL analysis, and LTL analysis (see their description in Sect. 4.1). To maximize tool participation, we further divided the four latter categories into subcategories containing only formulas with a restricted syntax. Each tool developer may choose in which categories/subcategories the tool participates.

In 2015, we consolidated the formula language and provided simplified XML metamodel for each (sub)category, while preserving backward compatibility with previous MCC editions. Since then, the only change to the general metamodel for formulas has been a redefinition of one atomic proposition (called *place-bound* and used only in the new upper-bounds computation category) because tool developers had reported it would be more convenient.

In 2016, we reduced the number of categories. Previously, subcases were simpler versions (with restricted grammar) of larger cases—for example we had *LTLFireability* and its simpler counterpart *LTLFireabilitySimple*. Since every tool participating in the simple subcategories was also participating in its more general counterpart (with similar results in both categories), they were not interesting any more.

For each model instance and each subcategory, 16 formulas are automatically generated and stored into a single XML file (of which a textual version is also provided for the convenience of tool developers). Each tool participating in the corresponding subcategory is requested to evaluate, on the corresponding instance, all or part of the formulas contained in the XML file.

To obtain formulas of good quality we apply the following process for reachability and CTL formulas (see Fig. 1). Using the grammar of each category, we generate 320 random formulas of up to a certain depth (7 operators) for each examination on each model instance. Then, we filter these generated formulas, in two steps to keep 16 of them:

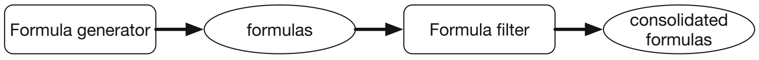


Fig. 1. Process to produce formulas in the MCC.

- First, we use SAT solving to filter out formulas that are equivalent to true or false independently of the model.
- Then, we pass formulas to SMC, an ad-hoc CTL bounded model checker specifically developed for the competition. If SMC is able to decide the satisfiability of a formula by examining only the first 1000 reachable states (using BFS exploration), then we discard the formula.

If too many formulas are filtered out, we stop filtering and just wait until we reach 16 formulas by random formulas (they thus may be easy to solve)—this happens in particular for models with less than 1000 reachable states.

Let us illustrate this process on the CTL and reachability formulas produced for surprise models in 2017. 175 360 formulas were produced by the formula generator. Then, only 8 768 consolidated formulas remained after filtering (which corresponds to 5% of the original formulas). Among these consolidated formulas, 8 548 are considered to be difficult[10] (97.5%).

For LTL, the second filter is not active yet and we only discard tautologic formulas by considering that atomic propositions are not trivial; this is less efficient than the process we have set-up for other types of formulas. So, our current focus is on providing a better filtering of LTL formulas (SMC, used to check for the complexity of the computation is specialized for CTL). We also aim at generating more realistic formulas (i.e., that would be as close as possible to human-written formulas).

3 Monitoring Environment and Experimental Conditions

Operating the MCC requires to run a tool many times (once per examination per model instance). There were 54 293 runs in 2013, 83 308 in 2014, 169 078 in 2015, 128 682 in 2016, and 91 710 in 2017. To control the increasing need for CPU, we decided to process tools on a subset of the model instances of the models: no model was discarded from the benchmark but most of the instances that could be processed by more than 70% of the tools in 2016 were discarded. Each run can last up to one hour of CPU.

Such a number of executions thus requires a dedicated software environment that can take benefits from recent multi-core machines and powerful clusters. Moreover, we need to measure key aspects of computation, such as CPU or peak memory consumption, in the least intrusive way. To achieve this in an automated way, we developed *B*ench*K*it [30], a software technology (based on QEMU) for

[10] Our ad-hoc CTL bounded model checker could not resolve it by exploring 1000 states.

measuring time and memory during tool execution. First used during the 2013 edition, *B*ench*K*it was regularly enhanced with new capabilities: dramatic reduction of the execution overhead (we need to boot a virtual machine for every run), the possibility to specify precisely the type of emulated processor (to avoid execution problems when several families of intel-compatible processors were used), and the support of multicore virtual machine to allow concurrent executions of tools.

In 2016 and 2017, the number of runs was reduced. In 2016, most tools only submitted one variant and, in 2017, only a subset of the benchmark was selected. On the one hand, this reduction only allowed CPU needs to stay stable despite the growth of the benchmark: 1549 days in 2015, 1481 in 2016 and 1547 in 2017. The main reason is that formulas are getting smarter (especially in the reachability and CTL examinations, extra work being required for LTL), and tools gradually support more examinations. To cope with this need for CPU, we used more machines kindly lent by their owners, namely:

– bluewhale03 and bluewhale07 (respectively 40 cores @ 2.8 GHz and 512 GB of memory and 32 cores @3.2 GHz and 1024 GB of memory) from the University of Geneva, Switzerland,
– Caserta (96 cores @ 2.2 GHz and 1024 GB of memory) from the University of Twente, The Netherlands,
– Ebro (64 cores @ 2.7 GHz and 1024 GB of memory) from the University of Rostock, Germany,
– quadhexa-2 (24 cores @ 2.66 GHz and 128 GB of memory) from the University of Paris Nanterre, France,
– small, 12 nodes (24 cores @ 2.4 GHz and 64 GB of memory each) out of the 23 of a cluster of machines at Sorbonne University, France.

Of course, to preserve a sound comparison between tools, runs were divided into several consistent subsets. All runs concerning a given model (i.e., all its instances) and for all the examinations were processed on the same computer.

Post-analysis scripts aggregate data, generate summary HTML pages as well as associated charts (there are 53 118), and compute scores for the contest. They are implemented using 15 kLOC of Ada and a bit of bash. *B*ench*K*it itself consists of approximately 1 kLOC of bash.

4 Participating Tools and Experimental Results

Nine tools participated in the 2017 edition of the Model Checking Contest: GreatSPN-meddly (Univ. Torino, Italy), ITS-Tools (Sorbonne Univ., Paris, France), LoLA (Univ. Rostock, Germany), LTSmin (Univ. Twente, The Netherlands), Marcie (Univ. Cottbus, Germany), smart (Iowa State Univ., USA), Spot (Epita, Le Kremlin Bicêtre, France), TAPAAL (Univ. Aalborg, Denmark) and Tina (LAAS/CNRS and Univ. Toulouse, France). Tina submitted two variants of the tool (in that case, the rules state that only the best variant is considered for a podium).

In this section, we first summarize the global results of the contest, together with our feedback (from Sects. 4.1–4.3). Then, each tool getting a gold or silver medal is briefly presented (Sect. 4.4 onwards).

4.1 Examinations in the Contest

Tools are confronted to several examinations: StateSpace, UpperBounds, Reachability, CTL and LTL. *StateSpace* requires the tool to compute the full state space of a specification and then provide information about it. Mandatory information concerns the number of states but tools may also provide additional information like the number of transitions, the maximum number of tokens per marking in the net and the maximum number of tokens that can be found in a place.

UpperBounds requires the tool to compute as a integer value, the exact upper bound of a list of places designated in a formula (there are 16 formulas per model instance).

Reachability, *CTL*, and *LTL* require the tool to evaluate whether formulas are satisfied or not. For each formula, we consider either state-based atomic propositions or transition-based atomic propositions (16 formulas of each type are provided per model instance). In the Reachability examination some formulas check for the existence of deadlocks.

4.2 Results – Podiums and Confidence Rate

Figure 2 shows the ranking of the three top tools in the various examinations proposed by the contest[11]. 100% represents the tool having scored the maximum points in the contest and followers' scores are expressed as percentages of this score. In the StateSpace examination, Tina.tedd was ranked second but the other variant was ranked 7th out of 8 participating tools. We can also note that, for the UpperBound and CTL examinations, the distance between the third tool and its followers was rather small. In the case of CTL, these tools rely on similar symbolic CTL technology. For UpperBounds, different technologies are used but clearly provide similar performances in the case of our benchmark.

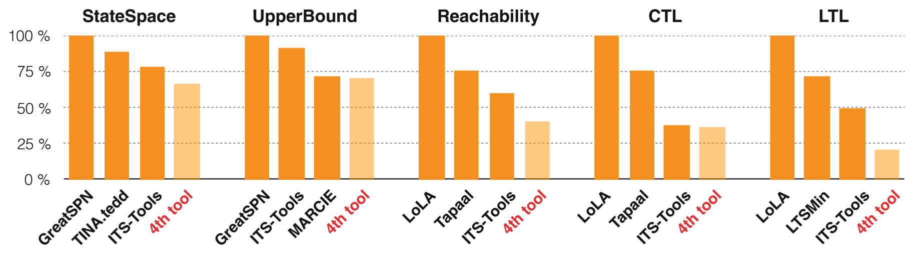


Fig. 2. Tools in the podium for each examination.

[11] Full results can be found at http://mcc.lip6.fr/2017/results.php.

One interesting issue of the contest is to evaluate the confidence improvement over the years. To do so, we introduced in 2015 the notion of confidence rate. This value is computed for each tool on the subset of results that are considered to be sound (a majority of at least three tools providing the same value). Then, among these values, the confidence rate is the ratio between the number of correct values and the number of computed values; this ratio is then converted into a percentage.

Table 2 shows the confidence rate of the participating tools for 2017. This year, we computed this rate both separately for each examination and globally for all the examinations together. The absence of value means the tool did not participate in the examination.

Table 2. Confidence rate of the participating tools in 2017.

Exam.	Global	StateSpace	UpperBounds	Reachability	CTL	LTL
GreatSPN-meddly	99.13%	100%	98.89%	99.18%	99.07%	–
ITS-Tools	96.91%	100%	100%	94.68%	100%	96.33%
LoLA	99.92%	–	100%	100%	99.62%	99.97%
LTSmin	100%	100%	100%	100%	100%	100%
Marcie	100%	100%	100%	100%	100%	–
smart	79.59%	79.59%	–	–	–	–
Spot	100%	–	–	–	–	100%
TAPAAL	100%	100%	100%	100%	100%	–
Tina.sift	97.84%	97.84%	–	–	–	–
Tina.tedd	100%	100%	–	–	–	–

Detailed results are provided at https://mcc.lip6.fr/2017. Let us highlight two interesting aspects: First, most errors were reported to be either in the model importation or formula importation (thus not in the verification algorithms). Some are also due to a divergence in the interpretation of some semantical points (it was the case in 2016 for some tools but no such evidence was detected in 2017). Second, over the years, tools are dramatically improving: from 89.65% in 2015, the average global confidence rate of participating tools moved to 94.20% in 2016 and then 97.34% in 2017. This can be considered as one of the major benefits of the contest for the community.

We note that importation errors are reducing year after year. This year, there was apparently a very few wrong values issued from erroneous implementation of the algorithmic heart of the tools. The lower confidence rate of smart this year is apparently due to some major changes in the architecture handled by master's students.

4.3 Involved Techniques

Tools developers were asked to report the techniques used by their tool. The result of this feedback is summarized in Table 3 for the three best tools in each category. Columns refer to techniques, except for the last three ones which show the type of execution: sequential, parallel in a portfolio mode (e.g., several techniques applied in parallel) or parallel (e.g., dedicated parallel algorithms).

Let us note that, while the winners for StateSpace and UpperBounds examinations are based on symbolic techniques, this is not true for the other examinations where explicit approaches (enriched with optimizations like state compressions, structural reductions or partial orders) are ranked before symbolic tools. This shows that explicit approaches, together with appropriate optimizations, still compete with symbolic tools (we will show more evidences later in this section).

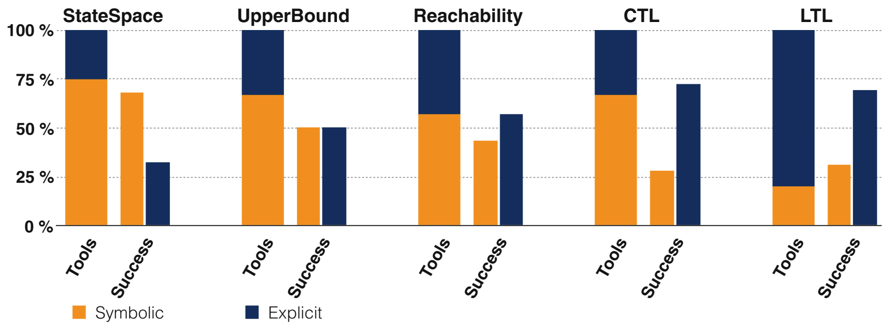
Table 3. Techniques activated by winning tools in 2017.

	Tool	Dec. diagrams	Explicit	SAT/SMT	State compression	Struct. reduction	Stubborn sets	Symmetries	Topological	Unfolding to P/T	Use of NUPNs	Sequential	Portfolio	Parallel
State space	GreatSPN-meddly	✓							✓	✓	✓	✓		
	Tina.tedd	✓			✓	✓			✓	✓		✓		
	ITS-Tools	✓							✓	✓	✓	✓		
Upper bound	GreatSPN-meddly	✓							✓	✓	✓	✓		
	ITS-Tools	✓							✓		✓	✓		
	Marcie	✓								✓		✓		
Reach-ability	LoLA		✓	✓	✓		✓	✓	✓		✓		✓	
	TAPAAL		✓		✓	✓	✓						✓	
	ITS-Tools	✓	✓	✓			✓		✓		✓		✓	
CTL	LoLA		✓		✓		✓		✓		✓		✓	
	TAPAAL		✓		✓								✓	
	ITS-Tools	✓							✓		✓	✓		
LTL	LoLA		✓		✓		✓		✓		✓		✓	
	LTSmin		✓		✓						✓			✓
	ITS-Tools	✓	✓		✓		✓		✓		✓		✓	

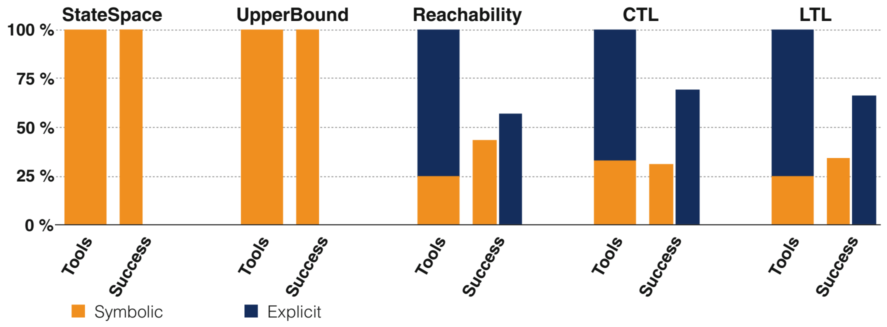
We also note that winning tools often combine several techniques, especially in portfolio mode. This shows a complementarity between the techniques and methods. So, depending on the models and/or the formulas, the best technique varies.

We also observe that the clustering of places that is provided for some models in the NUPN format is used by many tools (it is a way to extract information about the system's structure, for example to compute a better variable order to encode the state space with decision diagrams). A deeper analysis would be of interest to evaluate its impact on tool performances (unfortunately difficult with the currently collected data).

It is globally true that tools participating for several years regularly improve by getting fine-grained inputs from the increasing benchmark of the MCC. Nevertheless, Tina.tedd was globally well ranked for a first participation. At this stage, it is difficult to get more information since most tools so far declare the technique they use on the basis of the examination they compute, and not on the result they are processing. The explicit declaration of the techniques activated



(a) All participating tools



(b) Tools in the podium only

Fig. 3. Pondered analysis of symbolic and explicit tools. The first (large) bar in the charts shows the ratio between explicit and symbolic and the two (thin) ones the pondered success rate of each technique when answering examinations. 100% represents all tools or all the computed formulas.

result per result will be required in the next edition of the contest so that more accurate information could be gathered for analysis.

The study of the 2017 results can be completed by a pondered comparison between the use of symbolic techniques compared to the use of explicit model checking declared by all tools, and those declared by the tools in the podium (see Fig. 3). Since, depending on the examinations, there are sometimes more explicit tools than symbolic ones (and vice-versa), we pondered the techniques declared for correct answers according to the number of symbolic and explicit tools. In these charts, we consider only correct answers.

Figure 3b shows that, for StateSpace and UpperBound, tools in the podium are all symbolic. On the contrary, more tools in the podium rely on explicit techniques (in particular, ITS-Tools uses both, since it is a portfolio approach concurrently operating several techniques). However, an analysis of the results for computed formulas is necessary to refine this impression. Table 4 summarizes, for all examinations and for every categories of formulas, the number of satisfied and unsatisfied formulas. It also separates the results for all tools (including those in the podium) from those of the tools in the podium only.

Table 4. Dispatching of computed formulas in 2017 by all the tools in the left part, by tools in the podium only on the right part.

	For all tools			For podium tools		
	Reach.	CTL	LTL	Reach.	CTL	LTL
Satisfied formulas	43 599	29 946	15 251	28 921	19 512	13 521
Unsatisfied formulas	44 446	36 445	52 801	28 879	23 355	49 054
Total	88 045	66 391	68 052	57 800	42 867	62 575
Total formulas	222 488			163 242		

We can extract several lessons concerning formulas. First, it appears that 73% of the total computed formulas are produced by the tools in the podium. Second, it appears that most of the computed LTL formulas are unsatisfied (78%). This second fact is of interest because, it is a situation where explicit tools may be advantaged. This is probably related to the fact that, so far, the quality of LTL formulas is poor compared to the one of reachability and CTL ones. We are expecting progress in tackling this issue in the 2018 edition since we are working on a smarter LTL formula generator.

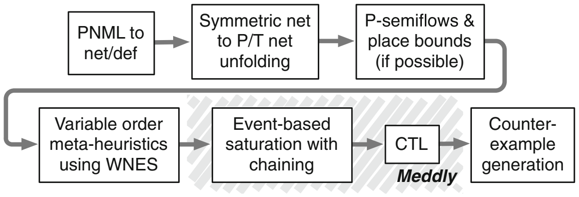
A last fact, not visible in the charts and tables, should be mentioned. Symbolic techniques are often associated with some sort of structural analysis (e.g., the use of NUPN information) to determine an efficient ordering of variables in the encoding of states. Similarly, explicit tools also operate compression techniques (e.g., partial orders).

4.4 GreatSPN-meddly

GreatSPN-meddly is a symbolic model checker that is part of the GreatSPN framework [3]. The main purpose of this tool consists in building the state space of a Petri net model using Decision Diagrams (DD), in order to verify reachability and CTL properties. All DD algorithms are implemented in the Meddly library[12].

GreatSPN-meddly operates on P/T nets with priorities, inhibitor arcs and marking dependent multiplicities. It also supports colored Petri nets through previous unfolding. A key strategy of the tool is the analysis of structural properties. P-semiflows are built before starting the state space, for the purpose of deriving place bounds and for some variable order heuristics. Unfortunately, P-semiflows cannot be built for all models (either because the model does not have any, or because it has an exponential number of them). Therefore, the availability of P-semiflows is optional. If place bounds are not known a priori, a heuristic strategy is used to guess them, with the possibility of restarting the saturation algorithm

[12] Meddly library: https://sourceforge.net/projects/meddly.


PNML to net/def
Symmetric net to P/T net unfolding
P-semiflows & place bounds (if possible)
Variable order meta-heuristics using WNES
Event-based saturation with chaining
CTL
Meddly
Counter-example generation

Method	W	P
Sloan 1/2	1.00	-
Sloan 1/16	1.15	-
Tovchigrecko/Noack	1.35	-
Cuthill-Mckee	1.01	-
P-chaining	1.00	y
Gradient-P	1.00	y
Force-P	0.50	y

Fig. 4. GreatSPN-meddly pipeline and WNES meta-heuristic table.

when the guessed bound is wrong. This is necessary because GreatSPN-meddly uses explicit encoding for transitions (MxD).

The tool includes a small collection of variable ordering heuristics [5], such as Force [2], Noack [23], bandwidth-reduction methods, and some methods explicitly written for GreatSPN-meddly (P-chaining and Gradient-P [4]). A meta-heuristic is also available, which computes multiple variable orders, scores them using a metric function, and selects the one with the smallest score. The tool uses a modified version of NES [5], called *Weighted* NES (WNES). The WNES value of each method is obtained from the NES score (sum of transition spans) multiplied by a method-dependent weight W. Figure 4 shows the meta-heuristics methods table, with the WNES weight W and if the algorithm requires P-semiflows. From the results of the MCC'2017 context, it appears that the new heuristic Gradient-P is very effective. Algorithm weights have been devised empirically.

The model state space is built using the Meddly library, which implements Saturation with chaining using events split by levels. Meddly employs fully-reduced DD for the state space, and identity-reduced DD for the transition relations. Once the state space is built, the tool may evaluate CTL properties on it. CTL formulas are read and evaluated one by one, in sequence. CTL evaluation may optionally produce a tree-like trace for ECTL formulas, in order to present counter-examples (or witnesses) of the evaluated formulas. In order to ensure correctness, the tool was run on thousands of formulas and models provided by the previous MCC editions on a cluster [1]. The availability of a large set of models from previous MCC editions has been used to fix various bugs in model handling (degenerate models, different interpretation of the CTL semantics for dead states, etc.).

Reported Strengths for 2017. The success of GreatSPN-meddly in the MCC'2017 edition is due to several factors. First, various improvements in the Meddly library made the Saturation algorithm performance significantly better. This improved the scalability of the tool on many large models that could not be computed previously. Secondly, the heuristic variable order strategy is more solid, allowing to find reasonably good variable orders for many models. In particular, some algorithms (Sloan [35], Noack and Gradient-P) proved to be very effective in handling the variety of models of the MCC benchmark [5]. Experience shows that no single heuristic is good enough to treat all models, and a good meta-heuristic is needed. While reasonably good, the current meta-heuristic fails on about 25% of the models. More research on good variable ordering metrics is still needed.

Handling the 2017 "Surprise" Models. GreatSPN-meddly does not exploit precomputed data for models, therefore known and surprise models are treated alike. The rational is that the current strategy, statistically evaluated on the known models set, should be "good enough" for arbitrary models. Of course, this is not always true, and in fact the tool failed, for instance, on the BART model.

Lessons Learned from the Contest. GreatSPN-meddly was prepared for the context by running a large benchmark [5] to test the correctness of the implementation (comparing results with the previous edition values) and the effectiveness of the various heuristics implemented by the tool. This benchmark proved to be of significant value, both to understand the actual performance of many known variable order heuristics, and to test new ones (Gradient-P). The availability of the model set and the previous editions' results is therefore of great value for tool development and testing, and its value goes beyond the MCC competition.

4.5 ITS-Tools

ITS-tools [40] is a model-checker supporting both multiple solution engines and multiple formalisms using an intermediate pivot called the Guarded Action Language (GAL). Both colored models and P/T models are translated to GAL. Properties are translated conjointly with the model to properly trace atomic propositions to their target expressions in GAL.

For colored models, ITS-tools rely on PNMLFW (http://pnml.lip6.fr/) a Java reference implementation of the PNML standard to parse the model, then translates the model to a Parametric GAL model (which is roughly the same size as the colored model). During this process, a decomposition of the system is inferred from the color domains, trying to maximize existence of similar sub components. The parametric GAL model is then instantiated to a plain GAL model, but this process exploits binding symmetry as well as sequences of alternatives that provide compact transition representations. As a result, the GAL model obtained is often orders of magnitude smaller than an equivalent P/T (but is semantically equivalent).

For P/T models, ITS-tools embed a simple parser in SAX, then translate to a plain GAL specifications (roughly the same size as the P/T). If NUPN information is available, it also build a corresponding decomposition of the system into an *Instantiable Transition System* (ITS, which gives the tool its name).

After this translation step, ITS-tools obtain a GAL model embedding a set of properties, and possibly a decomposition of the system. At this stage, ITS-Tools operate a number of basic structural reductions before going to state-space exploration. This is effective at removing constant marking places, detecting test arcs, redundant transitions... ITS-tools also test and simplify properties that can be decided immediately, either by looking at the initial state or when simplification yields a tautology for true or false.

Symbolic Engine. The resulting model is analyzed with our core symbolic solution engine based on Hierarchical Set Decision Diagrams (SDD) [9]. This engine exploits all the features of GAL, including hierarchy and the sequential composition semantics, so that small GAL models (e.g., as produced by our colored translation path) are often checked very efficiently. The engine benefits greatly from the model decomposition into ITS (colored or NUPN models). ITS-tools use a single variable ordering heuristic, based on Force [2]. All examinations are supported on this symbolic engine, and ITS-Tools did reach 100% confidence for answers from this solution engine.

For place bounds and reachability queries, one can simply navigate the SDD representing the full reachable state space, computed using advanced symbolic algorithms including variants of *saturation*.

For CTL, ITS-tools operate a translation to a forward CTL formula where possible, and use variants of constrained saturation to deal with EU and EG operators. ITS-Tools use a general yet precise symbolic invert to deal with predecessor relationships when translation to forward form is not feasible. Some early detection of emptiness was implemented, that helps reduce the workload for simpler formulas (there are yet many in the contest).

For LTL, ITS-tools rely on Spot [13] to translate the properties to Büchi variants, then use our SLAP hybrid algorithm [12] to perform the emptiness check. This algorithm leverages both the desirable on-the-fly properties of explicit techniques and the support for very large Kripke structures (state spaces) thanks to the symbolic SDD back-end. All symbolic operations benefit from state-of-the-art saturation variants where feasible.

SAT Modulo Theory Engine. ITS-tools implements translations of the semantic bricks of GAL to SMT predicates enabling the use of solvers such as Z3 (Microsoft) or Yices2 to answer a variety of questions on the model. The encoding of GAL semantics assumes that the model is deterministic however, so the tool needs to determinize the GAL model (an operation that can be explosive for models coming from colored nets) before using the SMT solver. ITS-tools encode the constraints that reflect transition steps, as well as a basis of P-invariants computed using a classic algorithm[13]. If the net is one-bounded (presence of NUPN information) that is also encoded.

With these bricks, ITS-tools implements a bounded-model checking (BMC) decision procedure able to assert that an invariant is contradicted at some depth of exploration. This task is run in parallel with a K-Induction decision procedure able to assert that an invariant holds. These decision procedures are currently rarely competitive with the SDD engine, but they are very good at detecting structurally unfeasible behavior (i.e., K-Induction at step 1 answers many queries), and BMC works well when short counter-example traces exist.

[13] Adapted from the APT tool https://github.com/CvO-Theory/apt.

These strategies also deal with unbounded nets, though there are very few of these in the contest.

Explicit and Partial Order Reduction Engine. ITS-tools implement a translation from GAL to PINS [29], the format used by the LTSmin tool set. This enables the use of their many verification strategies on GAL models. ITS-tools uses this translation to participate in the reachability and LTL categories only; the only engine for CTL in LTSmin is symbolic which seems redundant with respect to the SDD solution. Mostly the authors wanted to complement their symbolic methods with an explicit solution, that might deal well with models where symbolic techniques fail (e.g., due to not finding a good variable order). To be fair (it is also a competitor), LTSmin is severely limited in its resource usage. ITS-tools focused this year on enabling partial order reduction in LTSmin (POR) if possible (reachability and stuttering invariant LTL properties). To this end the SMT translation bricks were reused to compute event dependency information using the SMT solver.

Further information on the tool is available from http://ddd.lip6.fr.

Reported Strengths for 2017. ITS-tools are a mature toolset, able to compete in every single category of the contest including colored nets, and place on the podium in each of them. When possible, we ran the various solution engines in portfolio mode, dedicating up one thread and most memory to SDD, two threads and negligible memory for SMT, and (only) one thread with bounded memory for LTSmin.

ITS-Tools main strength remains its symbolic solution engine based on SDD, which historically held the gold in StateSpace for a long time. The SMT solution does answer some queries very fast in many cases, but these are often relatively trivial properties *in fine*. The explicit engine suitably complements the other solutions, answering many complex queries easily when POR can be activated.

Handling the 2017 "Surprise" Models. ITS-tools treat surprise models exactly like known models, except for Philosophers and SharedMemory models that were manually rebuilt to take advantage of ITS characteristics. The surprise models were handled relatively well by the SDD engine, yielding overall quite good results on this category this year.

Lessons Learned from the Contest. Unfortunately the integration of LTSmin within ITS-tools was buggy in certain cases (bad import/export) so that reliability in both Reachability and LTL was negatively impacted. These problems have been patched. In the post-analysis of the results, the authors detected some unexpected performance bottlenecks that could be solved (LTL translation, time to compute dependency matrices for LTSmin).

The competition drives the development of better and more efficient tools, certainly the authors were inspired by discussing with the other competitors and will integrate some of their ideas by next year, starting with GreatSPN's new meta-order heuristics [5].

4.6 LoLA

LoLA 2.0 is a model checker based on explicit state space exploration and structural Petri net methods. In the reachability competition, LoLA employs a portfolio consisting of state space generation (with stubborn set reduction), the state equation technique [41], a SAT based siphon/trap check [36], and a random walk procedure. In all other categories, LoLA used standard explicit model checking algorithms. In 2017, stubborn sets were added to the LTL and CTL categories. In the deadlock category, LoLA used the symmetry reduction as well [38].

Reported Strengths for 2017. In 2017, LoLA's developers enabled all the procedures to directly cope with FIREABLE predicates (instead of translating them to place-based predicates). In consequence, LoLA won all fireability subcategories in the MCC 2017. In the CTL and reachability competitions (counting only P/T nets), the advance in the fireability subcategory was large enough to compensate a backlog in the cardinality subcategory.

Many results for LoLA (especially in the reachability category) were produced by the non-standard techniques (symmetry, state equation, siphon/trap, and random walk). This way, LoLA earned many points for solving problems. In several cases, state space search would have solved the problems, too. However, the quick answers by the nonstandard procedures gained a lot of bonus points, and saved time that could be transferred to the remaining problems in an examination. LoLA used an enhanced time management for scheduling the available run time to the 16 queries of an examination.

In CTL, LoLA benefitted from its formula preprocessing. Several problems were found to be equivalent to a reachability query, so LoLA could use the much more advanced reachability portfolio. Points gained this way were just enough to stay ahead of TAPAAL.

In LTL, the model checker was completely re-implemented, for the purpose of removing the semantic discrepancies to other tools. The new model checker uses the acceptance condition proposed in [21] which permits much earlier termination in the FALSE case. In fact, about 80% of answers given by LoLA on LTL queries were FALSE answers.

Moreover, LoLA changed the translation of LTL formulas to Büchi automata. In particular, large Boolean combinations of atomic propositions are now treated as single atomic propositions. This way, a lot of translation time is saved. The resulting Büchi automata (hence, the resulting state space) becomes much smaller. These changes propelled LoLA from 70% success (in 2016) to more than 90% in 2017.

The state equation is responsible for more than 50% of the UNREACHABLE answers and quite some REACHABLE answers. Compared to 2016, the transformation of the reachability problem to a disjunctive normal form (DNF) required for the state equation was re-implemented. Instead of using the slow term rewriting system mentioned above, LoLA processes the transformation directly. In this course, LoLA added a simple static analysis that permits the detection of duplicates in the subformulas. This way, the resulting DNF is much smaller and LoLA can handle several problem instances that ran out of memory before.

Handling the 2017 "Surprise" Models. LoLA reads Petri net models in its own file format. Hence, translating the PNML [24] files of the MCC has always been an issue. Thanks to a translator provided by Silvano Dal Zilio (Toulouse), LoLA was able for the first time to process colored Petri nets. Furthermore, LoLA replaced a Python script for translating PNML to the LoLA format with a parser based on flex and bison. This way, translation time of some large net input files could be reduced from more than an hour to a few seconds. Unfortunately, the script did not anticipate all extensions of PNML, so a translation error caused the loss of all the scores for one of the surprise nets. Finally, LoLA has a new preprocessor for formulas. Huge conjunctions and disjunctions are no longer handled by a (too slow) term rewriting system. Instead, LoLA processed these subformulas with dedicated routines. In consequence, LoLA had more time in 2017 to work on the actual verification tasks.

Lessons Learned from the Contest. To win a category, it is necessary to have strong verification techniques. Under the conditions of the contest, explicit techniques appear to be ahead of symbolic methods. Thanks to the on-the-fly principle, it seems to be easier for explicit model checkers to earn the low hanging fruits. LoLA ranked only few points ahead of its strongest competitors. Hence, it becomes more and more important to look into data structures and algorithms and to remove all the unnecessary computations. The contest is an excellent motivation for investing time into LoLA. LoLA benefits a lot from the increased confidence (the large benchmark identified remaining bugs) and the efforts spent on better performance. At the same time, success in the contest considerably increases the visibility of LoLA.

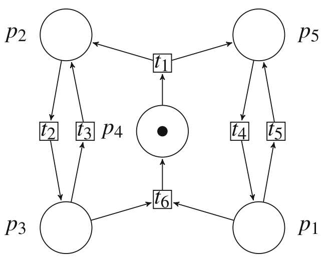
4.7 LTSmin

LTSmin[14] [29] has competed in the MCC since 2015. Already in the first editions, LTSmin participated in several subcategories, while since 2017 LTSmin competes in all subcategories, except for colored Petri nets, and reporting the number of fireable transitions in the marking graph.

LTSmin has been designed as a language independent model checker. This allows developers to reuse algorithms that are already implemented for other languages, such as Promela and mCRL2. For the MCC, the developers only needed to implement a PNML front-end and translate the MCC formula syntax. Improvements to the model checking algorithms, like handling integers in atomic formulas, can now in principle also be used in other languages.

LTSmin's main interface is called the Partitioned Interface to the Next-State function (PINS) [29]. Each PINS language front-end needs to implement the next-state function. It must also provide the initial state, and a dependency matrix (see below). The multi-core explicit-state and multi-core symbolic model checking back-ends of LTSmin use this information to compute the state space on the fly, i.e., new states and atomic predicates are only computed when necessary for the algorithm.

[14] http://ltsmin.utwente.nl.



$$\begin{array}{c|ccccc}
 & p_1 & p_2 & p_3 & p_4 & p_5 \\
\hline
t_1 & 0 & \mathbf{1} & 0 & \mathbf{1} & \mathbf{1} \\
t_2 & 0 & \mathbf{1} & \mathbf{1} & 0 & 0 \\
t_3 & 0 & \mathbf{1} & \mathbf{1} & 0 & 0 \\
t_4 & \mathbf{1} & 0 & 0 & 0 & \mathbf{1} \\
t_5 & \mathbf{1} & 0 & 0 & 0 & \mathbf{1} \\
t_6 & \mathbf{1} & 0 & \mathbf{1} & \mathbf{1} & 0
\end{array}$$

Fig. 5. Example model: Petri net (left) and dependency matrix (right)

A key part of LTSmin are its dependency matrices. Dependency matrices must be precomputed statically by the front-end, and are extensively used during reachability analysis and model checking. An example Petri net, with its dependency matrix, is given in Fig. 5. Here transition t_1 does not depend on p_3 or p_1 in any way. Also for properties, a dependency matrix (computed by LTSmin) indicates on which variables each atomic predicate depends. For instance, the dependency matrix of some invariant is shown in Fig. 6. This invariant demonstrates LTSmin's new native property syntax. A finer analysis that distinguishes read- and write-dependencies [34] pays off, in particular for 1-safe Petri nets.

The MCC's reachability and CTL categories are tackled with the multi-core symbolic back-end of LTSmin, which relies on the multi-core Decision Diagram framework Sylvan [11]. Consequently, in these categories, typically billions of states in the marking graph can be explored. The LTL category is handled by the multi-core explicit-state back-end, relying on efficient parallel SCC decomposition [8].

Reported Strengths for 2017. In the reachability analysis categories, LTSmin competes using the symbolic back-end pnml2lts-sym, handling enormous state spaces by employing decision diagrams. However, good variable orders are essential. LTSmin provides several algorithms to compute good variable orders, which operate on the transition dependency matrix, for instance Sloan's algorithm [35] for profile and wavefront reduction. LTSmin computes the marking graph symbolically and outputs its size. To compete in the UpperBounds category, LTSmin maintains the maximum sum of all tokens in all places over the marking graph. This can be restricted to a given set of places (using, e.g., `--maxsum` $= p_1 + p_2 + p_3$). For the ReachabilityDeadlock category, the symbolic tool performs deadlock checking on the fly. Also invariant checking (the approach for ReachabilityFireability, and ReachabilityCardinality) is performed on the fly. Invariants can be specified through the `--invariant` option.

LTSmin is unique in the application of multi-core algorithms for symbolic model checking. In particular, both high-level algorithms (exploring the marking graph, and traversing the parse tree of the invariant), as well as low-level algorithms (decision diagram operations) are parallelized. This form of true con-

$$AG(1 \leq p_2 + p_3 \wedge 1 \leq p_5 + p_1) \qquad \begin{array}{c} \\ 1 \leq p_2 + p_3 \\ 1 \leq p_5 + p_1 \end{array} \begin{array}{c} \begin{array}{ccccc} p_1 & p_2 & p_3 & p_4 & p_5 \end{array} \\ \begin{pmatrix} 0 & \mathbf{1} & \mathbf{1} & 0 & 0 \\ \mathbf{1} & 0 & 0 & 0 & \mathbf{1} \end{pmatrix} \end{array}$$

Fig. 6. Example invariant property and the dependency matrix on its atomic predicates

currency allows LTSmin to benefit from the four CPU cores made available in the MCC, instead of a portfolio approach.

The approach LTSmin uses to compete in the CTL model checking categories builds upon symbolic state space generation. After the marking graph is constructed symbolically, the CTL model checking starts. LTSmin's symbolic back-end employs a straightforward μ-calculus model checker with both high-level and low-level parallelism. Similarly to invariant checking the parse tree of the μ-calculus formula is traversed in parallel, as well as the low-level decision diagram operations. LTSmin translates CTL* (a strict superset of CTL) to μ-calculus using tableaux, and evaluates the translated formula. CTL model checking in LTSmin is triggered by the `--ctl` option to `pnml2lts-sym`. For LTL model checking, LTSmin uses explicit-state model checking. So here, each reachable marking of the Petri net is considered individually. However, advanced search techniques are employed to find counter-examples as quickly as possible. In particular, the parallel SCC decomposition using a concurrent Union-Find structure [8]. The multi-core explicit back-end of LTSmin constructs the cross product of the marking graph and Büchi automaton on the fly. This approach brought us at the first place in 2016, and second in 2017. LTL model checking is invoked as pnml2lts-mc with the `--ltl` option.

Handling the 2017 "Surprise" Models. LTSmin takes no special measures for handling the 2017 "Surprise" models. In fact, LTSmin handles both "Known", and"Stripped" as "Surprise" models. Handling models as if they are never seen before is in line with the philosophy that LTSmin should be a language independent model checker. If (precomputed) structural information of models is to be exploited, LTSmin should be able to do so for every specification language. So far structural information in Petri nets is not generalized to cover other languages too, except for the dependency matrix.

Lessons Learned from the Contest. The CTL back-end can still be improved by specializing the μ-calculus model checker. The surprise models revealed that the parallel algorithms of LTSmin are still lacking full saturation. This is expected to be improved before the 2018 edition of the MCC.

4.8 TAPAAL

TAPAAL [10] is a platform-independent tool suite for modeling, simulation and verification of Petri nets, and their timed extension called timed-arc Petri nets. The tool offers a graphical user interface for compositional design of Petri net models (the different components communicate via shared places or shared transitions), a powerful simulation and trace visualization mode, and a graphical query creation dialog that allows to call three different engines distributed together with TAPAAL: a continuous time engine `verifytapn`, a discrete time engine `verifydtapn`, and the untimed verification engine `verifypn` that participated in the model checking contests in the years 2014–2017. The tool also offers the option to translate timed-arc Petri net models into networks of timed automata and automatically call the UPPAAL engine as its back-end. It is possible to import Petri net models in the PNML format, together with the XML queries used in the model checking competition, as well as to export created nets and formulas in these exchangeable formats. TAPAAL can be downloaded at www.tapaal.net.

The untimed engine `verifypn` of TAPAAL performs an explicit-state model checking of weighted P/T nets with inhibitor arcs and it currently supports deadlock detection, reachability analysis of cardinality and fireability propositions, and more recently also verification of CTL formulas. TAPAAL moreover participates in the computation of upper-bounds and state-space exploration, even though these types of analysis are not in the main focus of the developers. Colored nets are not supported at the moment.

Reported Strengths for 2017. Compared to the 2016 version of the `verifypn` engine of TAPAAL (described in detail in [27]), the 2017 version comes with six main novelties: (i) a brand new successor-generator, (ii) a new data-structure for compressing and storing the state-space, (iii) advanced formula preprocessing using linear programming, (iv) siphon-trap technique for detecting deadlock freedom, (v) improved structural reductions, and (vi) newly implemented partial order reduction for the reachability analysis.

The novelty (i) can be seen as a pure refactoring, with a focus on creating a computationally lightweight and cache-friendly way of generating successors of markings. The faster successor generator also means that larger portions of the state-space can be explored within the given time limits. Hence it reduces the higher memory consumption by compressing the encodings of markings and storing them in a novel tree-like data-structure (ii) called PTrie [28], allowing the tool to efficiently share prefixes of the encoded markings. Early experiments with PTrie were done already in 2016 where the tool developers submitted the experimental version TAPAAL-exp that significantly improved the scores of the tool. Recent improvements of PTrie made the data-structure comparable in performance with state-of-the-art hashmaps, while having a significantly lower memory footprint [28]. In 2017 both the new successor generator as well as PTrie were used also in the CTL model checking.

In 2017, some of the methods originally introduced in [27] were revisited: the heuristic search-strategy has been marginally improved but most importantly, the developers refined and generalized the over-approximation technique from [27] based on state equations and linear programming. In novelty (iii), the tool uses these principles to recursively simplify reachability and CTL formulas, often resulting in significantly smaller or even trivially true/false formulas. The linear programming approach is also used to encode the (iv) siphon-trap property [36] that allows in some cases to show that a net is deadlock free without exploring its state-space. Regarding the novelty (v), the developers generalized and extended the structural reduction rules, taking greater care of weighted arcs and inhibitor arcs. Finally, for the novelty (vi), there is a new implementation of the classical partial order reduction technique based on stubborn sets.

Handling the 2017 "Surprise" Models. As in previous years, the developers took no special measures towards the surprise models. Both known and surprise models were attempted to be solved using formula preprocessing and different search-strategies run in parallel. Timeouts for formula preprocessing and for structural reduction were introduced, limiting their execution time to about one minute. The queries that were not solved in the first parallel phase where then sequentially verified one by one until the one hour time limit was reached.

Lessons Learned from the Contest. In 2017 as well as in 2016, TAPAAL received the second place after LoLA in both the reachability and CTL categories. Given the improvements listed above, there was a hope to challenge LoLA's first place in 2017, however, due to the fact that LoLA started in 2017 to support colored nets that are equally counted in the reachability and CTL categories as the P/T nets, the margin between TAPAAL and LoLA remained similar as in 2016.

It was also realized that the improvement by introducing a stubborn set reduction was not as significant as expected because the input nets in TAPAAL are already preprocessed by structural reduction techniques. Still there is a reasonable gain in combining both techniques. Currently, TAPAAL applies stubborn set reduction only for the reachability analysis but not for CTL model checking, while LoLA introduced in 2017 stubborn sets also for CTL formulas.

Post-contest analysis revealed that a number of small bugs in the newly introduced features of the tool had a significant impact on the tool performance. Most notably a wrong ordering of places in the successor generator and too "loose" over-approximation in the formula preprocessing led to a non-negligible loss of points. As the discovered errors had no impact on the correctness of the tool, these bugs were not noticed when preparing TAPAAL for the 2017 competition.

4.9 Tina

Tina (TIme Petri Net Analyzer) [7] is a toolbox for the editing and analysis of various extensions of Petri nets and Time Petri nets, developed at LAAS-CNRS. It provides a wide range of tools for state space generation, structural analysis, model checking, editing or simulation. Except for the graphic editor, all tools of the toolbox are developed in Standard ML (SML), a high-level and modular functional programming language.

For Tina's first participation to the MCC, two tools were proposed, `sift` and `tedd`, both to compete in the *State Space* category. The `sift` tool has been part of Tina for several years; it implements state-of-the-art enumerative techniques for state space analysis of Petri nets and Time Petri nets. `tedd` is a new symbolic (logic-based) analysis tool for Petri nets, soon to be enriched to handle Time Petri nets and included in Tina.

Reported Strengths for 2017. `tedd` results from the integration of four different components: a library handling Hierarchical Set Decision Diagrams (SDD) [9], a module providing a large choice of variable order heuristics, a preprocessor (shared with `sift`) providing structural reductions, and a tool unfolding high-level colored nets into equivalent P/T nets.

Although still in development, the SDD component has shown competitive performances, on par with most of the symbolic tools present in the contest. The variable order module provides a rich choice of variable orders based on net traversals, semi-flows, flows or names. The Force [2] heuristics can be used to improve any basic order. Hierarchical orders are available but have been seldom used so far.

The single component having most contributed to Tina's results is certainly the structural reduction preprocessor. The preprocessor applies a set of rules (in the style of [6]) aimed at reducing the number of places and transitions of the net while preserving its marking count. The transformations performed consist of removing redundant places, duplicated or statically dead transitions, or transforming start places. Simultaneously, a trace information is recorded so that the number of transitions and token counts of the original net can be reconstructed from those of the reduced net. Though a number of them cannot be reduced at all, many models of the contest can be significantly simplified this way, some drastically with up to 90% of the places removed. This benefits computation of the state space by the SDD module since it has less variables to handle.

The unfolding tool for high-level nets proved convenient in most cases, but unfolding some models yielded a tremendous number of transitions that makes the approach inapplicable. Those high-level models clearly require native enumeration methods.

Handling the 2017 "Surprise" Models. After preprocessing,"Known" models are analyzed using a variable order determined experimentally; no single variable order fits all models. For other types of models, a small number of variable orders likely to work are chosen from the experiments on "known" models. Analysis proceeds by trying to compute the state space using each of these orders for some amount of time, one after the other. These attempts could have been performed in parallel, with statically partitioned storage, but it was feared that some runs could be short of storage; instead a sequential strategy was chosen. The rationale is that if a variable order is adequate, then computation of the state space is fast and should complete in the amount of time allocated provided sufficient storage is available.

After computation of markings, it is proceeded with computation of the number of transitions. Since SDD do not require to precompute the transition relation, this does not simply amount here to compute the number of paths of some single decision diagram. Instead, an iterative algorithm is used, handling one transition at a time, with some weighting to take duplicated transitions into account. The method proved effective but, on some models, counting transitions has been measured orders of magnitude slower than building their state space. Another ad hoc algorithm has been devised to compute token counts in presence of reductions. The tokens found in redundant places are reconstructed from those found in the remaining places using the trace information of reductions. This could be done efficiently.

Lessons Learned from the Contest. Tina first participation to the contest was found costly in time. Notably, the tools had to be enriched to handle queries specific to the contest like token counts, and we had to develop from scratch a tool for handling high-level colored nets. Yet, thanks to the large number of considered models and their variety, the contest is a unique occasion to test thoroughly and strengthen Tina.

It is nice to notice that Tina, developed in a mostly functional language (which is atypical for model-checking tools), reached a competitive performance level. From the results, development of an efficient structural reduction tool revealed a wise choice. Reductions proved quite useful associated with both `tedd`, a symbolic tool, and `sift`, an enumerative one.

Tina developers conclude with some observations about execution on the virtual machine (VM) in which all tools of the contest are run. The VM does not seem to alter significantly processing times when compared with native executions. However Tina appear to be doubly penalized on storage consumption. The code compiled for Tina embeds a runtime providing automatic memory allocation with garbage collection. A first penalty is that, from the way it operates, the runtime system always requests more memory than needed by the applications. A second is that garbage collection heavily stresses the virtual memory management of the underlying machine, and that virtual memory management by the VM seems to be significantly slower than on the native machine. It was observed on some models a time overhead of over 100%, mostly due to memory management.

Authors. The main architect of the Tina toolbox is Bernard Berthomieu, with numerous insights and contributions by past and present members of the VERTICS team at LAAS, including Silvano Dal Zilio, Didier Le Botlan, François Vernadat, Alexandre Hamez and Pierre-Alain Bourdil. The SDD component of `tedd` has been written by Alexandre Hamez, also the author of `pnmc` [22]. `tedd` and `pnmc` do not share any code but rely on the same SDD technology.

5 Conclusion

This paper presented the outcomes of the 2017 edition of the Model Checking Contest, the detailed results of which can be found at http://mcc.lip6.fr/2017. The positive impact of the Model Checking Contest on the scientific community is demonstrated in at least two ways: the collection of models accumulated during the MCC is being used and cited in a growing number of publications[15], and the

[15] See http://mcc.lip6.fr/bibliography.php.

confidence level of the participating tools has been constantly increasing since 2015.

For the next editions of the Model Checking Contest, we plan to clarify or evolve certain rules of the contest. Such changes will be mostly based on the feedback received from tool developers. To this aim, a poll has been organized, which gathered answers from 80% of the developers of the tools that participated in the 2017 edition of the MCC. Among the most debated changes, one can mention: *(i)* a clarification of the meaning of transition counts in the StateSpace examination; *(ii)* the possibility to add new, larger instances for those models all the instances of which are simple enough to be solved by all the tools; *(iii)* the introduction, in the PNML files, of structural information about the models (e.g., structurally 1-bounded, simple free choice, strongly connected, etc.), so that tools can rely on such properties to increase efficiency by using dedicated algorithms; *(iv)* the removal of the notion of "stripped" (or "scrambled") models that was used in former editions of the MCC; *(v)* the status of colored nets with respect to P/T nets, and the way temporal-logic formulas are generated for P/T nets that are unfolded from a colored net; *(vi)* the decision whether to disclose in advance or not, to the tool developers, the temporal-logic formulas generated every year for the known models; *(vii)* the preventive measures that could be taken to avoid any bias in performance assessment that might arise from the use of virtual machines; and *(viii)* the requirement that tools should generate, in a common format to be agreed upon, counterexamples (e.g., execution traces) when a temporal formula evaluates to false—such counterexamples may be useful in the rare but definite situations where tools provide diverging answers for a given model-checking problem.

Other points of the discussion more specifically deal with the MCC scoring rules themselves. One can mention: *(ix)* provisions to ensure that parameterized models with many instances do not take excessive weight in the competition; *(x)* determination of the respective weights of known vs surprise models for the next MCC editions; *(xi)* assessment of the bonus points granted so far to the participating tools that compute fast or use less memory; and *(xii)* proposals that would provide incentives for "newcomers", i.e., participating tools that would enter the MCC competition for the first time. We expect that these various measures will noticeably improve the next editions of the MCC and will, perhaps, lead to significant changes in the future podiums.

References

1. Aldinucci, M., Bagnasco, S., Lusso, S., Pasteris, P., Vallero, S., Rabellino, S.: The open computing cluster for advanced data manipulation (OCCAM). In: 22nd International Conference on Computing in High Energy and Nuclear Physics, San Francisco (2016)
2. Aloul, F.A., Markov, I.L., Sakallah, K.A.: FORCE: a fast and easy-to-implement variable-ordering heuristic. In: ACM Great Lakes Symposium on VLSI, pp. 116–119. ACM (2003)

3. Amparore, E.G.: A new GreatSPN GUI for GSPN editing and CSL^{TA} model checking. In: Norman, G., Sanders, W. (eds.) QEST 2014. LNCS, vol. 8657, pp. 170–173. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10696-0_13
4. Amparore, E.G., Beccuti, M., Donatelli, S.: Gradient-based variable ordering of decision diagrams for systems with structural units. In: D'Souza, D., Narayan Kumar, K. (eds.) ATVA 2017. LNCS, vol. 10482, pp. 184–200. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68167-2_13
5. Amparore, E.G., Donatelli, S., Beccuti, M., Garbi, G., Miner, A.: Decision diagrams for Petri nets: which variable ordering? In: Petri Net Performance Engineering conference (PNSE), pp. 31–50. CEUR-WS (2017)
6. Berthelot, G.: Transformations and decompositions of nets. In: Brauer, W., Reisig, W., Rozenberg, G. (eds.) ACPN 1986. LNCS, vol. 254, pp. 359–376. Springer, Heidelberg (1987). https://doi.org/10.1007/978-3-540-47919-2_13
7. Berthomieu, B., Ribet, P.O., Vernadat, F.: The tool TINA-construction of abstract state spaces for Petri nets and Time Petri nets. Int. J. Prod. Res. **42**(14), 2741–2756 (2004)
8. Bloemen, V., van de Pol, J.: Multi-core SCC-based LTL model checking. In: Bloem, R., Arbel, E. (eds.) HVC 2016. LNCS, vol. 10028, pp. 18–33. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-49052-6_2
9. Couvreur, J.-M., Thierry-Mieg, Y.: Hierarchical decision diagrams to exploit model structure. In: Wang, F. (ed.) FORTE 2005. LNCS, vol. 3731, pp. 443–457. Springer, Heidelberg (2005). https://doi.org/10.1007/11562436_32
10. David, A., Jacobsen, L., Jacobsen, M., Jørgensen, K.Y., Møller, M.H., Srba, J.: TAPAAL 2.0: integrated development environment for timed-arc Petri nets. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 492–497. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28756-5_36
11. van Dijk, T., van de Pol, J.: Sylvan: multi-core decision diagrams. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 677–691. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_60
12. Duret-Lutz, A., Klai, K., Poitrenaud, D., Thierry-Mieg, Y.: Self-loop aggregation product—a new hybrid approach to on-the-fly LTL model checking. In: Bultan, T., Hsiung, P.-A. (eds.) ATVA 2011. LNCS, vol. 6996, pp. 336–350. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24372-1_24
13. Duret-Lutz, A., Lewkowicz, A., Fauchille, A., Michaud, T., Renault, É., Xu, L.: Spot 2.0—a framework for LTL and ω-automata manipulation. In: Artho, C., Legay, A., Peled, D. (eds.) ATVA 2016. LNCS, vol. 9938, pp. 122–129. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46520-3_8
14. Evrard, H.: DLC: compiling a concurrent system formal specification to a distributed implementation. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 553–559. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_34
15. Evrard, H., Lang, F.: Formal verification of distributed branching multiway synchronization protocols. In: Beyer, D., Boreale, M. (eds.) FMOODS/FORTE -2013. LNCS, vol. 7892, pp. 146–160. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38592-6_11
16. Evrard, H., Lang, F.: Automatic distributed code generation from formal models of asynchronous processes interacting by multiway rendezvous. J. Log. Algebraic Methods Program. **88**, 121–153 (2017)
17. Garavel, H.: Nested-unit Petri nets: a structural means to increase efficiency and scalability of verification on elementary nets. In: Devillers, R., Valmari, A. (eds.)

PETRI NETS 2015. LNCS, vol. 9115, pp. 179–199. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-19488-2_9

18. Garavel, H., Lang, F., Mateescu, R., Serwe, W.: CADP 2011 a toolbox for the construction and analysis of distributed processes. Int. J. Softw. Tools Technol. Transf. (STTT) **15**(2), 89–107 (2013)
19. Garavel, H., Lang, F., Serwe, W.: From LOTOS to LNT. In: Katoen, J.-P., Langerak, R., Rensink, A. (eds.) ModelEd, TestEd, TrustEd. LNCS, vol. 10500, pp. 3–26. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68270-9_1
20. Garavel, H., Serwe, W.: The unheralded value of the multiway rendezvous: illustration with the production cell benchmark. In: Hermanns, H., Höfner, P. (eds.) 2nd Workshop on Models for Formal Analysis of Real Systems (MARS). Electronic Proceedings in Theoretical Computer Science, vol. 244, pp. 230–270, April 2017
21. Geldenhuys, J., Valmari, A.: More efficient on-the-fly LTL verification with Tarjan's algorithm. Theor. Comput. Sci. **345**(1), 60–82 (2005)
22. Hamez, A.: A symbolic model checker for Petri nets: `pnmc`. In: Koutny, M., Desel, J., Kleijn, J. (eds.) Transactions on Petri Nets and Other Models of Concurrency XI. LNCS, vol. 9930, pp. 297–306. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53401-4_15
23. Heiner, M., Schwarick, M., Tovchigrechko, A.: DSSZ-MC – a tool for symbolic analysis of extended Petri nets. In: Franceschinis, G., Wolf, K. (eds.) PETRI NETS 2009. LNCS, vol. 5606, pp. 323–332. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02424-5_20
24. Hillah, L.M., Kindler, E., Kordon, F., Petrucci, L., Trèves, N.: A primer on the Petri net markup language and ISO/IEC 15909–2. Petri Net Newslett. **76**, 9–28 (2009)
25. Hoare, C.A.R.: Communicating sequential processes. Commun. ACM **21**(8), 666–677 (1978)
26. ISO/IEC: High-level Petri Nets - Part 2: Transfer Format. International Standard 15909-2:2011, International Organization for Standardization - Information Technology - Systems and Software Engineering, Geneva (2011)
27. Jensen, J.F., Nielsen, T., Oestergaard, L.K., Srba, J.: TAPAAL and reachability analysis of P/T nets. In: Koutny, M., Desel, J., Kleijn, J. (eds.) Transactions on Petri Nets and Other Models of Concurrency XI. LNCS, vol. 9930, pp. 307–318. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53401-4_16
28. Jensen, P., Larsen, K., Srba, J.: PTrie: data structure for compressing and storing sets via prefix sharing. In: Hung, D., Kapur, D. (eds.) ICTAC 2017. LNCS, vol. 10580, pp. 248–265. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67729-3_15
29. Kant, G., Laarman, A., Meijer, J., van de Pol, J., Blom, S., van Dijk, T.: LTSmin: high-performance language-independent model checking. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 692–707. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_61
30. Kordon, F., Hulin-Hubard, F.: BenchKit, a tool for massive concurrent benchmarking. In: 14th International Conference on Application of Concurrency to System Design (ACSD 2014), Tunis, Tunisia, pp. 159–165. IEEE Computer Society, June 2014
31. Kordon, F., et al.: Report on the model checking contest at Petri nets 2011. In: Jensen, K., van der Aalst, W.M., Ajmone Marsan, M., Franceschinis, G., Kleijn, J., Kristensen, L.M. (eds.) Transactions on Petri Nets and Other Models of Concurrency VI. LNCS, vol. 7400, pp. 169–196. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-35179-2_8

32. Kordon, F., Garavel, H., Hillah, L.M., Paviot-Adet, E., Jezequel, L., Rodríguez, C., Hulin-Hubard, F.: MCC'2015 – the fifth model checking contest. In: Koutny, M., Desel, J., Kleijn, J. (eds.) Transactions on Petri Nets and Other Models of Concurrency XI. LNCS, vol. 9930, pp. 262–273. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53401-4_12
33. Lamport, L.: The part-time parliament. ACM Trans. Comput. Syst. **16**(2), 133–169 (1998)
34. Meijer, J., Kant, G., Blom, S., van de Pol, J.: Read, write and copy dependencies for symbolic model checking. In: Yahav, E. (ed.) HVC 2014. LNCS, vol. 8855, pp. 204–219. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-13338-6_16
35. Meijer, J., van de Pol, J.: Bandwidth and wavefront reduction for static variable ordering in symbolic reachability analysis. In: Rayadurgam, S., Tkachuk, O. (eds.) NFM 2016. LNCS, vol. 9690, pp. 255–271. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40648-0_20
36. Oanea, O., Wimmel, H., Wolf, K.: New algorithms for deciding the Siphon-Trap property. In: Lilius, J., Penczek, W. (eds.) PETRI NETS 2010. LNCS, vol. 6128, pp. 267–286. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-13675-7_16
37. Ongaro, D., Ousterhout, J.: In search of an understandable consensus algorithm. In: USENIX Annual Technical Conference (USENIX ATC), pp. 305–319. USENIX Association (2014)
38. Schmidt, K.: How to calculate symmetries of Petri nets. Acta Informaticae **36**(7), 545–590 (2000)
39. Sorensen, T., Evrard, H., Donaldson, A.F.: Cooperative kernels: GPU multitasking for blocking algorithms. In: Bodden, E., Schäfer, W., van Deursen, A., Zisman, A. (eds.) 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE, pp. 431–441. ACM (2017)
40. Thierry-Mieg, Y.: Symbolic model-checking using ITS-tools. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 231–237. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_20
41. Wimmel, H., Wolf, K.: Applying CEGAR to the Petri net state equation. Log. Methods Comput. Sci. **8**(3) (2012)

Author Index