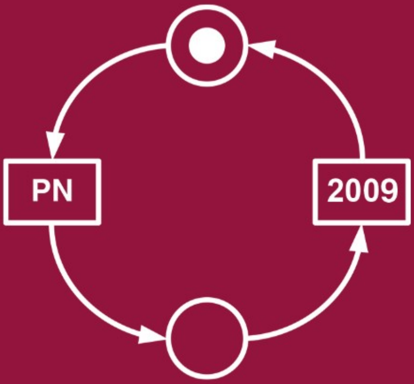
Giuliana Franceschinis
Karsten Wolf (Eds.)

LNCS 5606

# Applications and Theory of Petri Nets

30th International Conference, PETRI NETS 2009
Paris, France, June 2009
Proceedings



Springer

Giuliana Franceschinis    Karsten Wolf (Eds.)

# Applications and Theory of Petri Nets

30th International Conference, PETRI NETS 2009
Paris, France, June 22-26, 2009
Proceedings

Springer

Volume Editors

Giuliana Franceschinis
Università del Piemonte Orientale, Dipartimento di Informatica
viale Teresa Michel, 11, 15100 Alessandria, Italy
E-mail: giuliana@mfn.unipmn.it

Karsten Wolf
Universität Rostock, Institut für Informatik
Schwaansche Str. 2, 18051 Rostock, Germany
E-mail: karsten.wolf@uni-rostock.de

Library of Congress Control Number: Applied for

CR Subject Classification (1998): F.1-3, G.2.2, D.2, C.4, I.6, D.3

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

| | |
|---|---|
| ISSN | 0302-9743 |
| ISBN-10 | 3-642-02423-8 Springer Berlin Heidelberg New York |
| ISBN-13 | 978-3-642-02423-8 Springer Berlin Heidelberg New York |

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper    SPIN: 12700189    06/3180    5 4 3 2 1 0

# Preface

This volume consists of the proceedings of the 30th International Conference on Applications and Theory of Petri Nets and Other Models of Concurrency (PETRI NETS 2009). The Petri Net conferences serve as annual meeting places to discuss the progress in the field of Petri nets and related models of concurrency. They provide a forum for researchers to present and discuss both applications and theoretical developments in this area. Novel tools and substantial enhancements to existing tools can also be presented. The satellite program of the conference comprised four workshops and seven tutorials. This year, the conference was co-located with the 20th IEEE/IFIP International Symposium on Rapid System Prototyping (RSP 2009). The two conferences shared five invited speakers. Detailed information about PETRI NETS 2009 can be found at `http://petrinets2009.lip6.fr/`.

The PETRI NETS 2009 conference was organized by Université Pierre & Marie Curie as a part of MeFoSyLoMa[1], gathering research teams from numerous universities in Île-de-France: CNAM, ENS de Cachan, Université Evry-Val-d'Essone, Université Paris-Dauphine, Université Paris 12, Université Paris 13, and Telecom Paris-Tech. It took place in Paris, France, during June 22-26, 2009. We would like to express our deep thanks to the Organizing Committee, chaired by Fabrice Kordon, for the time and effort invested in the conference and for all the help with local organization. We are also grateful for the financial support by the Centre National de la Recherche Scientifique as well as by partners in the Île-de-France region, in particular: Université Pierre & Marie Curie, Université Paris 13, Laboratoire d'Informatique de Paris 6 and Laboratoire d'Informatique de Paris Nord. We also thank the City of Paris for hosting a reception in the Town Hall on June 24.

This year, we received 46 submissions by authors from 20 different countries. We thank all the authors who submitted papers. Each paper was reviewed by at least four referees. The Program Committee meeting took place in Turin, Italy, and was attended by 18 Program Committee members. At the meeting, 19 papers were selected, classified as: theory papers (13 accepted), application papers (1 accepted), and tool papers (5 accepted). After the conference, some authors were invited to publish an extended version of their contribution in the journal *Fundamenta Informaticae*. We wish to thank the Program Committee members and other reviewers for their careful and timely evaluation of the submissions before the meeting. Special thanks are due to Frank Holzwarth (Springer) and Martin Karusseit (University of Dortmund) for their friendly attitude and technical support with the Online Conference Service. Finally, we wish to express

[1] MeFoSyLoMa stands for "Méthodes Formelles pour les Systèmes Logiciels et Matériel" (Formal Methods for Software and Hardware Systems).

our gratitude to the invited speakers, Bernard Courtois, Gabriel Juhas, Grzegorz Rozenberg, Joseph Sifakis, and Bill Tonti for their contribution. As usual, the Springer LNCS team provided high-quality support in the preparation of this volume.

April 2009

Giuliana Franceschinis
Karsten Wolf

# Organization

## Steering Committee

## Program Committee

## Organizing Committee

|  |  |  |
|---|---|---|
| Nikola Trcka | Jan Martijn van der Werf | Bozena Wozna |
| Fernando Tricas | Matthias Wester-Ebbinghaus | Shingo Yamaguchi |
| Emilio Tuosto | Michael Westergaard | Tatsushi Yamasaki |
| Daniele Varacca | Elke Wilkeit | Shaofa Yang |
| Eric Verbeek | Józef Winkowski | Samir Youcef |
| Marc Voorhoeve |  | Tadanao Zanma |
| Pascal Weil |  |  |

# Table of Contents

# Component-Based Construction of Heterogeneous Real-Time Systems in Bip

Joseph Sifakis

Verimag Laboratory, Centre Equation, 2 ave de Vignate, 38610 GIERES, France

**Abstract.** We present a framework for the component-based construction of real-time systems. The framework is based on the BIP (Behaviour, Interaction, Priority) semantic model, characterized by a layered representation of components. Compound components are obtained as the composition of atomic components specified by their behaviour and interface, by using connectors and dynamic priorities. Connectors describe structured interactions between atomic components, in terms of two basic protocols: rendezvous and broadcast. Dynamic priorities are used to select amongst possible interactions - in particular, to express scheduling policies.

The BIP framework has been implemented in a language and a toolset. The BIP language offers primitives and constructs for modelling and composing atomic components described as state machines, extended with data and functions in C. The BIP toolset includes an editor and a compiler for generating from BIP programs, C++ code executable on a dedicated platform. It also allows simulation and verification of BIP programs by using model checking techniques.

BIP supports a model-based design methodology involving three steps:

1. The construction of a system model from a set of atomic components composed by progressively adding interactions and priorities.
2. The application of incremental verification techniques. These techniques use the fact that the designed system model can be obtained by successive application of property-preserving transformations in a three-dimensional space: Behavior $\times$ Interaction $\times$ Priority.
3. The generation of correct-by-construction distributed implementations from the designed model. This is achieved by source-to-source transformations which preserve global state semantics.

We provide two examples illustrating the methodology.

Further information is available at:

`http://www-verimag.imag.fr/~async/bip.php`

# Unifying Petri Net Semantics with Token Flows

Gabriel Juhás[1], Robert Lorenz[2], and Jörg Desel[3]

[1] Faculty of Electrical Engineering and Information Technology
Slovak University of Technology, Bratislava, Slovakia
gabriel.juhas@stuba.sk
[2] Department of Computer Science, University of Augsburg, Germany
robert.lorenz@informatik.uni-augsburg.de
[3] Department of Applied Computer Science
Catholic University of Eichstätt-Ingolstadt, Germany
joerg.desel@ku-eichstaett.de

**Abstract.** In this paper we advocate a unifying technique for description of Petri net semantics. Semantics, i.e. a possible behaviour, is basically a set of node-labelled and arc-labelled directed acyclic graphs, called token flows, where the graphs are distinguished up to isomorphism. The nodes of a token flow represent occurrences of transitions of the underlying net, so they are labelled by transitions. Arcs are labelled by multisets of places. Namelly, an arc between an occurrence $x$ of a transition $a$ and an occurrence $y$ of a transition $b$ is labelled by a multiset of places, saying how many tokens produced by the occurrence $x$ of the transition $a$ is consumed by the occurrence $y$ of the transition $b$. The variants of Petri net behaviour are given by different interpretation of arcs and different structure of token flows, resulting in different sets of labelled directed acyclic graphs accepted by the net. We show that the most prominent semantics of Petri nets, namely processes of Goltz and Reisig, partial languages of Petri nets introduced by Grabowski, rewriting terms of Meseguer and Montanari, step sequences as well as classical occurrence (firing) sequences correspond to different subsets of token flows. Finally, we discuss several results achieved using token flows during the last four years, including polynomial test for the acceptance of a partial word by a Petri net, synthesis of Petri nets from partial languages and token flow unfolding.

## 1 Introduction

Let us begin with a short story: An alien from a foreign planet comes with his UFO and observes the following situation. He sees a man pressing a button of his mobile phone, then the man is starting to call and after that sitting down. Actually, the man is totally surprised because he had never seen a UFO and an alien, so he immediately calls his wife and sits down to realize what he just saw. The alien writes a report to his planet: ”On Earth, if you want to sit down, you first have to press a button on a mobile phone and then to start to talk to the phone.” As we all know, this is not exactly true, because you can sit down independently (concurrently) from pressing a button and starting to call. But it is still partially true, because in order to call you have to press a button of your phone (or to do something equivalent).

In the case of a sequential machine with a single processor, it is not necessary to deal with the problem of independency of events or actions. One can at most observe non-determinism, e.g. observe sitting down and calling in any order. But concurrency substantially differs from non-determinism. Concurrency is not only the possibility to occur in any order, but to be independent of each other. A typical example making this difference clear, is the occurrence of two events (e.g. two persons wanting to call) sharing one resource (only one phone available). These two events can occur in any order but they obviously cannot happen in parallel. One could say that concurrency includes possible occurrence in any order and simultaneous occurrence. The introducing example also shows that concurrency is not only simultaneous occurrence, which is transitive: a man can sit down concurrently to pressing a button and concurrently to starting to call. However, pressing a button and starting to call are not concurrent, but causally dependent. Observe that the action *calling* in the short story causally depends on the action *pressing a button*, but we can make more calls and pressing buttons many times. If we say that *calling* is an action and *pressing a button* is another action, we speak about causal dependencies between *occurences* of actions rather than about causal dependencies between actions alone.

The study of concurrency as a phenomenon of systems behavior became much attention in recent years, because of an increasing number of distributed systems, multi-processors systems and communication networks, which are concurrent in their nature. There are many ways in the literature to describe non-sequential behaviour, most of them are based on directed acyclic graphs (DAGs). Usually, nodes of DAGs represent the occurrences of actions, i.e. they are labelled by the set of actions and the labelled DAGs (LDAGs) are distinguished up to isomorphism. Such LDAGs are reffered in the literature as abstract [16]. Very often LDAGs with the transitive arcs are used, i.e. the partial orders. Such structures are referred as partially ordered multisets, shortly pomsets, and can formally be seen as isomorphism classes of labelled partial orders [18]. Pomsets are also called partial words [9], emphasizing their close relation to words or sequences; the total order of elements in a sequence is replaced by a partial order.

Petri nets are one of the most prominent formalisms for both understanding the concurrency phenomenon on theoretical and conceptual level and for modelling of real concurrent systems in many application areas. There are many reasons for that, among others the combination of graphical notation and sound mathematical description, see e.g. [5] for a more detailed discussion.

A place/transition Petri net (shorty a Petri net), consists of a set of transitions (actions), which can occur (fire) and a set of places (buffers of tokens), which can carry a number of tokens. Distribution of tokens in places constitutes a state of a Petri net, called a marking. Formally, a marking is given by a multiset of places, i.e. by a function attaching to each place the number of tokens contained in the place. An occurrence of a transition in a marking removes prescribed fixed numbers of tokens from places, i.e. consume a multiset of places, and adds prescribed fixed numbers of tokens to places, i.e. produce a multiset of places, resulting in a new marking. A transition is enabled to occur in a marking, if there is enouhg number of tokens to consume by an occurrence of the transition. In the paper, occurrences of transitions will be referred as events. We also consider a fixed initial marking and a set of legal final markings.

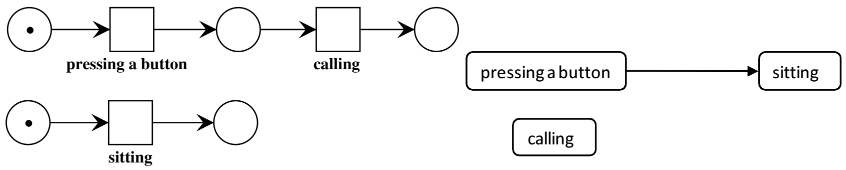


**Fig. 1.** A Petri net modelling the introductory story (left). To describe the independency relation we need at least two steps sequences: the sequence with step "pressing a button and sitting" followed by the step "calling" and the sequence with the step "pressing a button" followed by the step "sitting and calling". The underlying pomset of the behaviour (right).

There are many different ways how to define behaviour of Petri nets. The simplest way is to take occurrence sequences, i.e. sequences of occuring transitions.

Another possibility is to extend the sequences of occuring transitions to sequences of steps of transitions. Steps are multisets of transitions. A step is enabled to occur in a marking if there is enough tokens to consume by the simultaneous occurrences of transitions in the multiset. A situation described in the introductory example, where independence relation of transition occurrences is not transitive, cannot be decribed by a single step sequence, see Figure 1.

Therefore, pomsets seems to be a better choice to formalize non-sequential semantics (see e.g. [18,9]. The natural question arises: which pomsets do express behaviour of a Petri net? The answer has close relationships to the step semantics. In [9,12] it is suggested to take pomsets satisfying: For each co-set of events (i.e. for each set of unordered events) there holds: The step of events in the co-set is enabled to occur in the marking reached from the initial marking by occurrence of all events smaller than an event from the co-set.

Another possibility to express behaviour of Petri nets, is to take processes of [7,8], which are a special kind of acyclic nets, called occurrence nets, together with a labelling which associates the places (called conditions) and transitions (called events) of the occurrence nets to the places and transitions of the original nets preserving the number of consumed and produces tokens, see Figure 2. Processes can be understood as (unbranched) unfoldings of the original nets: every event in the process represents an occurrence of its label in the original net. Abstracting from conditions of process nets, LDAGs on events are defined. These LDAGs express the direct causality between events. Adding transitive arcs to these LDAGs, we get pomsets called runs, which express (not necessarily direct) causality between events. In contrast to enabled pomsets, events ordered by an arc in a run cannot be independent. A special role plays those runs, which are minimal w.r.t. extension: they express the minimal causality between events. An important result relating enabled pomsets and runs was proven in [12,19]: Every enabled pomset includes a run and every run is an enabled pomset. Therefore, minimal enabled pomsets equal minimal runs. In contrast to sequential semantics and step semantics, processes distinguish between the history of tokens. An example is shown in Figure 2. The process nets distinguish a token in place $p_3$ produced by the occurrence of transition $t_1$ from a token in place $p_3$ produced by the occurrence of transition $t_2$. As a consequence, one occurrence sequence, e.g. $t_1t_2t_3$, can be an extension of two

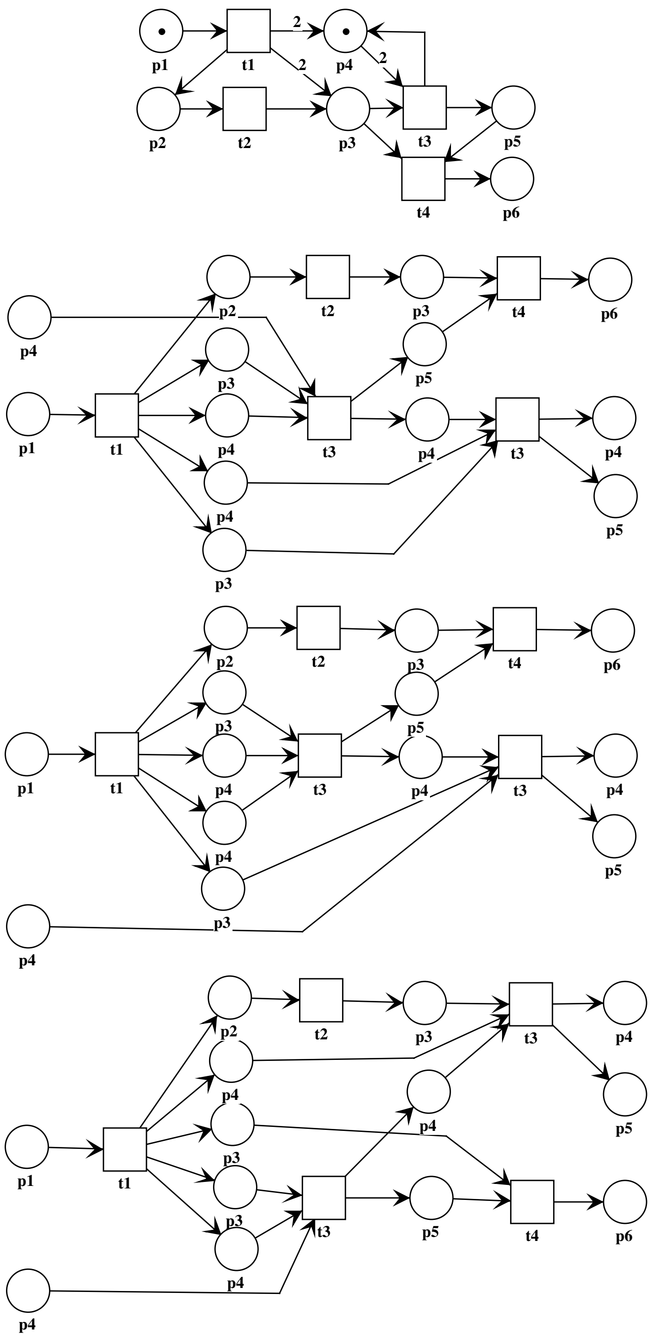


**Fig. 2.** A Petri net (above) with three processes (below)

different processes. The process semantics defined in [7] is also called individual token semantics. Notice that in the case of the process semantics of elementary nets (with at most one token in a place), any occurrence sequence and any step sequence uniquely determine a process. In [3] the collective token semantics, which does not distinguish between the history of tokens, is introduced. It is defined using an equivalence relation between processes. The equivalence relates processes differing only in permuting (swapping) unordered conditions representing tokens in the same place of the original net. For example, the processes in Figure 2 are equivalent w.r.t. swapping equivalence. For swapping equivalence classes, called commutative processes, there holds that any occurrence sequence and any step sequence uniquely determine a commutative process.

In [15] behaviour is described using rewrite terms generated from elementary terms using concurrent and sequential composition. In this algebraic approach any transition $t$ is an elementary rewrite term, allowing to replace the marking $consume(t)$ by the marking $produce(t)$. Any marking consisting of a single place $p$ is an elementary term rewriting $p$ by $p$ itself. Rewrite terms are constructed inductively from elementary terms using operator ; for sequential and operator $||$ for concurrent composition. Each term has associated an initial marking and final marking. Two terms can be composed sequentially only if the final marking of the first term coincides with the initial marking of the second one. For concurrent composition of two terms, the initial marking of the resulting term is obtained by multiset addition of the initial markings of the composed terms, and likewise for the final marking. The behavior of a net is given by equivalence classes of rewrite terms defined by a set of equations. In [4] it is shown that the equivalence class of rewrite terms as defined in [15] corresponds to the swapping equivalence class of processes. Obviously, one can attach pomsets to rewrite terms. Each process term $\alpha$ defines a partially ordered set of events representing transition occurrences in an obvious way: an event $e_1$ is smaller than an event $e_2$ if the rewrite term $\alpha$ contains a sub-term $\alpha_1; \alpha_2$ such that $e_1$ occurs in $\alpha_1$ and $e_2$ occurs in $\alpha_2$. The pomsets of rewrite terms have a special structure. It is proven in [6], that a pomset is generated by concurrent and sequential composition from single element pomsets if and only if it does not contain the shape of so called N-form. As a consequence, we get the characterization of pomsets which are associated to rewrite terms of a Petri net: an enabed pomset is associated with a rewrite term of a net if and only if it is N-free.

Most of the discussed results showing the correspondence between different semantics are quite complicated, despite the clear intuition. The differences in the technique used to describe the behaviour often cause, that even straightforward relationships have technically difficult proofs. The formal description of the intuition chosen for single semantics has some limitations too. Consider for example enabled pomsets and processes. The definition of enabled pomsets is inherently exponential: to test the enabledness using co-sets and steps in innefective. The main advantages of the processes, as claimed in the literature, is that they describe behaviour in the same modelling language, i.e. using occurrence nets. But this is also their main disadvantage, because to model each individual token by a single condition make the formal manipulation with processes difficult. In fact, there is no need to remember each single token (it is suitable for elementary nets, where at most one token for each place can be present in a marking, but not so efficient for place/transition Petri nets), but it is enough to remember how many tokens produced by an occurrence are consumed by another occurrence, i.e. how many tokens flow on the arcs connecting occurrences of transitions, abstracting from the individuality of conditions.

The natural question arise: Is there a possibility to express all semantics discussed above using a unifying but yet simple formal description? We give a positive answer to this question, presenting a framework for description of different variants of semantics of Petri nets. The semantics in the framework is basically a set of node-labelled and arc-labelled directed acyclic graphs, called token flows, where the graphs are distinguished up to isomorphism, defined originally in [10]. The nodes of a token flow represent occurrences of transitions of the underlying net, so they are labelled by transitions. Arcs are labelled by multisets of places. Namelly, an arc between an occurrence $x$ of a transition $a$ and an occurrence $y$ of a transition $b$ is labelled by a multiset of places, saying how many tokens produced by the occurrence $x$ of the transition $a$ is consumed by the occurrence $y$ of the transition $b$. A token flow of a Petri net have to fulfill so called token flow property:

– Ingoing token flow (i.e. the sum of multisets over ingoing arcs) of any occurrence of a transition $a$ equals the multiset of places consumed in the net by firing transition $a$.
– Outgoing token flow (i.e. the sum of multisets over outgoing arcs) of any occurrence of a transition $a$ equals the multiset of places produced in the net by firing transition $a$.

To keep the information which occurrences consume tokens directly from the initial marking and which occurrences produce unconsumed tokens in the final marking, we add a special single entry node and a special single exit node, which are labelled by the initial and a final marking. respectively. The outgoing token flow of the entry equals the initial marking and the ingoing token flow of the exit equals a final marking. An important question arises, when expressing the behaviour by a set of LDAGs, i.e. by an LDAG language: What is the interpretation of arcs in an LDAG? The answer is not unique. Using Petri nets, there are two basic interpretations:

1. An occurrence $y$ of an action $b$ *directly causaly depends* on an occurrence $x$ of an action $a$: By the occurrence $y$ of a transition $b$, transition $b$ consumes some tokens produced by the occurrence $x$ of $a$. This interpretaton is used by processes of Petri nets [7,8]. Direct causal interpretation of arcs is resulting in the requirement that there is a non-zero token flow between the occurrences, i.e. the multiset label of any arc does not equal empty multiset. The transitive closure of direct causality gives causal dependency.
2. An occurrence $y$ of an action $b$ *follows* an occurrence $x$ of an action $a$: The occurrence $y$ of a transition $b$ is either causaly dependent on the occurrence $x$ or the occurrences $y$ and $x$ are independent. This interpretation is used in the occurrence sequences, step sequences and enabled pomsets of Petri nets [9]. We call this interpretation occurrence interpretation. By occurrence interpretation, also arcs with the zero token flow are allowed.

The variants of Petri net behaviour are given by different interpretation of arcs and different structure of token flows, resulting in different sets of labelled directed acyclic graphs accepted by the net. We show that the most prominent semantics of Petri nets correspond to different subsets of token flows. Namely:

– Processes of Goltz and Reisig [7,8] correspond to *direct causal token flows*, i.e. to the token flows with direct causal interpretation of arcs, where all arcs have non-zero token flows.
– Token flows in which LDAGs are pomsets and at least the skeleton arcs (non-transitive arcs) have non-zero token flows are called the *causal token flows* of Petri nets and represent the causal semantics. They are obtained from direct causal token flows by adding all transitive arcs.
– Enabled pomsets, i.e. partial words introduced by Grabowski [9], correspond to *pomset token flows*, i.e. to the token flows, where LDAGs are pomsets with occurrence interpretation of arcs (arcs may have the zero token flow).
– Rewriting terms of Meseguer and Montanari [15] correspond to *N-free token flows*, i.e. to the pomset token flows where the underlying pomsets are N-free.

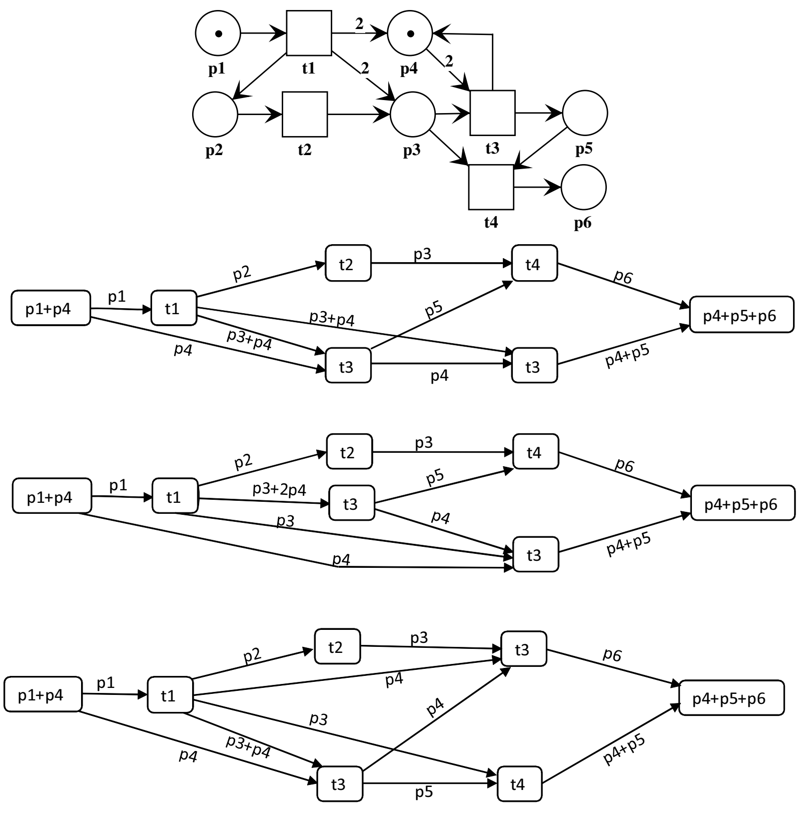


**Fig. 3.** Petri net from the Figurer̃efprocex (above) and the dicausal token flows corresponding to the processes in Figure 2

- Step sequences correspond to step ordered multiset token flows, shortly *somset token flows*, i.e. to the pomset token flows, where the relation given by unordered pairs of nodes is transitive.
- Occurrence sequences correspond to totally ordered token flows, shortly *tomset token flows*, i.e. to the pomset token flows where the relation given by unordered pairs of nodes is empty.

For better illustration, dicausal token flows corresponding to processes from Figure 2 are included in Figure 3.

An important role plays the equivalence given by symmetric and transitive closure of the extension relation of LDAGs, called *extension equivalence*: equivalence classes given by exchange equivalence on sequences, swapping equivalence on processes, and equivalence on rewrite terms correspond to restrictions of exchange equivalence classes to tomset token flows, direct causal token flows and N-free token flows, respectively.

Finally, we discuss the results achieved using token flows during the last four years, including the first polynomial test for the acceptance of a partial word by a Petri net, synthesis of Petri nets from partial languages and token flow unfolding.

## 2 Token Flows

We use $\mathbb{N}$ to denote the nonnegative integers. Given a finite set $A$, $|A|$ denotes the cardinality of $A$. Given a function $f$ from $A$ to $B$ and $C \subseteq A$ we write $f|_C$ to denote the restriction of $f$ to $C$. We write $A \setminus B$ to denote the usual set difference of sets $A$ and $B$. The set of all subsets of a set $A$ is denoted by $2^A$. The set of all multisets over a set $A$, i.e. the set of functions from $A$ to $\mathbb{N}$ is denoted by $\mathbb{N}^A$. We use $\emptyset$ to denote the empty multiset, i.e. $\forall a \in A : \emptyset(a) = 0$. The sum of multisets, the comparison of multisets and the difference of multisets are given as usual: given $m, m' \in \mathbb{N}^A$, $(m + m')(a) = m(a) + m'(a)$ for each $a \in A$, $m \geq m' \Leftrightarrow m(a) \geq m'(a)$ for each $a \in A$, and whenever $m \geq m'$ then $(m - m')(a) = m(a) - m'(a)$ for each $a \in A$.

We write $\sum_{a \in A} m(a)a$ to denote the multiset $m$ over $A$. Given a function $l$ from a set $V$ to a set $X$, and a subset $S \subseteq V$, we define the multiset $\sum_{s \in S} l(s) \subseteq \mathbb{N}^X$ by $(\sum_{s \in S} l(s))(x) = |\{v \in V \mid v \in S \wedge l(v) = x\}|$. Given a binary relation $R \subseteq A \times A$ over a set $A$, $R^+$ denotes the transitive closure of $R$ and $R^*$ the reflexive and transitive closure of $R$.

A *directed graph* is a pair $(V, \rightarrow)$, where $V$ is a finite *set of nodes* and $\rightarrow \subseteq V \times V$ is a binary relation over V called the *set of arcs*. Given a binary relation $\rightarrow$ we write $v \rightarrow v'$ to denote $(v, v') \in \rightarrow$ and $v \not\rightarrow v'$ to denote $(v, v') \notin \rightarrow$. We define $^\bullet v = \{v' \mid v' \rightarrow v\}$ and $v^\bullet \{v' \mid v \rightarrow v'\}$. A node $v \in V$ is called entry (also initial node, source, start or input), if $^\bullet v = \emptyset$. A node $v \in V$ is called exit (also final node, sink, end or output), if $v^\bullet = \emptyset$. Skeleton of a directed graph $(V, \rightarrow)$ is the directed graph $(V, \rightarrow')$ with $\rightarrow' = \{(v, v') \mid \nexists v'' : v \rightarrow^+ v'' \rightarrow^+ v'\}$ containing no transitiove arcs of $\rightarrow$.

A *partial order* is a directed graph $po = (V, <)$, where $<$ is irreflexive and transitive on $V$. Given a partial order $(V, <)$, for a set $S \subseteq V$ and a node $v \in V \wedge v \notin S$ we write $v < S$, if $v < s$ for a node $s \in S$. Two nodes $v, v'$ of a partial order $(V, <)$ are called *independent* if $v \not< v'$ and $v' \not< v$. By $co \subseteq V \times V$ we denote the set of all pairs of independent nodes of $V$. A *co-set* in a partial order $(V, <)$ is a subset $S \subseteq V$

fulfilling: $\forall x, y \in S : \; x\, co\, y$. A *cut* (also called a slice) is a maximal co-set. A partial order $to = (V, <)$ satisfying $\forall (v, v') \in V \times V : v \neq v' \Rightarrow (v < v' \vee v' < v)$ is a *total order*. A partial order $so = (V, <)$, where $co$ is transitive, is a *step order*. Given a step order $so = (V, <)$, we write $S < S'$ for cuts $S, S' \in V$ whenever there exist $v \in S$ such that $v < S'$. Because relation $co$ is symmetric and reflexive, in a step order it is an equivalence relation. A partial order $po = (V, <)$ satisfying $(v < v' \wedge w < w' \wedge v < w') \Rightarrow \neg(v\, co\, w \wedge v'\, co\, w \wedge v'\, co\, w')$ is an *N-free partial order*.

A *directed acyclic graph* is a directed graph $dag = (V, \rightarrow)$, where the $po_{dag} = (V, \rightarrow^+)$ is a partial order. Given directed acyclic graphs $(V, \rightarrow)$ and $(V, \rightarrow')$ we say that $(V, \rightarrow)$ is an extension of $(V, \rightarrow')$ iff $\rightarrow' \subseteq \rightarrow$.

A *labelled directed acyclic graph*, shortly an LDAG, is a triple $ldag = (V, \rightarrow, l)$, where $(V, \rightarrow)$ is a directed acyclic graph and $l$ is a labelling function from $V$ to a set of labels. Two LDAGs $(V_1, \rightarrow_1, l_1)$, $(V_2, <_2, l_2)$ are isomorphic iff there exists a bijection $\gamma$ from $V_1$ to $V_2$ between nodes which preserve the arcs and the labelling function, i.e. $\forall v_1, v_2 \in V_1 : v_1 \rightarrow_1 v_2 \iff \gamma(v_1) \rightarrow_2 \gamma(v_2) \wedge l_1(v_1) = l_2(\gamma(v_1))$. We are not interested in the identity of nodes of an LDAG, so we distinguish LDAGs up to isomorphism. Without lose of generality, we will use any LDAG from an isomorphism class of LDAGs to denote the whole class.

A *pomset* is an LDAG $lpo = (V, <, l)$, where $(V, <)$ is a partial order. A *tomset* is an LDAG $lto = (V, <, l)$, where $(V, <)$ is a total order. A *somset* is an LDAG $lso = (V, <, l)$, where $(V, <)$ is a step order. An *N-free pomset* is an LDAG $(V, <, l)$, where $(V, <)$ is an N-free partial order.

An LDAG language is a set of (isomorphism classes of) LDAGs. Given an LDAG language $L$, and a subset $L' \subseteq L$, $L'$ is called a sublanguage of $L$, and by $L_{min}$ we denote its minimal sublanguage $L_{min} = \{(V, \rightarrow, l) \in L \mid \nexists (V, \rightarrow', l) \in L : \rightarrow' \subset \rightarrow\}$.

We use special LDAGs with a single entry and a single exit. An $ldag = (V, \rightarrow, l)$ with a single entry and a single exit is called single-entry and single-exit LDAG, shortly a SESE LDAG, $entry(ldag)$ denotes its entry and $exit(ldag)$ denotes its exit.

We also remove nodes and delete arcs from an LDAG. Let $ldag = (V, \rightarrow, l)$ be an LDAG. Let $X \subseteq V$. We define $remove(ldag, X) = (V', \rightarrow', l')$, where $V' = V \setminus X$, $\rightarrow' = \rightarrow \cap (V' \times V')$ and $\forall v \in V' : l'(v) = l(v)$. Observe, that removing nodes from an LDAG one gets an LDAG. Moreover, removing nodes from a pomset one gets a pomset. Let $\rightharpoonup \subseteq \rightarrow$. We define $delete(ldag, \rightharpoonup) = (V, \rightarrowtail, l)$, where $\rightarrowtail = \{v \rightarrow v' \mid v \not\rightharpoonup v'\}$.

Now we are prepared to define token flow functions of LDAGs over a finite set, and ingoing and outgoing token flows.

**Definition 1 (Token flow function).** *Let $ldag = (V, \rightarrow, l)$ be an LDAG. Let $P$ be a finite set. A function $flow$ from the set of arcs $\rightarrow$ to $\mathbb{N}^P$ is called a* token flow function *of $ldag$ over $P$. The function $flow$ defines two functions attaching multisets over $P$ to nodes:*

- *the function $in_{flow}$ from $V$ to $\mathbb{N}^P$, given by $in_{flow}(v) = \sum_{v' \in {}^\bullet v} flow(v', v)$, called the* ingoing token flow *of $v$,*
- *the function $out_{flow}$ from $V$ to $\mathbb{N}^P$, given by $out_{flow}(v) = \sum_{v' \in v^\bullet} flow(v, v')$, called the* outgoing token flow *of $v$.*

Let us define the first central notion of the paper - a token flow over a finite set: it is an LDAG with a single entry, a single exit, and with arcs labelled by multisets over a finite set.

**Definition 2 (Token flow).** *Let $P$ be a finite set. A* token flow *over $P$ is a pair $flowdag = (ldag, flow)$, where $ldag = (V, \rightarrow, l)$ is a SESE LDAG, and $flow$ is a token flow function of $ldag$ over $P$. A set of token flows over $P$ is called $P$-token flow language, or shortly* token flow language.

## 3 Token Flows of Petri Nets

**Definition 3 (Petri Net).** *A place/transition Petri net (shortly a* Petri net*) is a 6-tuple $PN = (P, T, consume, produce, initial, final)$ where $P$ is a finite set of places, $T$ is a finite set of transitions, $T \cap (P \cup \mathbb{N}^P) = \emptyset$, consume and produce are functions from $T$ to $\mathbb{N}^P$, such that $\forall t \in T : consume(t) \neq \emptyset \wedge produce(t) \neq \emptyset$, multiset $initial \in \mathbb{N}^P$ is an initial marking and $final \subseteq N^P$ is a set of legal final markings.*

In the rest of the paper we suppose that a $PN = (P, T, consume, produce, initial, final)$ is given. A multiset $m \in \mathbb{N}^P$ is called a marking of $PN$. Sequential behaviour of the $PN$ is given by occurrences (firings) of transitions: A transition $t \in T$ is enabled to occur in a marking $m$ of $PN$ iff $m \geq consume(t)$. An occurrence of enabled transition $t$ in a marking $m$ leads to the follower marking $m' = m - comsume(t) + produce(t)$. We write $m \overset{t}{\longmapsto} m'$ to denote that $t$ is enabled to occur in $m$ and that its occurrence leads to $m'$. Sequential behaviour can easily be extended to the simplest way to dercsribe concurrent occurrences of transitions - to the occurrences of steps, which are multisets of transitions: Given a step $s \in \mathbb{N}^T$, denote by $consume(s)$ the multiset of places given by $\forall p \in P : consume(s)(p) = \sum_{t \in T} s(t)consume(t)(p)$. By $produce(s)$ denote the multiset of places given by $\forall p \in P : produce(s)(p) = \sum_{t \in T} s(t)produce(t)(p)$. A step $s \in \mathbb{N}^T$ is enabled to occur in a marking $m$ of $PN$ iff $m \geq consume(s)$. An occurrence of enabled step $s$ in a marking $m$ leads to the follower marking $m' = m - comsume(s) + produce(s)$. We write $m \overset{s}{\longmapsto} m'$ to denote that $s$ is enabled to occur in $m$ and that its occurrence leads to $m'$.

Le us notice, that the above definition differs from the usual definition of place Petri nets, however the difference is only technnical. Usually, a place/transition Petri net is given as a bipartite directed graph with weighted arcs, with nodes formed by places and transition, places and arcs labelled by nonnegative integers. The labelling of places gives the marking. From technical reasons in our definition we additionaly require that no transition equals a multiset of places. In a usual definition, instead of the functions $consume, produce$ the relationship between places and transitions is given using a set of arcs $F \subseteq ((P \times T) \cup (T \times P))$ (also called flow relation) and a weight function $W$ from $F$ to $\mathbb{N}$. Using our definition, $F$ and $W$ can be easily reconstructed: $F = \{(p,t) \in P \times T \mid consume(t)(p) \neq 0\} \cup \{(t,p) \in T \times P \mid produce(t)(p) \neq 0\}$, $\forall (p,t) \in F : W(p,t) = consume(t)(p), \forall (t,p) \in F : W(t,p) = produce(t)(p)$.

In a usual definition, the set of final marking is not defined. The intended meaning of the set of final markings is to allow acceptance of only a subset of LDAGs generated by processes. Obviously, taking the set of all multisets as legal final marking, one gets that all of the processes are accepted.

Now we are prepared to define the second central notion of the paper: token flows of $PN$, as token flows over $P$ such that the entry is labelled by the initial marking and the outgoing token flow of the entry equals the initial marking, the exit is labelled by a final marking and the outgoing token flow of the exit equals the final marking, and for all other nodes the ingoing token flow equals the $consume$ value of their label and outgoing token flow equals the $produce$ value of their label.

**Definition 4 (Token Flow of a Petri Net).** *Let* $PN = (P, T, consume, produce, initial, final)$ *be a Petri net and let* $flowdag = (ldag, flow)$ *be a token flow over* $P$ *with* $ldag = (V, \rightarrow, l)$. *Then* $flowdag$ *is called* token flow of $PN$*, and we say that* $flow$ *fulfils the token flow property iff:*

1. $\forall v \in V : v \notin \{entry(ldag), exit(ldag)\} \Rightarrow (l(v) \in T \wedge consume(l(v)) = in_{flow}(v) \wedge produce(l(v)) = out_{flow}(v))$
2. $l(entry(ldag)) = out_{flow}(entry(ldag)) = initial,$
3. $l(exit(ldag)) = in_{flow}(exit(ldag)) \subseteq final,$

*The set of all token flows of* $PN$ *is denoted by* $L^{tf}_{all}(PN)$ *and called the* token flow language of $PN$.

Observe that given a SESE LDAG $ldag = (V, \rightarrow, l)$ with the entry labelled by the initial marking, the exit labelled by any final marking and remaining nodes labelled by transitions of $PN$, the token flow functions fulfilling the token flow property are simply nonnegative integer solutions of the system (1 - 3) of linear equations from the previous definition, with $in_{flow}(v)$ replaced by $\sum_{v' \in {}^\bullet v} flow(v', v)$, $out_{flow}(v)$ replaced by $\sum_{v' \in v^\bullet} flow(v, v')$ for any $v \in V$, and unknown variables $flow(v, v')(p)$ for each $p \in P$ and each $v \rightarrow v'$. We will call such a system the *token flow system of the* $ldag$.

Using the interpretation of arcs in the token flows and the structure of the LDAGs, one can define all prominent semantics of $PN$. Basiacally, a semantics denoted by $sem$ is determinded by a subset $L^{tf}_{sem}(PN) \subseteq L^{all}tf(PN)$ of token flows of $PN$. Each semantics can be recognignized on four levels. The first level is given by a language obtained from token flows by forgetting the flow function. The second level is formed by the minimal sublanguage of this language. The third level is given by a language obained from the first level by forgetting the entry and exit. The fourth level is given by the minimal sublangage of the third language.

**Definition 5 (Languages of a Petri Net).** *Let* $L^{tf}_{sem}(PN) \subseteq L^{tf}_{all}(PN)$. *We derive following four LDAG languages from* $L^{tf}_{sem}(PN)$*:*

1. $L^{io}_{sem}(PN) = \{ldag \mid (ldag, flow) \in L^{tf}_{sem}(PN)\}$,
2. $L^{io}_{sem}(PN)_{min}$,
3. $L^{nio}_{sem}(PN) = \{remove(ldag, \{entry(ldag), exit(ldag)\}) \mid ldag \in L^{io}(PN)\}$,
4. $L^{nio}_{sem}(PN)_{min}$.

Given an $ldag \in L^{io}_{all}(PN)$, its final marking, i.e. the label of the final node, can be easily determined.

**Proposition 1.** *Let* $ldag \in L^{io}_{all}(PN)$ *with* $ldag = (V, \to, l)$*. Then* $l(exit(ldag)) = initial + \sum_{v \in V \setminus \{entry(ldag), exit(ldag)\}} (produce(l(v)) - consume(l(v)))$.

If $L^{io}_{sem}(PN)$ is a pomset language, then for each LDAG $ldag' \in L^{nio}_{sem}(PN)$, there exists one and only one SESE LDAGs $ldag \in L^{io}_{sem}(PN)$, such that $ldag' = remove\ (ldag, \{entry(ldag), exit(ldag)\}$. As a consequence for pomset languages we get that $L^{nio}_{sem}(PN)_{min}$ is the image of the restriction of $remove(ldag, \{entry(ldag),\ exit\ (ldag)\})$ to $L^{io}_{sem}(PN)_{min}$.

## 3.1 Token Flow Semantics of Petri Nets

As the first semantics we define the direct causal token flows. The arcs in a direct causal token flow represent direct causality, i.e. the fact, that the source of the arc produced at least one token consumed by the target of the arc.

**Definition 6 (Direct Causal Token Flow).** *Let* $flowdag = (ldag, flow)$ *with* $ldag = (V, \to, l)$ *be a token flow of* $PN$ *satisfying* $\forall (v, v') \in \to$: $flow(v, v') \neq \emptyset$*. Then* $flowdag$ *is called direct causal token flow of* $PN$*, shortly* dicausal token flow of $PN$*. The set of all dicausal token flows of* $PN$ *is denoted by* $L^{tf}_{dicausal}(PN)$ *and called* dicausal token flow language of $PN$*.*

Observe, that the inequations $flow(v, v') \neq \emptyset$ can be rewrited to the integer inequation $\sum_{p \in P} flow(v, v')(p) \neq 0$. Adding the inequations to the token flow system of $ldag$ we get the system of linear inequetions, called *dicausal token flow system of* $ldag$. Obviously, $ldag \in L^{io}_{dicausal}(PN)$ iff the dicausal token flow system of $ldag$ has a nonnegative integer solution.

Dicausal token flows of $PN$ contain complete information about causal dependency of transition occurrences, including the information which occurrences consume tokens from the initial marking and which occurrences produce tokens in the final marking. The difference between elements of $L^{io}_{dicausal}(PN)$ and $L^{nio}_{dicausal}(PN)$ is that in the elements from $L^{nio}_{sem}(PN)$ the information which occurrences consume tokens from the initial marking and which occurrences produce tokens in the final marking is forgotten.

The second semantics are the causal token flows. The arcs in a causal pomset represent causality between the source and the target, not necessarily the direct one.

**Definition 7 (Causal Token Flow).** *Let* $flowdag = (ldag, flow)$ *be a token flow of* $PN$ *such that* $ldag = (V, <, l)$ *is a pomset. Let* $(V, \to)$ *be the skeleton of* $(V, <)$*. If* $\forall (v, v') \in \to$: $flow(v, v') \neq \emptyset$ *then* $flowdag$ *is called* causal token flow of $PN$*. The set of all causal token flows of* $PN$ *is denoted by* $L^{tf}_{causal}(PN)$ *and called* causal token flow language of $PN$*.*

The next semantics are the pomset token flows. In a pomset token flow, the arcs represent the fact, that the source and the target occurred sequentially. This in fact means, that either these occurrences are independent or the target is causaly dependent on the source.

**Definition 8 (Pomset Token Flow).** *Let* $flowdag = (ldag, flow)$ *be a token flow of* $PN$ *such that* $ldag$ *is a pomset. Then* $flowdag$ *is called* pomset token flow of $PN$*. The set of all pomset token flows of* $PN$ *is denoted by* $L^{tf}_{pomset}(PN)$ *and called* pomset token flow language of $PN$*.*

The next semantics are token flows, where the underlying LDAGs are N-free pomsets.

**Definition 9 (N-Free Token Flow).** *Let $flowdag = (ldag, flow)$ be a token flow of $PN$ such that $ldag$ is an N-free pomset. Then $flowdag$ is called* N-free token flow of $PN$*. The set of all N-free token flows of $PN$ is denoted by $L^{tf}_{Nfree}(PN)$ and called* N-free token flow language of $PN$.

As the last two semantics we define somset and tomset token flows.

**Definition 10 (Somset Token Flow).** *Let $flowdag = (ldag, flow)$ be a token flow of $PN$ such that $ldag$ is a somset. Then $flowdag$ is called* somset token flow of $PN$*. The set of all somset token flows of $PN$ is denoted by $L^{tf}_{somset}(PN)$ and called* somset token flow language of $PN$.

**Definition 11 (Tomset Token Flow).** *Let $flowdag = (ldag, flow)$ be a token flow of $PN$ such that $ldag$ is a tomset. Then $flowdag$ is called* tomset token flow of $PN$*. The set of all tomset token flows of $PN$ is denoted by $L^{tf}_{tomset}(PN)$ and called* tomset token flow language of $PN$.

## 3.2 Relationship between Token Flow Semantics of Petri Nets

Directly from the above defintions we can see the following relationships between the presented token flow semantics of $PN$. Let $(ldag, flow)$ be a token flow of $PN$ with $ldag = (V, \to, l)$. We define $positive(ldag, flow) = delete(ldag, \{v \to v' \mid flow(v, v') = \emptyset\})$. Consider that $x \in \{io, nio\}$, i.e. $x$ can be replaced by either $io$ or $nio$, and $sem \in \{causal, pomset, Nfree, somset, tomset\}$:

$$L^{x}_{causal}(PN) = \{(V, \to^{+}, l) \mid (V, \to, l) \in L^{x}_{dicausal}(PN)\}$$
$$L^{io}_{dicausal}(PN) = \{positive(ldag, flow) \mid (ldag, flow) \in L^{tf}_{sem}(PN)\}$$
$$L^{x}_{tomset}(PN) \subseteq L^{x}_{somset}(PN) \subseteq L^{x}_{Nfree}(PN) \subseteq L^{x}_{pomset}(PN)$$
$$L^{x}_{causal}(PN) \subseteq L^{x}_{pomset}(PN)$$

Another important relationship between these semantics is the relationship w.r.t. extension. Taking two sets $X, Y$ of pomsets, we denote by $X \Supset Y$ that for each pomset $(V, <, l)$ from $X$ there exists a pomset $(V, <', l)$ from $Y$ such that $(V, <)$ is an extension of $(V, <')$. We observe the following:

$$L^{x}_{tomset}(PN) \Supset L^{x}_{somset}(PN) \Supset L^{x}_{Nfree}(PN) \Supset L^{x}_{pomset}(PN) \Supset L^{x}_{causal}(PN)$$

As a consequence:

$$L^{x}_{causal}(PN)_{min} = L^{x}_{pomset}(PN)_{min}$$

Observe, that the total order of a tomset token flow can be an extension of DAGs of several dicausal token flows with different DAGs. On the other hand, there can be several tomset token flows with different total orders, which are extensions of the DAG of a dicausal token flow. Therefore we introduce an equivalence on $L^{x}_{all}(PN)$ as the symmetric and transitive closure of the relation "being an extension". The equivalence is called the extension equivalence on $L^{x}_{all}(PN)$.

**Definition 12 (Extension Equivalence).** *Let $x \in \{io, nio\}$. Let $(V, <, l), (V, <', l) \in L^{x}_{all}(PN)$. Define $(V, \to, l) \propto (V, \to', l)$ if $(V, \to)$ is an extension of $(V, \to')$. The symmetric and transitive closure $\equiv$ of $\propto$ is called* extension equivalence *on $L^{x}_{all}(PN)$.*

### 3.3 Direct Causal Token Flows and Processes

In this subsection we discuss the relationship between dicausal token flows and processes of [7,8].

**Definition 13 (Occurrence Net).** *An* occurrence net *is a directed acyclic graph $O = (B \cup E, G)$, with two partitions of nodes denoted by $B$ and $E$ (called conditions and events) s. t. $(B \cup 2^B) \cap E = \emptyset$, and flow relation $G \subseteq (B \times E) \cup (E \times B)$, s. t. $| \bullet b|$, $|b \bullet | \leq 1$ for every $b \in B$ and no node from $E$ is an entry or an exit.*

The set of conditions of an occurrence net $O = (B\cup, G)$ which are entries and exits are denoted by $Min(O)$ $Max(O)$, respectively.

**Definition 14 (Process).** *Let $PN = (P, T, consume, produce, initial, final)$ be a Petri net. A* process *of $PN$ is a pair $K = (O, \rho)$, where $O = (B \cup E, G)$ is an occurrence net and $\rho : B \cup E \to P \cup T$ is a labelling function, satisfying*

*(i)* $\rho(B) \subseteq P$ *and* $\rho(E) \subseteq T$.
*(ii)* $\forall e \in E, \forall p \in P : \; |\{b \in \bullet e \mid \rho(b) = p\}| = consume(\rho(e))(p)$ *and*
$\forall e \in E, \forall p \in P : \; |\{b \in e \bullet \mid \rho(b) = p\}| = produce(\rho(e)), (p)$.
*(iii)* $\forall p \in P : \; |\{b \in Min(O) \mid \rho(b) = p\}| = initial(p)$
*(iv)* $\exists fin \in final$ *such that* $\forall p \in P : \; |\{b \in Max(O) \mid \rho(b) = p\}| = fin(p)$.

**Definition 15 (Canonical LDAG, Canonical Token Flow).** *Let $K = (O, \rho)$ be a process of a Petri net $PN$. Define*

$$entryarc(K) = (Min(O), e) \mid \exists b \in Min(O) : (b, e) \in G,$$
$$exitarc(K) = (e, Max(O)) \mid \exists b \in Max(O) : (e, b) \in G.$$

*The* canonical LDAG of process *$K$ is the LDAG $ldag_K = (V, \to, l)$, where*

$$V = E \cup \{Min(O), Max(O)\},$$
$$\to = G^2|_{E \times E} \cup entryarc(K) \cup exitarc(K),$$
$$l|_E = \rho|_E,\ l(Min(O)) = \textstyle\sum_{b \in Min(O)} \rho(b) \text{ and } l(Max(O)) = \textstyle\sum_{b \in Max(O)} \rho(b).$$

*The* canonical token flow function $flow_K$ of process *$K$ is the token flow function of $ldag_K$ over $P$ given by $flow_K(v, v') = \sum_{b \in (v^\bullet \cup {}^\bullet v')} \rho(b)$ for each $v \to v'$. The* canonical token flow of process *$K$ is the pair $(ldag_K, flow_K)$. The language of canonical token flows of all processes of $PN$ is denoted by $CL(PN)$.*

We have proven the following result in [10].

**Theorem 1.** *Let $PN$ be a Petri net. Then $CL(PN) = L^{tk}_{dicausal}(PN)$.*

In [3] so called swapping equivalence on processes is defined.

**Definition 16 (Swapping).** *Let $PN$ be a Petri net. Let $K = (O, \rho)$, be a process of $PN$ with $O = (B \cup E, G)$. Let $b_1, b_2 \in B$, $b_1 \, co \, b_2$ w.r.t. $G^+$ and $\rho(b_1) = \rho(b_2)$. Define $G_1 = \{(b_1, e) \mid (b_2, e) \in G\}$ and $G_2 = \{(b_2, e) \mid (b_1, e) \in G\}$. Define $G' = G_1 \cup G_2 \cup (G \cap (E \times B)) \cup (G \cap ((B \setminus \{b_1, b_2\}) \times E))$. $G'$ is obtained from $G$ by interchanging arcs from $b_1$ and $b_2$. Finally, define $swap(K, b_1, b_2) = ((B, E, G'), \rho)$.*

**Definition 17 (Swapping Relation).** *Let $K_1 = ((B \cup E, G), \rho)$ and $K_2$ be processes of $PN$. Let us define $K_1 \equiv_1 K_2$ if there are conditions $b_1, b_2 \in B$ satisfying $b_1 \, co \, b_2$ w.r.t. $G^+$, $\rho(b_1) = \rho(b_2)$ and $K_2$ is (isomorphic to) $swap(K_1, b_1, b_2)$.*

It is easy to see that $\equiv_1$ is symmetric. Thus, $\equiv_1^*$ is an equivalence relation on processes of $PN$.

**Definition 18 (Swapping Equivalence).** *The equivalence relation $\equiv_1^*$ on processes of $PN$ is called* swapping equivalence*. The equivalence classes of processes w.r.t. the swapping equivalence are called* commutative processes *of $PN$.*

We extend the swapping equivalence to canonical LDAGs: Given two processes $K_1, K_2$ of a Petri net $PN$, we define $ldag_{K_1} \equiv_1^* ldag_{K_2}$ whenever $K_1 \equiv_1^* K_2$. Based on the results in [10] we state that the extension equivalence restricted to $L^{io}_{dicausal}$ and swapping equivalence coincide:

**Theorem 2.** *For each $ldag_1, ldag_2 \in L^{io}_{dicausal} : ldag_1 \equiv_1^* ldag_2 \Leftrightarrow ldag_1 \equiv ldag_2$.*

## 3.4 Pomset Token Flows and Enabled Pomsets

In this subsection we recall the definition of enabled pomsets, also known as partial words [9,12,19] and discuss their relationship to pomset token flows.

**Definition 19 (Enabled Pomset).** *Let $PN = (P, T, consume, produce, initial, final)$ be a Petri net. A pomset $lpo = (V, <, l)$ with $l : V \to T$ is* enabled to occur *in $PN$ if the following statements hold:*

*(a) For each co-set $S$ of $(V, <)$:*

$$initial + \sum_{v \in V \wedge v < S} (produce(l(v)) - consume(l(v))) \geq \sum_{v \in S} consume(l(v)).$$

*(b) $m = initial + \sum_{v \in V} (produce(l(v)) - consume(l(v))) \in final$.*

*We say that occurrence of $lpo$ leads from $initial$ to $m$ and $m$ is the final marking of the $lpo$. The enabled pomsets are also called* partial words of $PN$ *and the language of all enabled pomsets is called the* partial language of $PN$ *and denoted by $PL(PN)$.*

Actually, the definition of enabledness can be reformulated considering only slices of labelled partial orders (for the proof see e.g. [19]). In [10] we have proven the following result.

**Theorem 3.** *Let $PN$ be a Petri net. Then $PL(PN) = L^{nio}_{pomset}(PN)$.*

## 3.5 N-Free Token Flows and Rewrite Terms

In this subsection we establish the relationship between N-free token flows and rewriting semantics originally introduced in [15]. In this subsection we write $t : m \to m'$ to denote that $t \in T$, $consume(t) = m$ and $produce(t) = m'$.

**Definition 20 (Rewrite Term Semantics).** *Let $PN = (P, T, consume, produce, initial, final)$ be a Petri net. The set of general rewrite terms $\mathcal{GT}(PN)$ of $PN$ is defined inductively by the following production rules:*

$$\frac{m \in \mathbb{N}^P}{m : m \to m \in \mathcal{GT}(PN)}$$

$$\frac{t \in T}{t : consume(t) \to produce(t) \in \mathcal{GT}(PN)}$$

$$\frac{\alpha_1 : m_1 \to m_1' \in \mathcal{GT}(PN) \wedge \alpha_2 : m_2 \to m_2' \in \mathcal{GT}(PN)}{(\alpha_1 \parallel \alpha_2) : m_1 + m_2 \to m_1' + m_2' \in \mathcal{GT}(PN)}$$

$$\frac{\alpha_1 : m \to m' \in \mathcal{GT}(PN) \wedge \alpha_2 : m' \to m'' \in \mathcal{GT}(PN)}{(\alpha_1 ; \alpha_2) : m \to m'' \in \mathcal{GT}(PN)}$$

*These rules define binary operations, called* concurrent composition ($\|$) *and* sequential composition ($;$) *of rewrite terms. The set of* rewrite terms of $PN$ *denoted by $\mathcal{T}(PN)$ is the subset of $\mathcal{GT}(PN)$ given by $\mathcal{T}(PN) = \{\alpha : initial \to m \in \mathcal{GT}(PN) \mid m \in final\}$.*

*Given a rewrite term $\alpha : m \to m'$, we shortly say that $\alpha$ is a term and we denote by $pre(\alpha) = m$ the initial marking and by $post(\alpha) = m'$ the final marking of $\alpha$.*

**Definition 21 (Pomset of a Rewrite Term).** *Define inductively the pomset $lpo_\alpha$ of a term $\alpha$:*

- *Given a marking $m$, $lpo_m = (\emptyset, \emptyset, \emptyset)$.*
- *Given a transition $t \in T$, $lpo_t = (\{v\}, \emptyset, l)$, where $l(v) = t$.*
- *Given terms $\alpha_1$ and $\alpha_2$ with $lpo_{\alpha_1} = (V_1, <_1, l_1)$ and $lpo_{\alpha_2} = (V_2, <_2, l_2)$, $lpo_{\alpha_1 \| \alpha_2} = (V_1 \cup V_2, <_1 \cup <_2, l_1 \cup l_2)$, where $V_1$ and $V_2$ are assumed to be disjoint (what can be achieved by appropriate renaming of nodes).*
- *Given terms $\alpha_1$ and $\alpha_2$ with $lpo_{\alpha_1} = (V_1, <_1, l_1)$ and $lpo_{\alpha_2} = (V_2, <_2, l_2)$, $lpo_{\alpha_1;\alpha_2} = (V_1 \cup V_2, <_1 \cup <_2 \cup \{(a,b) \mid a \in V_1, b \in V_2\}, l_1 \cup l_2)$, where $V_1$ and $V_2$ are assumed to be disjoint (what can be achieved by appropriate renaming of nodes).*

*The language of pomsets associated to all rewrite terms of $PN$ is denoted by $TL(PN)$. The elements of $TL(PN)$ are called* term pomsets of $PN$.

The previous definition is sound in the sense, that the structures attached to terms are pomsets. Pomsets of rewrite terms of Petri nets coincide with so called finite series-parallel pomsets, i.e. with pomsets generated from single element pomsets by concurrent composition (disjoint union side by side) and sequential composition. A finite pomset is series-parallel iff it is N-free (for a proof see e.g. [6]). As a consequence we get the following result based on [10]:

**Theorem 4.** *Let $PN$ be a Petri net. Then $TL(PN) = L^{nio}_{Nfree}(PN)$.*

Rewrite terms are identified by an equivalence relation $\sim$ which preserves the operations $\|$ and ; (i.e. by a congruence w.r.t. the operations $\|$ and ;), given by the following axioms: Let $m, m' \in \mathbb{N}^P$ and $\alpha_1, \alpha_2, \alpha_3, \alpha_4$ be rewrite terms.

(1) $(\alpha_1 \parallel \alpha_2) \sim (\alpha_2 \parallel \alpha_1)$.
(2) $((\alpha_1; \alpha_2); \alpha_3) \sim (\alpha_1; (\alpha_2; \alpha_3))$, whenever these terms are defined.
(3) $((\alpha_1 \parallel \alpha_2) \parallel \alpha_3) \sim (\alpha_1 \parallel (\alpha_2 \parallel \alpha_3))$.
(4) $((\alpha_1 \parallel \alpha_2); (\alpha_3 \parallel \alpha_4)) \sim ((\alpha_1; \alpha_3) \parallel (\alpha_2; \alpha_4))$, whenever these terms are defined.
(5) $(\alpha_1; post(\alpha_1)) \sim \alpha_1 \sim (pre(\alpha_1); \alpha_1)$.
(6) $m + m' \sim (m \parallel m')$
(7) $\alpha_1 + \emptyset \sim \alpha_1$ for the empty multiset $\emptyset$.

Observe that for any two equivalent terms $\alpha_1 \sim \alpha_2$, we have $pre(\alpha_1) = pre(\alpha_2)$ and $post(\alpha_1) = post(\alpha_2)$.

We extend the equivalence $\sim$ to the set $TL(PN)$: Given two terms $\alpha_1, \alpha_2$ of a Petri net $PN$, we define $lpo_{\alpha_1} \sim lpo_{\alpha_2}$ whenever $\alpha_1 \sim \alpha_2$. Based on the results in [10] we state that the extension equivalence restricted to $TL(PN)$ and $\sim$-equivalence coincide:

**Theorem 5.** *For each $ldag_1, ldag_2 \in TL(PN) : ldag_1 \sim ldag_2 \Leftrightarrow ldag_1 \equiv ldag_2$.*

## 3.6 Somset Token Flows and Step Sequences

We briefly mention the relationship between step sequences and somset token flows.

**Definition 22 (Step Sequence).** *Let $PN$ be a Petri net. A finite sequence of steps of $PN$ $\sigma = s_1 \dots s_n$ $(n \in \mathbb{N})$ is called a* step sequence of $PN$ *if there exists a sequence of markings $m_1, \dots, m_n$ such that initial $\xmapsto{s_1} m_1 \xmapsto{s_2} \dots \xmapsto{s_n} m_n$ and $m_n$ is a legal final marking of $PN$.*

**Definition 23.** *Let $PN$ be a Petri net. Let $\sigma = s_1 \dots s_n$ be a step sequence of $PN$. Then the somset $lso_\sigma = (V, \prec, l)$ with $l : V \to T$ and with cuts $S_1, \dots S_n$ satisfying $|S_i| = s_i$ and $i < j \Rightarrow S_i \prec S_j$ for every $i, j \in \{1, \dots n\}$ is associated to $\sigma$. The language of somsets associated to all step sequences of $PN$ is denoted by $SL(PN)$.*

**Theorem 6.** *Let $PN$ be a Petri net. Then $SL(PN) = L^{nio}_{somset}$.*

## 3.7 Tomset Token Flows and Occurrence Sequences

Finally, we discuss the relationship between occurrence sequences and tomset token flows.

**Definition 24 (Occurrence sequence).** *Let $PN$ be a Petri net. A finite sequence of transitions of $PN$ $\sigma = t_1 \dots t_n$ $(n \in \mathbb{N})$ is called* occurrence sequence of $PN$ *if there exists a sequence of markings $m_1, \dots, m_n$ such that initial $\xmapsto{t_1} m_1 \xmapsto{t_2} \dots \xmapsto{t_n} m_n$ and $m_n$ is a legal final marking of $PN$.*

**Definition 25.** *Let $PN$ be a Petri net. Let $\sigma = l(v_1) \dots l(v_n)$ is an occurrence sequence of $PN$. Then the tomset $lto = (\{v_1, \dots, v_n\}, \prec, l)$ satisfying $i < j \Rightarrow v_i \prec v_j$ for every $i, j \in \{1, \dots n\}$ is associated to occurrence sequence $\sigma$. The language of tomsets associated to all occurrence sequences of $PN$ is denoted by $OL(PN)$.*

**Theorem 7.** *Let $PN$ be a Petri net. Then $OL(PN) = L^{nio}_{tomset}$.*

In [3] the exchange equivalence on occurrence sequences is defined.

**Definition 26 (Exchange Relation).** *Let $PN$ be a Petri net.*
*Let $\sigma_1 = t_1 \dots t_{i-1} t_i t_{i+1} t_{i+2} \dots t_n, \sigma_2 = t_1 \dots t_{i-1} t_{i+1} t_i t_{i+2} \dots t_n$ be occurrence sequences of $PN$. Then $\sigma_1 \equiv_0 \sigma_2$ iff $\sigma = \{t_1\} \dots \{t_{i-1}\}\{t_i, t_{i+1}\}\{t_{i+2}\} \dots \{t_n\}$ is a step sequence of $PN$.*

It is easy to see that $\equiv_0$ is symmetric and therefore $\equiv_0^*$ is an equivalence relation.

**Definition 27 (Exchange Equivalence).** *The equivalence relation $\equiv_0^*$ on occurrence sequences of $PN$ is called* exchange equivalence.

Based on the results in [10] we state that the extension equivalence restricted to $OL(PN)$ and exchange equivalence coincide:

**Theorem 8.** *For each $lto_1, lto_2 \in OL(PN) : lto_1 \equiv_0^* lto_2 \Leftrightarrow lto_1 \equiv lto_2$.*

## 4 Results and Related Works

As we mentioned in Introduction, motivation for introducing token flows was to have not only a unifying framework for different flavours of Petri net semantics, but to have also a simple formalism and effective technique to solve problems. In this section we briefly discuss the results achieved using token flows during the last four years, including the first polynomial test for the acceptance of a partial word by a Petri net, the first algorithm for synthesis of Petri nets from partial languages and token flow unfolding.

*Testing Pomsets of Petri Nets:* The important question arise: Given a pomset $lpo$ an a Petri net $PN$, is $lpo$ enabled in $PN$? The definition of enabledness of pomsets is inherently exponential, since a pomset can have exponentially many cuts in the number of nodes. That means, the definition is not appropriate to develop a test for partial words.

Using the fact that partial language $PL(PN)$ equals $L_{pomset}^{nio}$ of $PN$, which is obtained by forgetting token flow function and the entry and the exit in token flows from $L_{pomset}^{tk}$, the problem is reduced to answer the question : Given a SESE pomset, can we label its arcs by a token flow function to get a token flow of $PN$? The answer is positive, if and only if the token flow system of the pomset is solvable in nonnegative integers. Unfortunately, the solvability of a system of linear equations in nonnegative integers is in general NP-complete [20]. That means, to use a general algorithm for solving linear equations in nonnegative integers is not appropriate to develop a test for partial words.

In [11,14] we present algorithms to test a partial word in polynomial time, i.e we answer the question whether a pomset $lpo$ belongs to $L_{pomset}^{x}(PN)$, where $x \in \{io, nio\}$, in a polynomial time. In [11,14] we have shown that decision whether a pomset is a minimal causal pomset, i.e. whether a pomset $lpo$ belongs to $L_{causal}^{x}(PN)_m in$, where $x \in \{io, nio\}$, can be obtained in a polynomial time.

*Synthesis of Petri Nets from Pomset Languages:* In papers [13,2] token flows are used to synthetize a Petri net $PN$ (with all markings beeing final) from a pomset language $PL$

(closed w.r.t. extension and prefixes) in such a way that either $PL = L^{nio}_{pomset}(PN)$ or $PL \subseteq L^{nio}_{pomset}(PN)$ and there is no Petri net $PN'$ satisfying $PL \subseteq L^{nio}_{pomset}(PN') \subset L^{nio}_{pomset}(PN)$). Obviously, the labels of the pomsets give transitions $T$. The synthesis reduces to finding places, the values of $consume$ and $produce$ functions as well as the initial marking in such a way, that still all pomsets are accepted. In the synthesis procedure, the pomsets are extended by adding a single entry and a single exit. The entry is labelled in all pomsets by the same unique symbol, not used as a label of other nodes. The exit in each pomset is labelled by a different symbol, not used in the labels of other nodes. The main idea is to consider simple token flow functions, which attach a nonnegative integer to each arc in each pomset. If such a simple token flow function fulfils that equally labelled nodes in all pomsets have equal ingoing token flows and equal outgoing token flows, then it is called a token flow region and defines a place $p$. In the token flow region, we can speak about ingoing token flows and outgoing token flows of labels: the ingoing token flow of a label $t \in T$ defines the $consume(t)(p)$, the outgoing token flow of a label $t \in T$ defines $produce(t)(p)$, and the ingoing token flow of the entry label defines the initial marking of the place $p$. Adding a place determined by a token flow region still all pomsets will be accepted by the net. Adding places given by all regions, we get the seeked Petri net $PN$. Similarly to token flow systems of an LDAG, the token flow regions are nonnegative integer solutions of a system of linear equations, where the single equations just states that the equally labelled nodes in pomsets have equal ingoing and outgoing token flow. If the number of pomsets is finite, then the number of nodes is finite and the system have finite number of equations. The number of solutions, and therefore places, can still be infinite. Fortunatelly, it is enouhg to take places derived from the Hilbert basis of the system, which is finite. Namely, the net with the places derived from the Hilbert basis accept the same pomsets as the net $PN$.

*Token flow unfolding:* The idea of token flow unfolding presented in [1] is a straightforward extension of token flows. Instead of attaching a token flow function to pomsets, obtaining causal token flows, the idea is to attach a token flow function to prime event structures, to get token flow event structures, which are actually unions of causal token flows.

*Token flow Hasse diagrams:* Another idea to extend token flows can be found in the paper [17] in this volume. Instead of considering pomsets, authors consider Hasse diagrams, which are actually skeletons of LDAGs. It is shown in [17], that the token flow function of a pomset can be reconstructed from the extended token flow function of its skeleton, called interlaced flow. The interlaced flow attaches four multisets of tokens to each arc $v \rightarrow v'$ of the skeleton: the first multiset says how many tokens produced by $v$ are consumed by $v'$, the second says how many tokens produced by $v$ are consumed in the future of $v'$, the third counts how many tokens produced in the past of $v$ and consumed by $v'$; and the last multiset says how many tokens produced in the past of $v$ and consumed in the future of $v'$.

## References

1. Bergenthum, R., Lorenz, R., Mauser, S.: Faster Unfolding of General Petri Nets Based on Token Flows. In: van Hee, K.M., Valk, R. (eds.) PETRI NETS 2008. LNCS, vol. 5062, pp. 13–32. Springer, Heidelberg (2008)
2. Bergenthum, D.J., Lorenz, R., Mauser, S.: Synthesis of Petri Nets from Finite Partial Languages. Fundamenta Informaticae 88(4), 437–468 (2008)
3. Best, E., Devillers, R.: Sequential and Concurrent Behaviour in Petri Net Theory. Theoretical Computer Science 55(1), 87–136 (1987)
4. Degano, E., Meseguer, J., Montanari, U.: Axiomatizing the Algebra of Net Computations and Processes. Acta Informatica 33(7), 641–667 (1996)
5. Desel, J., Juhás, G.: What is a Petri Net? In: Ehrig, H., Juhás, G., Padberg, J., Rozenberg, G. (eds.) APN 2001. LNCS, vol. 2128, pp. 1–25. Springer, Heidelberg (2001)
6. Gischer, J.L.: The equational theory of pomsets. Theoretical Computer Science 61(2-3), 199–224 (1988)
7. Goltz, U., Reisig, W.: The Non-Sequential Behaviour of Petri Nets. Information and Control 57(2-3), 125–147 (1983)
8. Goltz, U., Reisig, W.: Processes of Place/Transition Nets. In: Díaz, J. (ed.) ICALP 1983. LNCS, vol. 154, pp. 264–277. Springer, Heidelberg (1983)
9. Grabowski, J.: On Partial Languages. Fundamenta Informaticae 4(2), 428–498 (1981)
10. Juhás, G.: Are these events independent? It depends! Habilitation thesis, Katholic University Eichstätt-Ingolstadt (2005)
11. Juhás, G., Lorenz, R., Desel, J.: Can I Execute my Scenario in Your Net? In: Ciardo, G., Darondeau, P. (eds.) ICATPN 2005. LNCS, vol. 3536, pp. 289–308. Springer, Heidelberg (2005)
12. Kiehn, A.: On the Interrelationship between Synchronized and Non-Synchronized Behavior of Petri Nets. Journal Inf. Process. Cybern. EIK 24(1-2), 3–18 (1988)
13. Lorenz, R., Juhás, G.: Toward Synthesis of Petri Nets from Scenarios. In: Donatelli, S., Thiagarajan, P.S. (eds.) ICATPN 2006. LNCS, vol. 4024, pp. 302–321. Springer, Heidelberg (2006)
14. Lorenz, R., Juhás, G., Bergenthum, R., Desel, J., Mauser, S.: Executability of scenarios in Petri nets. Theoretical Computer Science 410(12-13), 1190–1216 (2009)
15. Meseguer, J., Montanari, U.: Petri nets are monoids. Information and Computation 88(2), 105–155 (1990)
16. Priese, L.: Semi-rational sets of dags. In: De Felice, C., Restivo, A. (eds.) DLT 2005. LNCS, vol. 3572, pp. 385–396. Springer, Heidelberg (2005)
17. Oliveira, M.: Hasse Diagram Generators and Petri Nets. In: Petri Nets 2009. LNCS. Springer, Heidelberg (to appear, 2009)
18. Pratt, V.: Modelling Concurrency with Partial Orders. Int. Journal of Parallel Programming 15(1), 33–71 (1986)
19. Vogler, W.: Partial words versus processes: a short comparison. In: Rozenberg, G. (ed.) APN 1992. LNCS, vol. 609, pp. 292–303. Springer, Heidelberg (1992)
20. Schrijver, A.: Theory of linear and integer programming. Wiley, Chichester (1986)

# Reaction Systems: A Formal Framework for Processes

Grzegorz Rozenberg

Leiden University, Leiden Centre of Advanced Computer Science, Niels Bohrweg 1, 2333 CA Leiden, The Netherlands

**Abstract.** The functioning of a living cell consists of a huge number of individual reactions that interact with each other. These reactions are regulated, and the two main regulation mechanisms are facilitation/acceleration and inhibition/retardation. The interaction between individual biochemical reactions takes place through their influence on each other, and this influence happens through the two mechanisms mentioned above.

In our lecture we present a formal framework for the investigation of biochemical reactions - it is based on reaction systems. We motivate this framework by explicitely stating a number of assumptions/axioms that (we believe) hold for a great number of biochemical reactions - we point out that these assumptions are very different from the ones underlying traditional models of computation such as Petri Nets. We discuss some basic properties of reaction systems, and demonstrate how to capture and analyze, in our formal framework, some biochemistry related notions.

The lecture is of a tutorial character and self-contained.

# Simple Composition of Nets

Wolfgang Reisig

Humboldt-Universität zu Berlin

**Abstract.** Petri nets are frequently composed of given nets. Literature suggests a lot of different composition operators, for different purposes and different classes of Petri nets. Formal definitions are frequently surprisingly technical, not matching the intuitive elegance of their graphical counterpart.

We provide the formal framework for a simple composition operator, adequate for many classes of Petri net applications. It requires a minimum of fairly intuitive technicalities from its users and readers. The operator furthermore is associative, thus meeting the minimal algebraic requirements when composing a large system out of several smaller ones.

## 1 Introduction

Composition of nets is a fundamental principle to construct large system models. A lot of different composition operators have been defined during recent decades. Many operators come with quite detailed technicalities that the user may not really be interested in.

In this contribution we suggest a composition operator that can be applied in many different areas of Petri nets. Formulated more precisely, we present a fairly large class $C$ of nets together with a composition operator, with following properties:

Firstly, any two nets $N_1, N_2 \in C$ can be composed, resulting in a net $N_1 \cdot N_2 \in C$. In particular, the operator "$\cdot$" is not parameterized i.e. does not decompose into a family $\cdot_i$ of operators, with $i$ any kind of labelling. We rather suggest just *one* operator

$$\cdot : C \times C \rightarrow C. \tag{1}$$

In particular, for $N \in C$, the composition $N \cdot N$ is well defined.

The second property, decisively contributing to simplicity, is *associativity* of composition: For any three nets $N_1, N_2, N_3 \in C$,

$$(N_1 \cdot N_2) \cdot N_3 = N_1 \cdot (N_2 \cdot N_3). \tag{2}$$

Associativity of composition guarantees that a large net $N$, composed from a set of smaller nets, is independent of the history of its construction.

Associativity is the bare minimum for an algebraically satisfying theory of composition. This is why e.g. composition of symbol sequences over an alphabet $\Sigma$ is defined as an associative operation. True, composition of Petri nets has frequently been defined by means of non-associative operators. Inspection of their application might reveal, however, that a different, associative operator might be both intuitive and technically more simple to apply.

We will show a series of types of nets with quite different character, properties and application areas, all equipped with a composition operator that follows the above described principles.

The rest of this paper is organized as follows: Section 2 presents a choice of different examples where classes of nets are to be composed in different manners. These examples set standards for the generality and variability of the operator. The operator itself will then be defined in Sec. 3. Section 4 discusses advantages and limits of the operator. The non-trivial proof of the operator's associativity is outlined in the appendix.

## 2 A Choice of Examples

We select different areas where Petri nets are composed: Concurrent runs, service nets and branching processes. Additionally we consider *nondeterministic* composition of nets.

### 2.1 Composition of Concurrent Runs

A concurrent run of an elementary system net $N$ is itself a net, labelled by the elements of $N$. For the sake of simplicity we stick to *finite* concurrent runs. A concurrent run of $N$, starting in the initial state may consist of "snippets". Fig. 1 shows an example of a 1-bounded elementary system net $N$. Typical run snippets of $N$ are shown in Fig. 2.

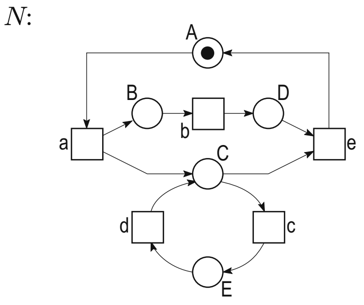


**Fig. 1.** Elementary system net, $N$

We are interested in the *composition* of such snippets. Fig. 3 shows two examples of runs, composed from the snippets of Fig. 2. Intuitively and graphically this kind of composition is fairly simple. The formal counterpart of a composition operator that would allow to write the runs of Fig. 3 as

$$r_1 \cdot r_2 \cdot r_1 \text{ and } r_1 \cdot r_3 \cdot r_2 \tag{3}$$

should likewise be simple.

### 2.2 Composition of Service Nets

As an entirely different kind of example, Fig. 4 shows a *service net*, buyer. This net shows how a buyer component orders goods, expects to receive an invoice, will pay the invoice, and – concurrently to those actions – receives the goods. The four places order, invoice, payment and goods build the component's *environment*.

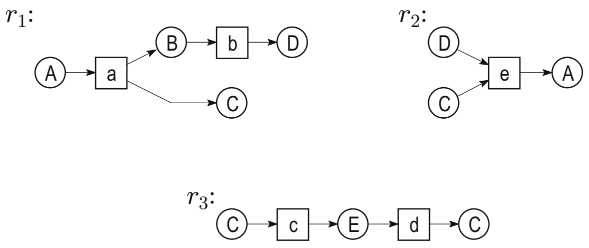


**Fig. 2.** Three run snippets of the net N of Fig. 1

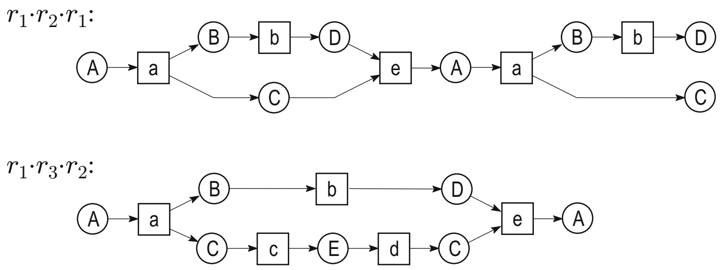


**Fig. 3.** Two concurrent runs, composed from the run snippets of Fig. 2

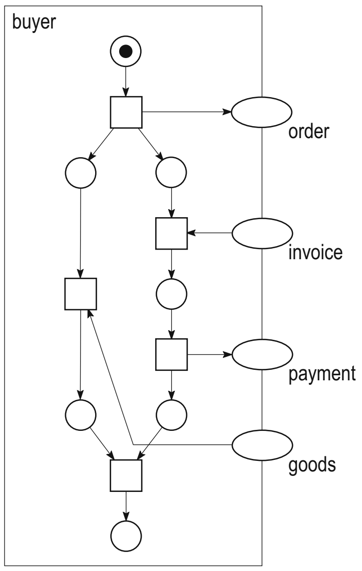
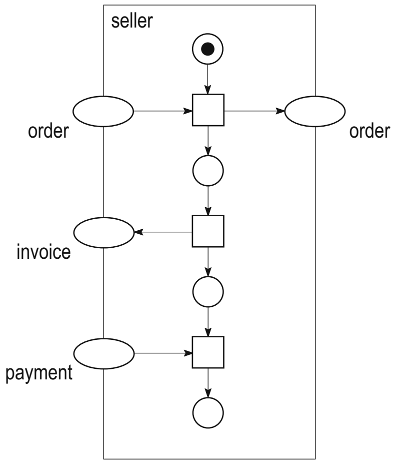


**Fig. 4.** Service net of the buyer

We may assume a retailer company, running a sales department (seller) to accept the buyer's orders, to send an invoice back to the buyer, to receive the payment, and to advice the companies' warehouse to deliver the ordered goods. Fig. 5 shows these two components. The job of the warehouse is to just deliver goods, upon the sales department's request.

An *interface net* such as in Fig. 4 or Fig. 5 is a marked net with a distinguished set of *interface places*. Graphically, an interface net is drawn inside a box, with the interface places on its surface.

Composition $N_1 \cdot N_2$ of two interface nets $N_1$ and $N_2$ "glues" $N_1$ and $N_2$ along their interface places: Some places are turned into internal places, the rest constitutes the interface of $N_1 \cdot N_2$. The internal structure of $N_1 \cdot N_2$ is just the union of the internal structure of $N_1$ and $N_2$, and the newly gained internal places.



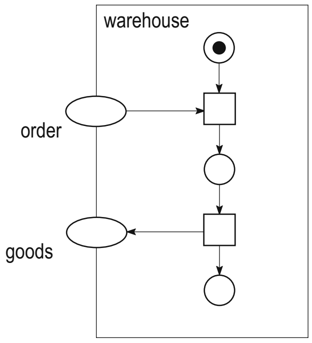


**Fig. 5.** Service nets of the seller and the warehouse

As an example, Fig. 6 describes the retailer company as the composition of the seller and the warehouse. At the end of the day we will write

$$\mathsf{retailer} =_{def} \mathsf{seller} \cdot \mathsf{warehouse}. \tag{4}$$

The place order that links the seller and the warehouse is turned into an internal place. However, the goods place of the warehouse is an interface place of the retailer, as it is required to communicate with the buyer.

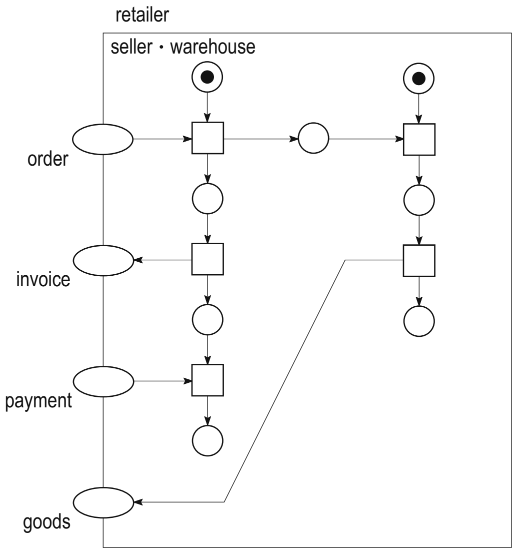


**Fig. 6.** Service net seller · warehouse

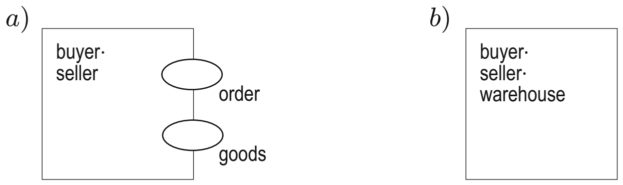


**Fig. 7.** Some more compositions of components

Fig. 7 shows the resulting interfaces of some more compositions. As a – somewhat unusual – composition, buyer·seller "glues" the three places at the left side of seller with the corresponding places of buyer and turns them into inner places. There remain two places, the seller's order on its right side, and the buyer's goods. This net provides a perfect environment for the warehouse. Finally, composing all three components, the environment remains empty.

As a variant, the retailer company may run a *second* warehouse, warehouse', that alternatively to the given one may deliver goods. The two warehouses share their environment places order and goods, as indicated in Fig. 8. Consequently, the interface of the composition

$$\mathsf{warehouse}' \cdot \mathsf{warehouse}$$

should be identical with the interface of warehouse alone.

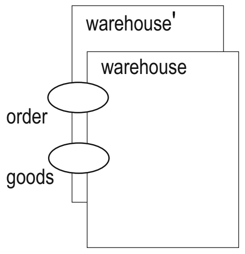


**Fig. 8.** Alternative warehouses

## 2.3 Composition of Branching Processes

With the alternative warehouse in Fig. 8 we have seen already a case where two nets $N_1$ and $N_2$ are composed such that in $N_1 \cdot N_2$, some elements of $N_1$ and of $N_2$ access the same interface places. This kind of composition is also required whenever *branching processes* are to be composed.

For example, Fig. 9 shows an initial part of the branching process of the system net $N$ of Fig. 1. The problem here is to identify a small number of run snippets and a composition operator such that Fig. 9 can be written as a composition of snippets.

## 2.4 Nondeterministic Composition of Nets

In the examples considered so far, places of interfaces of different nets may be equally labelled. But different places of the interface of *one* net were never equally labelled. This is however not what we require. Typical examples are concurrent runs of marked nets that are not 1-bounded. Fig. 10 shows an example of a system net $N$ with initially *two* tokens at place A, together with two concurrent

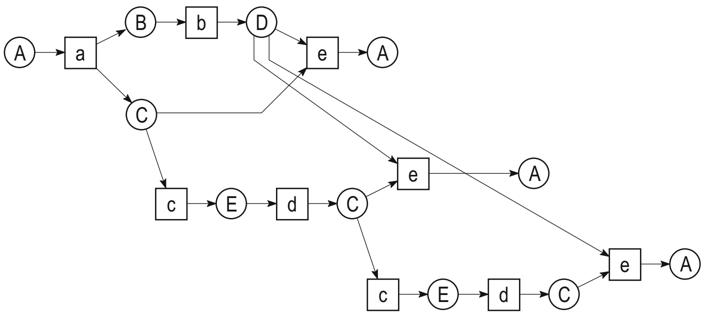


**Fig. 9.** Prefix of the branching process of the net $N$ as in Fig. 1

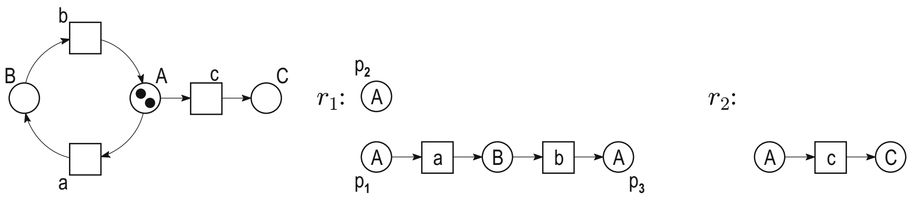


**Fig. 10.** 2-bounded system net $N$ with two concurrent runs

runs $r_1$ and $r_2$ of $N$. The environment of $r_1$ contains two equally labelled places. This gives rise to *two* different composed runs, $r_3$ and $r_4$, as shown in Fig. 11. This is an example of *non-deterministic* composition.

## 2.5 Summary

Each of the above examples stands for a *class* of nets. It should be intuitively obvious how nets of each such class are to be composed. Together these classes cover a wide range of nets and corresponding composition operators found in the literature. The above examples happened to compose nets by merging places only. The operator will allow also transitions to be merged.

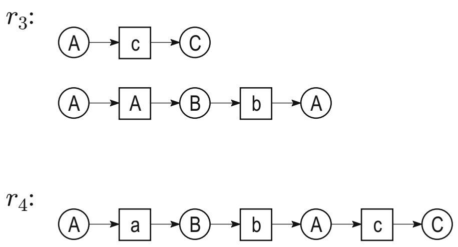


**Fig. 11.** Two versions of composing $r_1$ and $r_2$, yielding two concurrent runs, $r_3$ and $r_4$

## 3 A Composition Operator

### 3.1 The Idea of Ports

The above examples show that composition and decomposition of nets in many cases is intuitively fairly simple. The graphical representation of nets supports this claim decisively.

A formal definition of a composition operator should cover all aspects and variants as they occurred above, and coincidently remain as simple as intuition suggests.

Here we take a closer look at what the above examples do have in common. We furthermore outline the notion of *left* and *right port* of a net. These ports govern the composition of nets.

First of all we notice that the composition $N_1 \cdot N_2$ of two nets $N_1$ and $N_2$ always is the *union* of $N_1$ and $N_2$, with some pairs of equally labelled elements of $N_1$ and $N_2$ replaced by *one* element in $N_1 \cdot N_2$. However, not *all* equally labelled elements of $N_1$ and $N_2$ turn into one place in $N_1 \cdot N_2$, nor is each element of $N_1$ and $N_2$ necessarily labelled at all. To cope with this observation, a net $N$ that is intended to be composed with other nets has two kinds of elements: Elements of the *interface* of $N$ are those directly affected by the composition (i.e. those where upon composition, a new arrow will start or end). All other elements are *inner* elements of $N$.

For example, in $r_1$ of Fig. 2, composition affects the places labelled A, C and D. They constitute the interface of $r_1$. The place with label B and both transitions labelled a and b are the inner elements.

The decisive concept is the notion of *port*: The interface of a net $N$ is the union of *two* subsets $L_N$ and $R_N$ of elements of $N$, denoted as the *left* and *right port* of $N$. The two ports can reflect various different aspects of real systems. Typical examples include

- front end and back end,
- input and output,
- standard case and exception,
- buy side and sell side,
- customers and suppliers.

Composition of nets along their ports motivates the denotation of "left" and "right" port: In the composed net $N_1 \cdot N_2$, elements of the right port of $N_1$ and elements of the left port of $N_2$ are "glued" and turned into inner elements of $N_1 \cdot N_2$.

In concurrent runs such as $r_1$ in Fig. 2, the notions of "left port" and "right port" are particularly intuitive: The left port of $r_1$ contains the A-labelled place. Its right port consists of the two places labelled D and C. Consequently, for $r_3$, the left port as well as the right port consists of one place. Both are labelled by C.

Generally formulated, the ports of a concurrent run such as $r_1$, $r_2$ or $r_3$ in Fig. 2 are canonically defined: The left port comprizes all places with empty preset. The right port comprizes all places with empty postset. This principle is likewise applied in case of nondeterministic composition of concurrent runs, as discussed in Sec. 2.4.

Service nets, as considered in Sec. 2.2, have their interface places on their surrounding box. The places of its left and right port are placed on the left and right edge of the box, respectively.

Other classes of nets do not exhibit canonical ports. The designer of an interface net has the freedom to determine them according to his or her needs and interests. In particular, the two ports of an interface net are not necessarily assumed to be disjoint. This is in particular exploited when branching processes are composed as in Sec. 2.3. Taken to extreme, if a net is assumed to have just a unique interface $I$ instead of two ports, $I$ is conceived as the left as well as the right port.

## 3.2 Composing Nets along Their Ports

As a general setting, supported by all examples of Section 2, a port of a net $N$ is a subset of *labelled* elements of $N$. With $L_i$ and $R_i$ denoting the left and right ports of two nets $N_i$ ($i = 1, 2$), the following rule of thumb yields the composition $N_1 \cdot N_2$ of $N_1$ with $N_2$:

- Identify equally labelled elements in $R_1$ and $L_2$, glue them, and make them inner elements of $N_1 \cdot N_2$.
- The remaining elements of $L_2$ and $R_1$ go to the left and right port $L_{12}$ and $R_{12}$ of $N_1 \cdot N_2$, respectively.
- $L_1$ becomes a subset of $L_{12}$ and $R_2$ a subset of $R_{12}$.

Fig. 12 outlines this construct. The symbol — ∼ — links equally labelled elements of $R_1$ and $L_2$. The symbol — = — depicts their identification in $N_1 \cdot N_2$.

A closer look reveals however that composition can not be made as simple as this. In particular, we have to cope with the case where a port has two (or more) identically labelled elements.

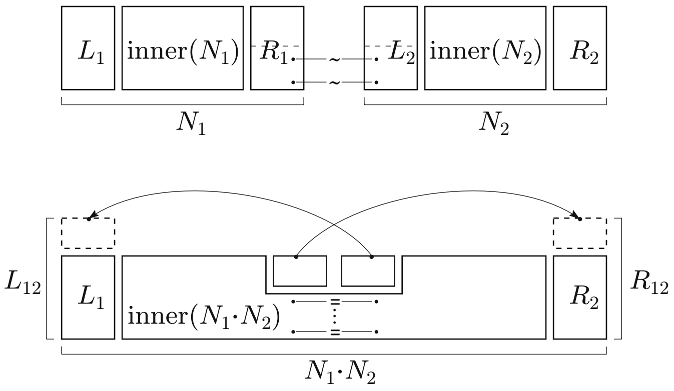


**Fig. 12.** A first idea to compose nets

One might be tempted to simply exclude such ports. But this would exclude important classes of nets too, including those considered in Sec. 2.4. Even worse, they anyway may arise as the result of composing two nets.

We construct a composition operator for nets where a port may very well have identically labelled elements. We do so with the mild assumption that identically labelled elements of a port are *ordered*.

Nevertheless we may start more generally with a net where one (or both) of its ports has identically labelled but unordered elements. Obviously, they can be ordered in different ways. With each such order we can proceed as described above. All together, they provide the means to express *nondeterministic* composition, as discussed in Sec. 2.4.

## 3.3 Interface Nets

Labels at net elements are usually employed to relate the elements to some items outside the net. Here, we concentrate on the technical aspects of labelling, and apply a convenience:

**Assumption 1.** In the rest of this paper, we assume a set $L$ of *labels*.

Labelling of a set is defined as usual:

**Definition 1.** *For a set $A$, a mapping $\lambda : A \to L$ is a* labelling *of $A$.*

Labelling is usually not injective, i.e. different elements may carry the same label.

**Notations.** If a labelling $\lambda$ of a set $A$ can be assumed from context, $A$ is said to be *labelled*, an element $a \in A$ is *$l$-labelled* if $\lambda(a) = l$, and $A_l$ denotes the set of $l$-labelled elements of A.

In the sequel we require the elements of a set $A_l$ of equally labelled elements to be *ordered*. We represent order by means of *indices*, i.e. numbers $1, 2, \ldots$. Technically this is achieved by means of an *index function*:

**Definition 2.** *Let $A$ be a set with $n$ elements. Then a bijective mapping $\delta : A \to \{1, \ldots, n\}$ is an* index function *for $A$.*

**Notations.** If an index function $\delta$ for a set $A$ can be assumed from context, $\delta(a)$ is called the *index of $a$ in $A$*.

As explained above already, we will consider labelled sets $A$ where each subset $A_l$ of equally labelled elements is indexed:

**Definition 3.** *Let $A$ be a finite, labelled set. For each label $l$, let $\delta_{A,l}$ be an index function of $A_l$. Then $A$ is* indexed *(by the index function $\delta_{A,l}$).*

We are now prepared to define the version of labelled nets as employed in the sequel. We first realize that places and transitions of a net definitely represent different kinds of items outside the net. This partitions the set $L$ of labels:

**Assumption 2.** In the sequel we assume the set $L$ of labels to consist in two disjoint sets $L_P$ and $L_T$.

Element labelled nets are now defined as usual:

**Definition 4. i.** *Let $P$ and $T$ be disjoint, finite sets, and let $F \subseteq (P \times T) \cup (T \times P)$. Then $N = (P, T, F)$ is a* net. *Elements in $P, T$ and $F$ are denoted as* places, transitions *and* arcs, *respectively.*

**ii.** *$N$ together with two labellings $\lambda_P : P \to L_P$ and $\lambda_T : T \to L_P$ is a* labelled net. *As a notation, $P \cup T$ contains the* elements of $N$*.*

We frequently require not every element of a net to be labelled. One may assume a "trivial" labelling (e.g. "$\varepsilon$") for the rest.

An interface net is now defined as a net together with two indexed subsets of elements, its *ports*:

**Definition 5.** *Let $N$ be a finite labelled net, and let $L$ and $R$ be indexed subsets of elements of $N$. Then $N$ together with $L$ and $R$ is an* interface net. *The sets $L$ and $R$ are the* left *and the* right ports of $N$, *respectively.*

This kind of nets has been motivated in Sec. 3.1 already. Composition along ports, as discussed in Sec. 3.2, requires some insight into properties of indexed sets.

## 3.4 Properties of Indexed Sets

If we want the union $A \cup B$ of two indexed sets $A$ and $B$ to be indexed again, we have to define the indexing of $A_l \cup B_l$, for each label $l$. This can not be achieved as a symmetrical operation (Which element gets index 1?). We rather suggest indexing of one set, $A_l$, say, to remain, and to place the elements of $B_l$ on top: $A_l$ is *extended* by $B_l$.

**Definition 6.** *Let $A$ and $B$ be finite, disjoint sets, indexed by index functions $\delta_{A,l}$ and $\delta_{B,l}$, respectively, for all labels $l$. Furthermore, let $n_l$ be the number of $l$-labelled elements of $A$ (i.e. $n_l = |A_l|$). Then the* extension of $A$ by $B$ *is the indexed set $C =_{def} A \cup B$, where for each label $l$, the index function $\delta_{C,l}$ is defined by*

$$\delta_{C,l}(c) = \begin{cases} \delta_{A,l}(c) & \text{if } c \in A \\ n_l + \delta_{B,l}(c) & \text{if } c \in B \end{cases}$$

As a notation, if $C$ is the extension of $A$ by $B$, we frequently say that *$C$ is $A$ extended by $B$*. Of course, the extension of $A$ by $B$ in general differs from the extension of $B$ by $A$.

Elements of two indexed sets may be *partners*: Both their indices are identical:

**Definition 7.** *Let $A$ and $B$ be indexed sets. Then $b \in B$ is a* partner of $a \in A$ *iff $a$ and $b$ carry the same label, and their indices in $A$ and in $B$ coincide.*

Of course, $a \in A$ has at most one partner $b \in B$. Furthermore, partners are elements with small indices:

**Lemma 1.** *Let $A$ and $B$ be indexed sets. To each label $l$ there exists a number $\tilde{l} \geq 0$ such that the partners in $A$ and $B$ have indices $1, \ldots, \tilde{l}$.*

*Proof.* Definitions 2, 3 and 7 imply for each $b \in B$ with a partner $a \in A$, both with some label $l$ and some index $k$: To each $i < k$ there exist elements $a' \in A$ and $b' \in B$ which are partners with label $l$ and index $i$. As $A_l$ and $B_l$ are finite, there is a greatest such $k$. With $k =_{def} 0$ in case $A$ and $B$ contain no partners with label $l$, choose $\tilde{l} =_{def} k$.

With $\tilde{l}$ as in the above Lemma, there may remain elements $a_{\tilde{l}+1}, \ldots, a_n$ in $A_l$ (i.e. those without a partner in $B_l$). They constitute the *overhead* of $A$ over $B$. This set is canonically indexed, with indices $\tilde{l}+1, \ldots, n$ of $A_l$ "dropping down" to $1, \ldots, n - \tilde{l}$.

**Definition 8.** *Let $A$ and $B$ be indexed sets. Then the* overhead of $A$ over $B$ *is the indexed set $C \subseteq A$ of all elements of $A$ without a partner in $B$. For each label $l$, the index function $\delta_{C,l}$ is defined by $\delta_{C,l}(a) = \delta_{A,l}(a) - \tilde{l}$, where $\tilde{l}$ is as in Lemma 1.*

### 3.5 Composition of Interface Nets

For disjoint nets $N_1$ and $N_2$ with indexed subsets $E_1$ and $E_2$ of elements, we define *the fusion $N_1 \cdot N_2$ along $E_1$ and $E_2$*. This is intuitively simple: $N_1 \cdot N_2$ is the union of $N_1$ and $N_2$, where each pair of partners in $E_1$ and $E_2$ is *one* element in $N_1 \cdot N_2$. Technically we retain $E_1$ and skip all elements $e_2$ of $E_2$ which have a partner $e_1$ in $E_1$. Arcs $(x, e_2)$ and $(e_2, y)$ of $N_2$ are replaced by $(x, e_1)$ and $(e_1, y)$ in $N_1 \cdot N_2$.

**Definition 9.** *For $i = 1, 2$ let $N_i = (P_i, T_i, F_i)$ be disjoint, labelled nets, and let $E_i$ be indexed subsets of their elements. Let $E \subseteq E_2$ be the subset of $E_2$-elements with a partner in $E_1$. The* fusion of $N_1$ and $N_2$ at $E_1$ and $E_2$ *is the net $N = ((P_1 \cup (P_2 \backslash E)), (T_1 \cup (T_2 \backslash E)), F)$, with*

$$(x,y) \in F \text{ iff } \begin{cases} (x,y) \in F_1 \cup F_2 \text{ and } x, y \notin E \\ x \in E_1,\ x \text{ has partner } z \text{ in } E, \text{ and } (z,y) \in F_2, \\ y \in E_1,\ y \text{ has partner } z \text{ in } E, \text{ and } (x,z) \in F_2, \\ x, y \in E_1,\ x \text{ and } y \text{ have partners } z_x \text{ and } z_y \text{ in } E, \\ \qquad \text{respectively, and } (z_x, z_y) \in F_2. \end{cases}$$

Fig. 13 depicts the 2nd and 3rd case of this definition.

In this definition, partners $x$ and $z$ (and likewise partners $y$ and $z$) are equally labelled (by Def. 7). Hence they either are both places or both transitions (by Assumption 2). Hence, the definition is consistent.

We are now ready to define *composition* $N_1 \cdot N_2$ of interface nets $N_1$ and $N_2$ just as the fusion of $N_1$ with $N_2$ at the right port $R_1$ of $N_1$ and the left port $L_2$ of $N_2$. To make $N_1 \cdot N_2$ again an interface net, the left port of $N_1 \cdot N_2$ extends the left port of $N_1$ by the left overhead of $N_2$. Consequently, the right port of $N_1 \cdot N_2$ extends the right port of $N_2$ by the right overhead of $N_1$.

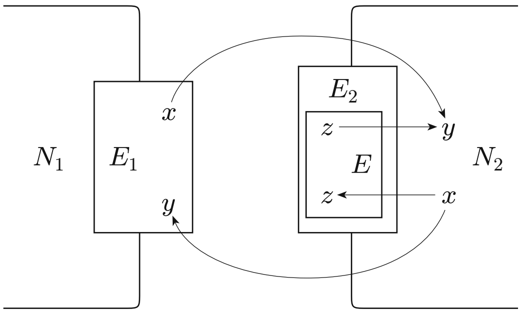


**Fig. 13.** Depicting the 2nd and 3rd case of Def. 9

**Definition 10.** *For $i = 1, 2$ let $N_i$ be disjoint interface nets with left and right ports $L_i$ and $R_i$. Then the* composition $N_1 \cdot N_2$ *of $N_1$ with $N_2$ is the fusion of $N_1$ with $N_2$ at $R_1$ and $L_2$. Furthermore, $N_1 \cdot N_2$ is an interface net with*

- *its left port $L_{12} =_{def} L_1$, extended by the overhead of $R_1$ over $L_2$*
- *its right port $R_{12} =_{def} R_2'$, extended by the overhead of $L_2$ over $R_1$, where each $x \in R_2 \cap L_2$ is in $R_2'$ replaced by $y$, if $x$ is in $L_2$ a partner of $y$ in $R_1$.*

Figure 14 shows two interface nets $N_1$ and $N_2$, as well as the products $N_1 \cdot N_2$ and $N_1 \cdot N_1$, exemplifying the most general configurations of labels. We follow the convention of drawing an element with higher index on top of an equally labelled element with lower index. Hence, in $N_2$, $\delta_{R_2,\mathsf{A}}(\mathsf{i}) = 1$ and $\delta_{R_2,\mathsf{A}}(\mathsf{j}) = 2$. The indices of $\mathsf{i}$ and $\mathsf{j}$ are retained in the right port $R_{12}$ of $N_1 \cdot N_2$; furthermore, $\delta_{R_{12},\mathsf{A}}(\mathsf{e}) = 3$. To construct $N_1 \cdot N_1$, the elements of the second instance of $N_1$ are primed. Consequently, $\delta_{L_{11},\mathsf{B}}(\mathsf{b}) = 1$, $\delta_{L_{11},\mathsf{B}}(\mathsf{b'}) = 2$, $\delta_{R_{11},\mathsf{A}}(\mathsf{e'}) = 1$ and $\delta_{R_{11},\mathsf{A}}(\mathsf{e}) = 2$.

As discussed above already, composition is associative:

**Theorem 1.** *Let $N_1, N_2, N_3$ be interface nets. Then $(N_1 \cdot N_2) \cdot N_3 = N_1 \cdot (N_2 \cdot N_3)$.*

Proof of this Theorem is not trivial at all. Its central arguments are outlined in the Appendix.

## 3.6 Examples Revisited

Turning back to the composition of nets in Sec. 2, we just have to indicate the ports of the involved nets. Composition then follows Definition 10.

The ports $L$ and $R$ of the run snippets $r_1, r_2, r_3$ in Fig. 2 are obvious: $L$ contains the places $p$ with empty pre-set ${}^{\bullet}p$, and $R$ the places $p$ with empty post-set $p^{\bullet}$. This convention makes $r_1, r_2$ and $r_3$ interface nets, with compositions as shown in Fig. 3. Matters are more involved for $r_1$ in Fig. 10, with its left port $L = \{\mathsf{p}_1, \mathsf{p}_2\}$ and its right port $R = \{\mathsf{p}_2, \mathsf{p}_3\}$. The index function $\delta_{R,\mathsf{A}}$ is crucial here, because both places of $R$ are equally labelled with $\delta_{R,\mathsf{A}}(\mathsf{p}_2) = 1$ and

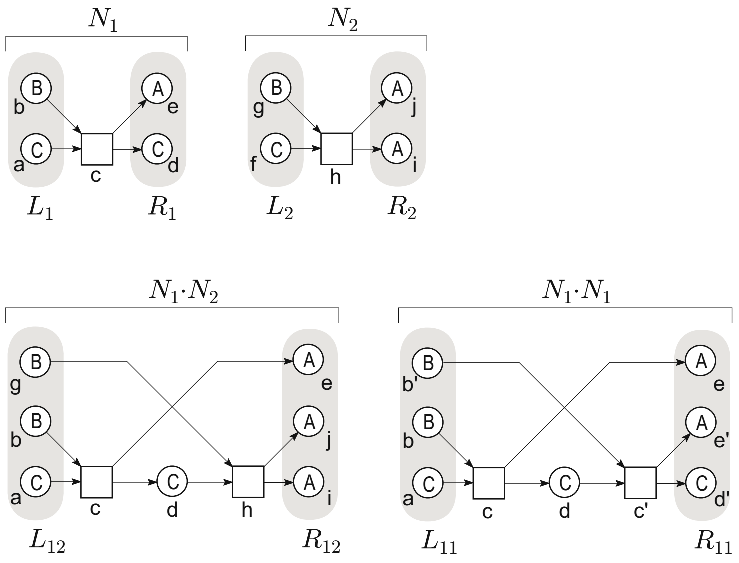


**Fig. 14.** Interface nets $N_1$ and $N_2$, and compositions $N_1 \cdot N_2$ and $N_1 \cdot N_1$

$\delta_{R,\mathsf{A}}(\mathsf{p}_3) = 2$, the composition $r_1 \cdot r_2$ returns $r_3$ of Fig. 11. Reversed indexing yields $r_4$ in Fig. 11.

Summing up, the ports $L$ and $R$ of a concurrent run contain the places $p$ with empty pre-set ${}^{\bullet}p$ and empty post-set $p^{\bullet}$, respectively. For 1-bounded system nets, these ports are injectively labelled. Otherwise composition is the non-deterministic combination of all potential index functions.

Each of the service nets in Fig. 4 – 7 is an interface net with the graphical representation depicting the elements of the left and right port at the left and

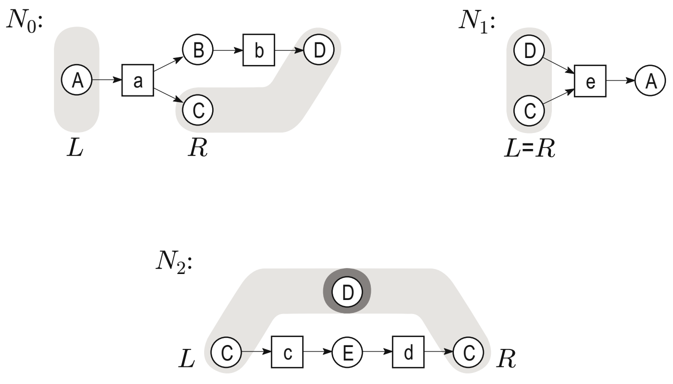


**Fig. 15.** Three concurrent runs, equipped with ports

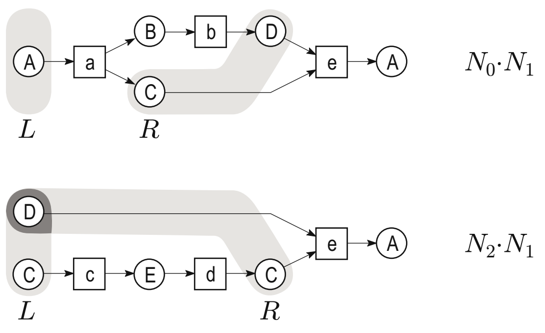


**Fig. 16.** The products $N_0 \cdot N_1$ and $N_2 \cdot N_1$ with their ports

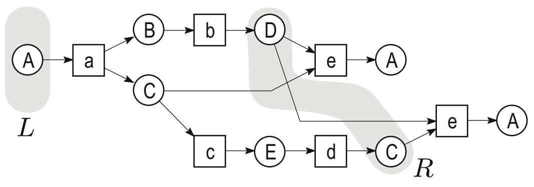


**Fig. 17.** The composition $N_0 \cdot N_1 \cdot N_2 \cdot N_1$ with its ports $L$ and $R$

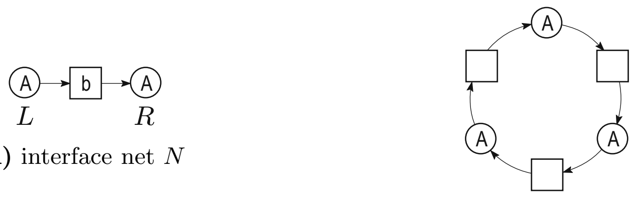
right side of the surrounding box, respectively. This fairly intuitive convention works for disjoint ports, including the case of empty ports, such as the right port of warehouse in Fig. 5. The variant warehouse' has the left port $L$ of warehouse as its left *and* its right port. Fig. 8 is a graphically unsatisfactory representation of warehouse' · warehouse.

The prefix in Fig. 9 of the branching process of $N$ in Fig. 1 can be composed from the concurrent run snippets of Fig. 2, with ports as shown in Fig. 15. Their intuitive meaning is fairly obvious: The labels of the elements of the right ports of $N_0, N_1$ and $N_2$ in Fig. 15 and of the composed nets in Fig. 16 are C and D. Hence, the ports describe the marking where alternatives can occur in the system net $N$ of Fig. 1. This is what Fig. 17 exploits. Even more, the prefix in Fig. 9 just reads

$$N_0 \cdot N_1 \cdot N_2 \cdot N_1 \cdot N_2 \cdot N_1.$$

## 4 Limits of This Operator

One may think of composition operators that are – in whatever sense – more general than the one suggested above. But utmost generality is not what makes life easier: What eventually will prevail is the right balance between intuitive simplicity, and sufficient generality. This is what the operator in this paper is intended to deliver.



**a)** interface net $N$

**b)** cycle of three instances of $N$

**Fig. 18.** Cyclic structure

## 4.1 Cyclic Structures

Composition of interface nets never yields cyclic structures. Fig. 18 shows a simple example. To gain the net in Fig. 18 $a$), one has to merge the left and right ports of the net $N \cdot N \cdot N$.

Technically formulated, partners $a$ and $b$ in indexed subsets $A$ and $B$ of a net $N$ are merged into *one* element. In analogy to Def. 9, technically we skip $b$ and link $a$ to the neighbors of $b$.

**Definition 11.** *Let $N = (P, T, F)$ be a labelled net. Let $A, B \subseteq P \cup T$ be indexed, and let $B' \subseteq B$ be the subset of $B$-elements with a partner in $A$. Then the net $(P \backslash B', T \backslash B', F')$ with*

$$(x, y) \in F' \textit{ iff } \begin{cases} (x, y) \in F \textit{ and } x, y \notin B', \\ x \in A \textit{ with partner } z \in B' \textit{ and } (z, y) \in F, \\ y \in A \textit{ with partner } z \in B' \textit{ and } (x, z) \in F, \\ x, y \in A \textit{ with partners } z_x \textit{ and } z_y\textit{, respectively,} \\ \qquad\qquad \textit{in } B'\textit{, and } (z_x, z_y) \in F. \end{cases}$$

*is the* fusion of $N$ at $A$ and $B$.

The cyclic *closure $N^c$ of an interface net $N$* is the fusion of $N$ at its ports, with the respective port overhead of $N$ as new ports of $N^c$:

**Definition 12.** *Let $N$ be an interface net with ports $L$ and $R$. Then the* cyclic closure $N^c$ of $N$ *is the fusion of $N$ at $L$ and $R$, with the overhead of $L$ over $R$, and the overhead of $R$ over $L$ as the left and right ports of $N^c$, respectively.*

As an example, Fig. 18 $b$) can now be written as

$$(N \cdot N \cdot N)^c,$$

with $N$ the net in Fig. 17 $a$).

## 4.2 Composition in the Style of Process Algebras

A port may contain places as well as transitions. Composition of nets with ports that contain transitions only, mimics synchronous composition as in process algebras. Vogler in [4] combines fusion of transitions with nondeterministic choice. Our framework fails to express his operator.

### 4.3 Place- and Arc Inscriptions

Our composition operator only affects bare net structures $(P, T, F)$. If token inscribed nets $N_1$ and $N_2$ are to be composed, tokens on partner places of $R_1$ and $L_2$ may be summed up at their joint place in $N_1 \cdot N_2$. This would retain the operator's associativity.

Arc inscriptions such as arc weights or formulas of high-level nets may cause a problem in case of an inscribed arc $p \to t$ or $t \to p$ with $p, t \in R_1$, and an inscribed arc $p' \to t'$ or $t' \to p'$, with $p', t' \in L_2$. Arc inscriptions of high level nets are usually multisets of terms with addition, or terms with addition. Adding the arc inscriptions would again preserve associativity.

### 4.4 Regulated Nondeterminism

In Sec. 2.4 we demonstrated by means of an example, that a port with two or more identically labelled elements may cause more than one composed net. This may be intended. If not intended, the net designer has to "individualize" identically labelled elements. We suggest to *index* such elements, i.e. to order them, as described in Sec. 3.6. This appears to be the most simple means of "individualization". Furthermore, each theoretically possible combination can be achieved this way.

One may suggest more complicated versions of nondeterministic choice, such as alternating outcome at each of its invocation. Our operator could not express this behaviour.

## 5 Conclusion

The composition operator suggested in this paper is intended to fullfill three requirements:

– It is simple to use: Upon composing two nets $N_1$ and $N_2$, one only has to identify ports of $N_1$ and $N_2$. The ports are then automatically re-organized in $N_1 \cdot N_2$.
– It is associative: This is crucial for an algebraically manageable, history independent and intuitively simple operator.
– It has widespread applications: This has been shown by means of examples. Many nets anyway have canonical left and right ports.

Our operator accounts only for the structure $(P, T, F)$ of nets. It was easy, however, to include (initial) markings or arc inscriptions in a number of ways. [1], [2] and [3] survey numerous versions of Petri Net composition operators. None of them meets all three of the above requirements.

## Acknowledgements

With pleasure I acknowledge valuable hints of one of the referees. Dirk Fahland, Christian Stahl and Walter Vogler contributed important hints at a previous version.

## References

1. Best, E., Devillers, R., Koutny, M.: Petri Net Algebra. Springer Monographs in Theoretical Computer Science (2001)
2. Christensen, S., Petrucci, L.: Modular Analysis of Petri Nets. Computer Journal 43(3), 224–242 (2001)
3. Gomes, L., Barros, J.P.: Structuring and Composability Issues in Petri Nets Modeling. IEEE Transactions on Industrial Informatics 1(2), 112–123 (2005)
4. Vogler, W.: Modular Construction and Partial Order Semantics of Petri Nets. LNCS, vol. 625. Springer, Heidelberg (1992)

## Appendix: Outline of Proof of Theorem 1

We employ shorthands for interface nets $N_x$ and $N_y$, writing $N_{xy}$ for $N_x \cdot N_y$, and $L_x$ and $R_x$ for $L_{N_x}$ and $R_{N_x}$, respectively. The Theorem's central claim then reads

$$N_{(12)3} = N_{1(23)}.$$

Furthermore, for indexed sets $A$ and $B$ we write
$A \sqcup B$ for the extension of $A$ by $B$, and
$A - B$ for the overhead of $A$ over $B$.

Throughout the proof, let $a \in L_1 \cup L_2 \cup L_3$. For an indexed set $A$ with $a \in A$,
– $\mathit{index}(a)$ denotes the index of $a$ in $A$, and
– $\bar{A}$ denotes $|A_l|$, with $l$ the label of $a$.

For an interface net $N$ let $\mathit{inner}(N)$ denote the set of elements of $N$ that are not in ports of $N$. From the Definitions follows:

**Lemma 2.** *Let $A, B, C$ be indexed sets. Then*

**a)** *$a \in A$ implies $a \in A \sqcup B$, with $\mathit{index}(A \sqcup B) = \mathit{index}(A)$.*
**b)** $\overline{A \sqcup B} = \bar{A} + \bar{B}$
**c)** $\overline{A - B} = \bar{A} - \bar{B}$

**Lemma 3.** *Let $x, y \in \{1, 2, 3, 12, 23\}$.*

**a)** *For $a \in \mathit{inner}(N_x)$ holds $a \in \mathit{inner}(N_{xy})$ and $a \in \mathit{inner}(N_{yx})$.*
**b)** *If $\overline{R_x} \geq \overline{L_y}$ then $\overline{L_{xy}} = \overline{L_x}$.*
**c)** *If $\overline{R_x} \leq \overline{L_y}$ then $\overline{R_{xy}} = \overline{R_y}$.*
**d)** *For $a \in L_y$ holds:*
*If $\overline{R_x} \geq \mathit{index}(L_y)$ then $a \in \mathit{inner}(N_{xy})$.*
*If $\overline{R_x} < \mathit{index}(L_y)$ then $a \in L_{xy}$, with $\mathit{index}(L_{xy}) = \overline{L_x} + \mathit{index}(L_y) - \overline{R_x}$.*

Proof of the Theorem is composed of three Lemmata.

**Lemma 4.** $L_{(12)3} = L_{1(23)}$

We distinguish three cases:

**Case 1:** $a \in L_1$

As a shorthand, let $n =_{def} index(L_1)$. We draw two conclusions:

a) $a \in L_1 \sqcup (L_2 - R_1)$ with $index(L_1 \sqcup (L_2 - R_1)) = n$.
Then $a \in L_{12}$ with $index(L_{12}) = n$.
Then $a \in L_{12} \sqcup (L_3 - R_{12})$ with $index(L_{12} \sqcup (L_3 - R_{12})) = n$.
Then $a \in L_{(12)3}$ with $index(L_{(12)3}) = n$.

b) $a \in L_1 \sqcup (L_{23} - R_1)$ with $index(L_1 \sqcup (L_{23} - R_1)) = n$.
Then $a \in L_{1(23)}$ with $index(L_{1(23)}) = n$.

**Case 2:** $a \in L_2$

We distinguish two subcases:

**Case 2.1:** $\overline{R_1} \geq index(L_2)$

We draw two conclusions:

a) $a \in inner(N_{12})$.
Then $a \in inner(N_{(12)3})$.
Then $a \notin L_{(12)3}$.

b) $a \in L_2 \sqcup (L_3 - R_2)$ with $index(L_2 \sqcup (L_3 - R_2)) = index(L_2)$.
Then $a \in L_{23}$ with $index(L_{23}) = index(L_2)$.
Then $a \in inner(N_{1(23)})$.
Then $a \notin L_{1(23)}$.

**Case 2.2:** $\overline{R_1} < index(L_2)$

As a shorthand, let $n =_{def} \overline{L_1} + index(L_2) - \overline{R_1}$. We draw two conclusions:

a) $a \in L_{12}$ with $index(L_{12}) = n$.
Then $a \in L_{12} \sqcup (L_3 - R_{12})$ with $index(L_{12} \sqcup (L_3 - R_{12})) = n$.
Then $a \in L_{(12)3}$ with $index(L_{(12)3}) = n$.

b) $a \in L_2 \sqcup (L_3 - R_2)$ with $index(L_2 \sqcup (L_3 - R_2)) = index(L_2)$.
Then $a \in L_{23}$ with $index(L_{23}) = index(L_2)$.
Then $a \in L_{1(23)}$ with $index(L_{1(23)}) = \overline{L_1} + index(L_2) - \overline{R_1}$.
Then $a \in L_{1(23)}$ with $index(L_{1(23)}) = n$.

**Case 3:** $a \in L_3$

We distinguish three subcases:

**Case 3.1:** $\overline{R_2} \geq index(L_3)$

We draw two conclusions:

a) $a \in inner(N_{23})$.
Then $a \in inner(N_{1(23)})$.
Then $a \notin L_{1(23)}$.

b) $\overline{R_2} + \overline{(R_1 - L_2)} \geq index(L_3)$.
Then $\overline{R_2 + (R_1 - L_2)} \geq index(L_3)$.
Then $\overline{R_{12}} \geq index(L_3)$.
Then $a \in inner(N_{(12)3})$.
Then $a \notin L_{(12)3}$.

**Case 3.2:** $\overline{R_2} < index(L_3)$ and $index(L_3) \leq \overline{R_{12}}$
We draw two conclusions:

a) $a \in inner(N_{(12)3})$.
Then $a \notin L_{(12)3}$.

b) $a \in L_{23}$ with $index(L_{23}) = \overline{L_2} + index(L_3) - \overline{R_2}$.
Then $a \in L_{23}$ with

$$\begin{aligned} index(L_{23}) &\leq \overline{L_2} + \overline{R_{12}} - \overline{R_2} \\ &= \overline{L_2} + \overline{(R_2 \sqcup R_1) - L_2} - \overline{R_2} \\ &= \overline{L_2} + (\overline{R_2 \sqcup R_1}) - \overline{L_2} - \overline{R_2} \\ &= \overline{L_2} + \overline{R_2} + \overline{R_1} - \overline{L_2} - \overline{R_2} \\ &= \overline{R_1}. \end{aligned}$$

Then $a \in inner(N_{1(23)})$.
Then $a \notin L_{1(23)}$.

**Case 3.3:** $\overline{R_{12}} < index(L_3)$ As a shorthand, let $n =_{def} \overline{L_1} + \overline{L_2} + index(L_3) - \overline{R_1} - \overline{R_2}$. We draw two conclusions:

a) $a \in L_{(12)3}$ with $index(L_{(12)3}) = \overline{L_{12}} + index(L_3) - \overline{R_{12}}$.
We distinguish two subcases:

aa) $\overline{R_1} \geq \overline{L_2}$. Then

$$\begin{aligned} index(L_{(12)3}) &= \overline{L_1} + index(L_3) - \overline{R_{12}} \\ &= \overline{L_1} + index(L_3) - (\overline{R_2} + (\overline{R_1} - \overline{L_2})) \\ &= n. \end{aligned}$$

ab) $\overline{R_1} < \overline{L_2}$. Then

$$\begin{aligned} index(L_{(12)3}) &= \overline{L_{12}} + index(L_3) - \overline{R_2} \\ &= \overline{L_1 \sqcup (L_2 - R_1)} + index(L_3) - \overline{R_2} \\ &= \overline{L_1} + (\overline{L_2 - R_1}) + index(L_3) - \overline{R_2} \\ &= \overline{L_1} + \overline{L_2} - \overline{R_1} + index(L_3) - \overline{R_2} \\ &= n. \end{aligned}$$

b)

$$\begin{aligned} \overline{R_2} &\leq \overline{R_2} + (\overline{R_1 - L_2}) \\ &= \overline{R_2 \sqcup (R_1 - L_2)} \\ &= \overline{R_{12}} \end{aligned}$$

Then $\overline{R_2} < index(L_3)$.
Then $a \in L_{23}$ with $index(L_{23}) = \overline{L_2} + index(L_3) - \overline{R_2}$.
Then $a \in L_{1(23)}$, with

$$\begin{aligned} index(L_{1(23)}) &= \overline{L_1} + \overline{L_2} + index(L_3) - \overline{R_2} - \overline{R_1} \\ &= n. \end{aligned}$$

Symmetrical arguments apply for the proof of

**Lemma 5.** $R_{(12)3} = R_{1(23)}$.

Furthermore, Lemma 2 and Lemma 3 imply

**Lemma 6.** $inner(N_{(12)3}) = inner(N_{1(23)})$.

Proof of the Theorem is now completed by by Lemmata 3, 4 and 5.

# Towards a Standard for Modular Petri Nets: A Formalisation

Ekkart Kindler[1] and Laure Petrucci[2]

[1] Informatics and Mathematical Modelling
Technical University of Denmark
DK-2800 Lyngby, Denmark
eki@imm.dtu.dk

[2] LIPN, CNRS UMR 7030, Université Paris XIII
99, avenue Jean-Baptiste Clément
F-93430 Villetaneuse, France
petrucci@lipn.univ-paris13.fr

**Abstract.** When designing complex systems, mechanisms for structuring, composing, and reusing system components are crucial. Today, there are many approaches for equipping Petri nets with such mechanisms. In the context of defining a standard interchange format for Petri nets, modular PNML was defined as a mechanism for modules in Petri nets that is independent from a particular version of Petri nets and that can mimic many composition mechanisms by a simple import and export concept.

Due to its generality, the semantics of modular PNML was only informally defined. Moreover, modular PNML did not define which concepts could or should be subject to import and export in high-level Petri nets.

In this paper, we formalise a minimal version of modular high-level Petri nets, which is based on the concepts of modular PNML. This shows that modular PNML can be formalised once a specific version of Petri net is fixed. Moreover, we present and discuss some more advanced features of modular Petri nets that could be included in the standard. This way, we provide a formal foundation and a basis for a discussion of features to be included in the upcoming standard of a module concept for Petri nets in general and for high-level nets in particular.

**Keywords:** Modular Petri Nets, Standardisation, High-Level Nets.

## 1 Introduction

It is well-known that, in order to design large and complex systems, a mechanism to break the system down into smaller pieces is needed. Although Petri nets are often blamed for not having a structuring mechanism, there actually are many proposals for composing Petri nets and for splitting large models into smaller ones (see related work in Sect. 8). Moreover, for industrial size systems, it is not only important to have a system composed of smaller subsystems or modules; a module concept must also cater for re-use and abstraction.

The standard on High-level Petri nets, ISO/IEC 15909 Part 1 [1], however, does not define a concept for modules yet. Structuring issues and net extensions were left for the future Part 3 of the standard. In this paper, we make a proposal for a module concept for high-level Petri nets and its mathematical underpinning. Note that there are two major directions for constructing nets from some parts, which we call *composition* and *modularity*. In composition, basically, any two or more nets can be composed by one or more composition operators, which gives a new net, which, in principle, can be very different from the original nets. In modularity, we can basically instantiate different *module definitions* with clear *interfaces*; these *instances* can be connected with each other at their interfaces, but their structure is not changed. Here, we propose a modularity approach; still, this approach can mimic the main composition operators for nets like place and transition fusion by the help of *import* and *export nodes* and *symbols* in the interfaces.

The module concept which we propose here is—up to some details—the one from *modular PNML* [2,3], which was defined along with the early version of the *Petri Net Markup Language*[1] for interchanging all kinds of Petri nets. Actually, modular PNML was more ambitious and more general since it was intended to work for all kinds of Petri nets—in the terminology of PNML, for all *Petri net types*. The downside of modular PNML's generality, however, is that the semantics was defined informally and only in terms of syntactical substitutions, copies, and replacements, which was called flattening. Moreover, a concept for a proper and syntactically correct use of symbols was proposed only very recently [5]. Here, we focus on the module concepts for high-level Petri nets and provide both, some semantical concepts for a more elegant way of dealing with modular high-level Petri nets and a clear semantics.

One problem with most of the mathematical formalisations of high-level Petri nets is that the sort and operation symbols and their underlying meaning (the algebra) is monolithic. This becomes a problem when using and combining symbols from different module instances. In this paper, we solve this problem by a simple concept called *generators*.

## 2 Introductory Example

In this section, we discuss the main concepts of modular PNML with the help of an example. The example as well as its explanation are a revised version of the example from [5], which was based on examples from [2,3].

### 2.1 Module Definition

Figure 1 shows an example of a module Channel that transmits some information from a place *p1* to a place *p2*. To be more precise, Fig. 1 shows the *module definition*. It consists of two parts: The upper part in the bold-faced box defines the *interface* of the module and its name Channel. The interface consists of

[1] PNML is currently under the final ballot as an interchange format for High-level Petri nets, subject of Part 2 of the standard [4].

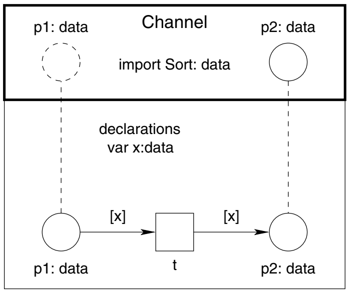


**Fig. 1.** The module Channel

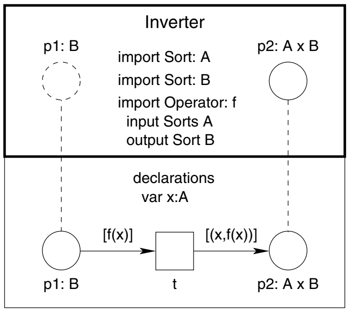


**Fig. 2.** The module Inverter

different parts: it imports a place (the dashed circle on the left-hand side) and it exports a place (the solid circle on the right-hand side). The difference between import and export nodes will become clear later in this paper. Intuitively, the import place will be provided by the environment of the module when it is used; conversely, the export place is a place that can be used by the environment of the module. In addition to the import and export nodes, the module definition also imports a sort symbol. As indicated by its name, this sort represents the type data that should be transmitted over the channel. In order to make the symbol an import symbol, we use the keyword import[2], the additional text Sort indicates that this symbol is a sort.

The lower part of the module definition in the thinly outlined box is the implementation of the module. Basically, this is a "normal" high-level Petri net. The only difference is that it uses the sort data provided via the interface, and some of its elements refer to the interface. The place on the left-hand side is associated with the import place, while the place on the right-hand side is associated with the export place, as graphically indicated by the two dashed lines.

Moreover, there is a transition between these two places; the annotations of the two arcs are [x], where x is a variable of sort data. This variable is defined in the declaration of the implementation; the declaration makes use of the imported sort symbol data. The bracket notation around variable x indicates that the arc expression denotes a multiset with a single element[3] bound to variable x.

Note that both places in the module implementation have *type* data, which exactly corresponds to the type of the import and export places in the interface.

## 2.2 Module Instances

Next, we build a simple system from the module Channel. Figure 3 shows the use of three *instances* of the module Channel. The instances are named ch1, ch2, and ch3, respectively. To indicate the instantiation, we use the name of the instance followed by the name of the module definition (inspired by UML).

[2] Actually, the keywords import and export, as well as the graphical notation for import and export nodes, are not the point of this paper.

[3] Often, the notation 1‘x is also used, e.g., for coloured nets in CPNTools [6].

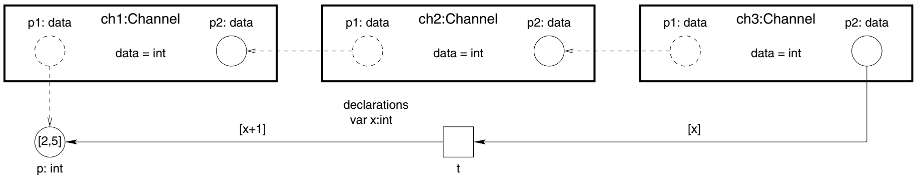


**Fig. 3.** Using instances of module Channel

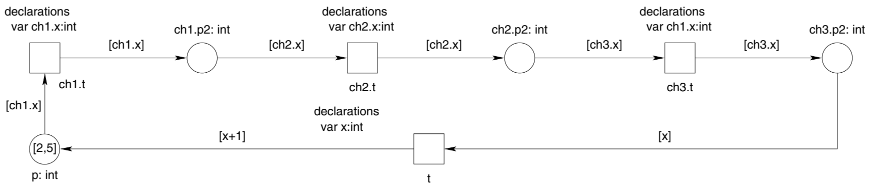


**Fig. 4.** The resulting model

With the help of this example, we can explain the meaning of import places and import symbols. For each instance of the module, the import place needs to refer to some place outside the module, which will be the one imported for that instance. This is indicated by dashed arrows from the use of the interface to some other parts of the system. Note that export nodes of module instances are seen outside a module. Hence, they can be referred to from import nodes. This way, we get a sequence of three channels. Once the data from the leftmost place is transmitted to the right-most channel, the additional transition increments the token value and sends it back to the start place.

This is where the import symbol data representing a sort comes in again. For each instance of the module Channel, we must provide a sort for the symbol data. In this example, we use the sort int, which is a built-in sort of high-level Petri nets. This way, the chain of channels transmits integer values. However, we could have used any other built-in or user-defined sort for that.

From the model in Fig. 3 and the definition of the module Channel as shown in Fig. 1, the actual Petri net defined is the one shown in Fig. 4. It is obtained by making a copy of the module implementation for each module instance and by merging the nodes identified by the references. Moreover, every occurrence of the sort data in the module implementation is now replaced by the sort it is assigned in this instance of the module: int in our example.

## 2.3 More Advanced Concepts

In the rest of this paper, we will formalise these ideas. The formal definitions will however be more general. In our example, we had only import and export places. In the general definition, there will be also import and export transitions, with basically the same mechanism: fusing the respective transitions.

Moreover, modules can also import operation symbols. Figure 2 shows an example. For some operation $\mathsf{f} : \mathsf{A} \to \mathsf{B}$ and some value $y$ of sort $\mathsf{B}$, which is put to the import place, it calculates a pair $(x, y)$ such that $\mathsf{f}(x) = y$ and puts this pair to the output place—if such a pair exists: it magically computes the inverse of $\mathsf{f}$. Note that the module is independent of a particular operator $\mathsf{f}$; it works for any operator, which will be provided when the module is instantiated.

In addition to the import of an operation symbol, this example shows another important feature. In the implementation of the module, we make use of the imported sorts $\mathsf{A}$ and $\mathsf{B}$ and build a new one, the product sort. This is, actually, one of the technically tricky issues of the formalisation.

Finally, our formalisation allows for exporting sort and operation symbols.

## 3 Basic Definitions

In this section, we formalise algebraic Petri nets and all the pre-requisites. We introduce the standard concepts of algebraic specifications [7] and of algebraic Petri nets [8,9,10,11,1], but in a notation easing the definition of modules.

### 3.1 Basic Notations

As usual, $\mathbb{N}$ stands for the set of *natural numbers* (including 0), and $\mathbb{B}$ stands for the set of *booleans*, i.e., $\mathbb{B} = \{\mathit{false}, \mathit{true}\}$. For some set $A$, $A^+$ denotes the set of all non-empty finite sequences over $A$. For some function $f : A \to B$ and some set $C$, the restriction of $f$ to $C$ is defined as the function $f|_C : A \cap C \to B$ with $f|_C(a) = f(a)$ for all $a \in A \cap C$. For two functions $f : A \to B$ and $g : C \to D$ with disjoint domains $A$ and $C$, we define $f \cup g$ as the function $(f \cup g) : A \cup C \to B \cup D$ with $(f \cup g)(a) = f(a)$ for all $a \in A$ and $(f \cup g)(c) = g(c)$ for all $c \in C$.

For some set $I$, a set $A$ together with a mapping $i : A \to I$ is an *$I$-indexed set* $(A, i)$. The $I$-indexed set $(A, i)$ is *finite* if $A$ is finite. When $i$ is understood from the context, we often use $A$ for denoting the $I$-indexed set. For every $j \in I$, we define the set of all elements indexed by $j$: $A_j = \{a \in A \mid i(a) = j\}$. By definition, all $A_j$ are disjoint. For an $I$-indexed set $(A, i)$ and some set $B$, we define $(A, i) \cap B = (A \cap B, i|_B)$.

For some set $A$, a mapping $m : A \to \mathbb{N}$ is called a multiset over $A$ if $\sum_{a \in A} m(a)$ is finite. The set of all multisets over $A$ is denoted by $\mathit{MS}(A)$.

### 3.2 Signatures and Algebras

The idea of high-level nets is that there are different kinds of tokens, which are often called colours. Mathematically, the tokens can come from some set which is associated with a place. Different functions allow for manipulating them. In order to represent these sets and functions, some syntax must be introduced. Here, we use the approach of algebraic nets, where we use signatures for the syntax, and the associated algebras for their underlying meaning.

**Definition 1 (Signature).** *A* signature *$SIG = (S, O)$ consists of a set of* sort symbols *$S$ (often called sorts for short) and an $S^+$-indexed set of* operation symbols *$O$. The set $S \cup O$ is called the set of* symbols *of $SIG$. For some signature $SIG$, we denote the set of its sorts by $S^{SIG}$ and the set of its operations by $O^{SIG}$.*

For some set of symbols $A$ and some signature $SIG = (S, O)$ the *restriction* of $SIG$ to symbols in $A$ is $SIG|_A = (S \cap A, O \cap A)$. Note that $SIG|_A$ is not always a signature (e. g. if $A$ contains an operation of $O$ but not the operation sorts).

**Definition 2 (Signature extension).** *A signature $SIG'$* extends a signature *$SIG$, if for some set $A : SIG'|_A = SIG$. This is denoted by $SIG \subseteq SIG'$. Let $SIG = (S, O)$ and $SIG' = (S', O')$ be two signatures with a disjoint set of symbols, then we define the union $SIG \cup SIG' = (S \cup S', O \cup O')$.*

By definition, $SIG \cup SIG'$ is a signature, which extends $SIG$ and $SIG'$.

**Definition 3 (Signature homomorphism).** *For two signatures $SIG = (S, O)$ and $SIG' = (S', O')$, a mapping $\sigma : S \cup O \to S' \cup O'$ is called a* signature homomorphism*, if for every $s \in S$ we have $\sigma(s) \in S'$ and for every $o \in O_{s_1 \ldots s_n}$ we have $\sigma(o) \in O'_{\sigma(s_1)\ldots\sigma(s_n)}$.*

**Definition 4 (Algebra).** *A $SIG$-*algebra *$\mathcal{A}$ assigns a* carrier set *to every sort of $SIG$ and a* function *to every operation of $SIG$.*

Technically, $\mathcal{A}$ is a mapping such that, for every $s \in S$, $\mathcal{A}(s)$ is a set and, for every $o \in O_{s_1 \ldots s_n s_{n+1}}$, $\mathcal{A}(o)$ is a function with $\mathcal{A}(o) : \mathcal{A}(s_1) \times \ldots \times \mathcal{A}(s_n) \to \mathcal{A}(s_{n+1})$.

**Definition 5 (Algebra extension).** *Let $SIG$ and $SIG'$ be two signatures with $SIG \subseteq SIG'$, and let $\mathcal{A}$ be a $SIG$-algebra and $\mathcal{A}'$ be a $SIG'$-algebra. Algebra $\mathcal{A}'$* extends *algebra $\mathcal{A}$, if $\mathcal{A}'|_{S^{SIG} \cup O^{SIG}} = \mathcal{A}$, written $\mathcal{A} \subseteq \mathcal{A}'$.*

## 3.3 Variables and Terms

Let $SIG = (S, O)$ be a signature. An $S$-indexed set $X$ is a set of *$SIG$-variables*, if $X$ is disjoint from $O$. From the set of operations $O$ of the signature and of variables $X$, *terms* of some sort $s$ can be constructed inductively:

**Definition 6 (Terms).** *The sets of all $SIG$-*terms *of sort $s$ over a set of variables $X$ is denoted by $\mathbb{T}_s^{SIG}(X)$. It is inductively defined as follows:*

- *$X_s \subseteq \mathbb{T}_s^{SIG}(X)$.*
- *For every operation symbol $o \in O_{s_1 \ldots s_n s_{n+1}}$, and, for every $k$ with $1 \leq k \leq n$, $t_k \in \mathbb{T}_{s_k}^{SIG}(X)$, we have $(o, t_1, \ldots, t_n) \in \mathbb{T}_{s_{n+1}}^{SIG}(X)$.*

When $SIG$ is clear from the context, we also write $\mathbb{T}_s(X)$ instead of $\mathbb{T}_s^{SIG}(X)$. The set of all terms is $\mathbb{T}^{SIG}(X) = \bigcup_{s \in S} \mathbb{T}_s^{SIG}(X)$. Terms without variables are called *ground terms* and are defined by $\mathbb{T}^{SIG} = \mathbb{T}(\emptyset)$ and by $\mathbb{T}_s^{SIG} = \mathbb{T}_s^{SIG}(\emptyset)$.

Note that, in practice, terms are written $o(t_1, \ldots, t_n)$ to make clear that the operation is applied to the arguments. In order to emphasise the syntactical nature of terms, we use the tuple notation $(o, t_1, \ldots, t_n)$ in all formal definitions.

**Definition 7 (Compatible mapping).** *Let SIG and SIG′ be two signatures and $\sigma$ a signature homomorphism from SIG to SIG′. Let $X$ be a set of SIG-variables and $X'$ be a set of SIG′-variables. A mapping $\xi : X \to X'$ is said to be* compatible *with $\sigma$ if, for every variable $x \in X_s$, we have $\xi(x) \in X'_{\sigma(s)}$. The mappings $\sigma$ and $\xi$ can be canonically extended to a mapping $\overline{\sigma \cup \xi} : \mathbb{T}^{SIG}(X) \to \mathbb{T}^{SIG'}(X')$ by*

– $\overline{\sigma \cup \xi}(x) = \xi(x)$ *for every $x \in X$, and*
– $\overline{\sigma \cup \xi}((o, t_1, \ldots, t_n)) = (\sigma(o), \overline{\sigma \cup \xi}(t_1), \ldots, \overline{\sigma \cup \xi}(t_n))$, *for every operation symbol $o \in O_{s_1 \ldots s_n s_{n+1}}$, and, for all terms $t_k \in \mathbb{T}^{SIG}_{s_i}(X)$.*

## 3.4 Generators

In high-level nets and high-level net modules in particular, we often have some sorts provided, and we need to construct other sorts from them in a standard way. For example, for a given sort $s$, we need a sort that represents the multiset sorts over that sort, $ms(s)$. We may also want to build the product sort over some sorts (see Fig. 2 for an example). Moreover, the sets associated with these new sorts are defined based on the sets associated with the underlying sorts. For example, the set associated with $ms(s)$ is the set of all multisets over $\mathcal{A}(s)$.

In module definitions, we also want to import sorts to be used in the module implementation without yet knowing which concrete set will be associated with it, since this will only be known when the module is instantiated. Still, we would like to use these sorts and sorts built from them in the module definition. For that purpose, we need a mechanism for constructing new sorts and operations from some signature and a way to define their meaning. To this end, we introduce *generators*. A generator defines which new sorts and operators can be constructed out of existing sorts, and once the associated sets are known for every sort, what the meaning of the corresponding constructed sorts and operators should be. Since generators are needed anyway, we also use them for defining the standard sorts, such as *bool*, along with their operations.

**Definition 8 (Generator).** *A* generator $G = (GS, GA)$ *consists of*

– *a* sort generator function $GS$ *that, for any given signature $SIG = (S, O)$, returns a signature $GS(SIG) = (S', O')$ such that $S \subseteq S'$ and $O \subseteq O'$; $GS(SIG)$ is called the signature generated from SIG by the generator G;*
– *an* algebra generator function $GA$ *that, for any SIG-algebra $\mathcal{A}$, returns a $GS(SIG)$-algebra such that the algebra $GA(\mathcal{A})$ extends algebra $\mathcal{A}$, i.e., $\mathcal{A} \subseteq GA(\mathcal{A})$.*

Throughout this paper, we will use a single generator[4] $G = (GS, GA)$, which will be defined in this section. The basic idea is to include, in addition to the

[4] This is a very minimalistic version; there could be many more built-in sorts, generated sorts, and operations (see ISO/IEC 15909-2 [4]); but this is beyond the scope of this paper; our module concept will work for any generator extending this one.

existing sorts, the booleans, the associated multiset sort $ms(s)$ for every sort $s$, and all the product sorts. In order to emphasise the syntactical nature, and to distinguish the newly constructed sorts, we use the notation $(\mathit{bool})$, $(ms, s)$ and $(\times, s_1, \ldots, s_n)$ for these generated sorts. Likewise, the generator will generate the boolean constants $(\mathit{true})$ and $(\mathit{false})$ and the standard operations on booleans, the tupling operation $((), s_1, \ldots, s_n)$, and the operation $(([], s), s, \ldots, s)$ which makes a multiset out of a list of elements.

**Definition 9 (Sort generator).** *Let $SIG = (S, O)$ be an arbitrary signature, then $GS(SIG) = (S', O')$ is defined as follows:*

- *$S'$ is the least set for which the following conditions hold:*
  1. *$S \subseteq S'$,*
  2. *$(\mathit{bool}) \in S'$,*
  3. *$(ms, s) \in S'$ for every $s \in S'$, and*
  4. *$(\times, s_1, \ldots, s_n) \in S'$ for all sorts $s_1, \ldots, s_n \in S'$.*
- *$O'$ is the least $S'$-indexed set for which the following conditions hold:*
  1. *$O \subseteq O'$,*
  2. *$(\mathit{true}), (\mathit{false}) \in O'_{(bool)}$,*
  3. *$(\mathit{not}, (\mathit{bool})) \in O'_{(bool)(bool)}$,*
  4. *$(\mathit{and}, (\mathit{bool}), (\mathit{bool})), (\mathit{or}, (\mathit{bool}), (\mathit{bool})) \in O'_{(bool)(bool)(bool)}$,*
  5. *$(([], s), s, \ldots, s) \in O'_{s\ldots s(ms,s)}$ for every sort $s \in S'$, where the number of $s$ is the same in both constructs,*
  6. *$(+, (ms, s), (ms, s)) \in O'_{(ms,s)(ms,s)(ms,s)}$ for every $s \in S'$, and*
  7. *$((), s_1, \ldots, s_n) \in O'_{s_1\ldots s_n(\times,s_1,\ldots,s_n)}$ for all $s_1, \ldots, s_n \in S'$.*

**Definition 10 (Algebra generator).** *Let $\mathcal{A}$ be a SIG-algebra with $SIG = (S, O)$ and let $GS(SIG) = (S', O')$. Then we define $GA(\mathcal{A})$ by:*

- *The mapping of the sorts of $GA(\mathcal{A})$ is defined as follows:*
  1. *$GA(\mathcal{A})|_S = \mathcal{A}|_S$,*
  2. *$GA(\mathcal{A})((\mathit{bool})) = \mathbb{B}$,*
  3. *$GA(\mathcal{A})((ms, s)) = MS(GA(\mathcal{A})(s))$ for every sort $s \in S'$, and*
  4. *$GA(\mathcal{A})((\times, s_1, \ldots, s_n)) = GA(\mathcal{A})(s_1) \times \ldots \times GA(\mathcal{A})(s_n)$ for all sorts $s_1, \ldots, s_n \in S'$.*
- *The mapping of the operations of $GA(\mathcal{A})$ is defined as follows:*
  1. *$GA(\mathcal{A})|_O = \mathcal{A}|_O$,*
  2. *$GA(\mathcal{A})((\mathit{true})) = \mathit{true}$ and $GA(\mathcal{A})((\mathit{false})) = \mathit{false}$,*
  3. *$GA(\mathcal{A})((\mathit{not}, \mathit{bool})) = \neg$, where $\neg$ is the boolean negation function,*
  4. *$GA(\mathcal{A})((\mathit{and}, \mathit{bool}, \mathit{bool})) = \wedge$ and $GA(\mathcal{A})((\mathit{or}, \mathit{bool}, \mathit{bool})) = \vee$, where $\wedge$ and $\vee$ are the boolean conjunction and disjunction functions,*
  5. *$GA(\mathcal{A})((([], s), s, \ldots, s))(a_1, \ldots, a_n) = [a_1, \ldots, a_n]$, for every sort $s \in S'$ and all $a_1, \ldots, a_n \in GA(\mathcal{A})(s)$; i. e. the multiset over $s$ containing exactly the elements $a_1, \ldots, a_n$,*
  6. *$GA(\mathcal{A})(+, (ms, s), (ms, s))) = +$ for every sort $s \in S'$, where $+$ denotes the addition of two multisets over $GA(\mathcal{A})(s)$, and*

7. *$GA(\mathcal{A})(((), s_1, \ldots, s_n))(a_1, \ldots, a_n) = (a_1, \ldots, a_n)$ for all $s_1, \ldots, s_n \in S'$ and $a_1 \in GA(\mathcal{A})(s_1), \ldots, a_n \in GA(\mathcal{A})(s_n)$, i.e., the usual tupling.*

Note that, to avoid overly complex mathematics, we assume that all the symbols used in a basic signature *SIG* are disjoint from symbols introduced by the generators $GS(\mathit{SIG})$. We assume that the symbols in *SIG* are flat and unstructured, whereas the symbols introduced in $GS(\mathit{SIG})$ are tuples—some of them, like $(\mathit{bool})$, are 1-tuples. Since this is just needed for making the mathematics work, our examples will use *bool* for $(\mathit{bool})$ and $ms(s)$ for $(ms, s)$. However, we stick to the technical notations $(\mathit{bool})$ and $(ms, s)$ in all formal definitions.

**Definition 11 (Sort generator homomorphism).** *A signature homomorphism $\sigma$ from some signature SIG to some signature SIG$'$ carries over to a signature homomorphism $\sigma^G$ from $GS(\mathit{SIG})$ to $GS(\mathit{SIG}')$ in a canonical way:*

- 1. *$\sigma^G(s) = \sigma(s)$ for every $s \in S$,*
  2. *$\sigma^G((\mathit{bool})) = (\mathit{bool})$,*
  3. *$\sigma^G((ms, s)) = (ms, \sigma^G(s))$ for every $s \in S$, and*
  4. *$\sigma^G((\times, s_1, \ldots, s_n)) = (\times, \sigma^G(s_1), \ldots, \sigma^G(s_n))$ for all $s_1, \ldots, s_n \in S$.*
- 1. *$\sigma^G(o) = \sigma(o)$ for every operation $o \in O$,*
  2. *$\sigma^G((\mathit{true})) = (\mathit{true})$ and $\sigma^G((\mathit{false})) = (\mathit{false})$,*
  3. *$\sigma^G((\mathit{not}, (\mathit{bool}))) = (\mathit{not}, (\mathit{bool}))$,*
  4. *$\sigma^G((\mathit{and}, (\mathit{bool}), (\mathit{bool}))) = (\mathit{and}, (\mathit{bool}), (\mathit{bool}))$, and $\sigma^G((\mathit{or}, (\mathit{bool}), (\mathit{bool}))) = (\mathit{or}, (\mathit{bool}), (\mathit{bool}))$,*
  5. *$\sigma^G(([], s, \ldots, s)) = (([], \sigma^G(s)), \sigma^G(s), \ldots, \sigma^G(s))$ for every sort $s \in S$,*
  6. *$\sigma^G((+, (ms, s), (ms, s))) = (+, (ms, \sigma^G(s)), (ms, \sigma^G(s)))$ for every sort $s \in S$, and*
  7. *$\sigma^G(((), s_1, \ldots, s_n)) = ((), \sigma^G(s_1), \ldots, \sigma^G(s_n))$ for all $s_1, \ldots, s_n \in S$.*

In the following, we even use the symbol $\sigma$ instead of $\sigma^G$.

## 3.5 Nets, Algebraic Net Schemes, and Algebraic Nets

Now we are prepared to define the basic concepts of this paper.

**Definition 12 (Net).** *A* net *$N = (P, T, F)$ consists of two disjoint sets $P$ and $T$ and a set of arcs $F \subseteq (P \times T) \cup (T \times P)$.*

For a clear separation between syntax and semantics, we distinguish between *algebraic net schemes* and *algebraic nets.*

**Definition 13 (Algebraic net scheme).** *An* algebraic net scheme *is a tuple $\Sigma = (N, \mathit{SIG}, X, \mathit{sort}, l, c, m)$ consisting of:*

1. *a* net *$N = (P, T, F)$,*
2. *a* signature *SIG,*
3. *a set of $GS(\mathit{SIG})$-*variables *$X$,*
4. *a* place sort *mapping $\mathit{sort} : P \to S^{GS(\mathit{SIG})}$,*

5. *an* arc label *mapping* $l : F \rightarrow \mathbb{T}^{GS(SIG)}(X)$ *such that:*
   – *for all* $(p,t) \in F \cap (P \times T) : l((p,t)) \in \mathbb{T}^{GS(SIG)}_{(ms,sort(p))}(X)$
   – *for all* $(t,p) \in F \cap (T \times P) : l((t,p)) \in \mathbb{T}^{GS(SIG)}_{(ms,sort(p))}(X)$,
6. *a* transition condition *mapping* $c : T \rightarrow \mathbb{T}^{GS(SIG)}_{(bool)}(X)$,
7. *an* initial marking $m : P \rightarrow \mathbb{T}^{GS(SIG)}$ *such that,* $m(p) \in \mathbb{T}^{GS(SIG)}_{(ms,sort(p))}$ *for every place* $p \in P$.

**Definition 14 (Algebraic net).** *An* algebraic net $(\Sigma, \mathcal{A})$ *is an algebraic net scheme* $\Sigma$ *equipped with a* $SIG$*-algebra* $\mathcal{A}$.

In this paper, we focus on the definition of modules and how they can be used to define other modules. We are not so much interested in their actual behaviour. Therefore, we do not define the firing rule for algebraic nets here. A formalisation of the abstraction in terms of the behaviour of a module is an interesting endeavour—but much beyond the scope of this paper.

## 4 Modules Interfaces and Implementation

In this section, we formalise the notion of module interfaces and their implementation, informally introduced in Sect. 2. The module interface describes which places and transitions are imported or exported, and which sort and operation symbols are imported or exported on a purely syntactical level. Moreover, the interface defines the sort of each of the import and export places.

**Definition 15 (Module interface)**
*A* module interface $\mathcal{I} = ((SIG_I, P_I, T_I), (SIG_O, P_O, T_O), sort_{IO})$ *consists of two signatures* $SIG_I$ *and* $SIG_O$ *with disjoint sets of symbols, four pairwise disjoint sets* $P_I$, $T_I$, $P_O$, $T_O$ *and a mapping* $sort_{IO} : P_I \cup P_O \rightarrow S^{GS(SIG_I \cup SIG_O)}$.

We call $(SIG_I, P_I, T_I)$ the *import interface*, $SIG_I$ the *imported signature*, $P_I$ the *imported places*, and $T_I$ the *imported transitions*. We call $(SIG_O, P_O, T_O)$ the *export interface*, $SIG_O$ the *exported signature*, $P_O$ the *exported places*, and $T_O$ the *exported transitions*. The mapping $sort_{IO}$ assigns a sort to every place of the interface. Note that this can be any sort that can be generated from the sorts of the import and export signatures.

**Definition 16 (Module implementation)**
*Let* $\mathcal{I} = ((SIG_I, P_I, T_I), (SIG_O, P_O, T_O), sort_{IO})$ *be a module interface. Then, a* module implementation $\mathcal{M} = (\mathcal{I}, \Sigma, \mathcal{A})$ *of interface* $\mathcal{I}$*, consists of:*

1. *the* interface $\mathcal{I}$ *itself,*
2. *an* algebraic net scheme $\Sigma = (N, SIG, X, sort, l, c, m)$ *with* $N = (P, T, F)$ *where* $SIG = (S, O)$ *extends* $SIG_I$ *and* $SIG_O$ *such that* $SIG$ *restricted to the non-imported part,* $SIG \setminus SIG_I = SIG|_{S \setminus S^{SIG_I} \cup O \setminus O^{SIG_I}}$, *is a signature,* $P \supseteq P_I \cup P_O$, $T \supseteq T_I \cup T_O$, *and* $sort \supseteq sort_{IO}$,
3. *a* $SIG \setminus SIG_I$-algebra $\mathcal{A}$.

Note that this definition does not require that $\mathcal{A}$ is a *SIG*-algebra since some symbols from *SIG* are imported from $SIG_I$. The interpretation of the symbols from $SIG_I$ will come from the imported symbols when the module is instantiated. In order to assign the meaning to the remaining symbols, we require $SIG \setminus SIG_I$ to be a signature, and $\mathcal{A}$ to be a $SIG \setminus SIG_I$-algebra. In general, $(\Sigma, \mathcal{A})$ is not an algebraic net; however, it is an algebraic net if $SIG_I$ is empty.

## 5 Modules Definitions and Implementations

Up to now, we have defined module interfaces and their implementation. The implementation was given by a monolithic algebraic net. The purpose of modules, however, is to use instances of some modules for defining a system or other modules, which we call *module definitions*. In this section, the notion of module definition as well as its meaning is formalised.

### 5.1 Module Definition

Let $\mathcal{J}$ be a set of module interfaces, which can then be used for defining another module. For each $\mathcal{I} \in \mathcal{J}$, we denote: $\mathcal{I} = ((SIG_I^{\mathcal{I}}, P_I^{\mathcal{I}}, T_I^{\mathcal{I}}), (SIG_O^{\mathcal{I}}, P_O^{\mathcal{I}}, T_O^{\mathcal{I}}), sort_{IO}^{\mathcal{I}})$, $SIG_I^{\mathcal{I}} = (S_I^{\mathcal{I}}, O_I^{\mathcal{I}})$ and $SIG_O^{\mathcal{I}} = (S_O^{\mathcal{I}}, O_O^{\mathcal{I}})$.

First, we introduce a notation for *module instances* resp. the *use* of modules.

**Definition 17 (Module instances and uses).** *For some $n \in \mathbb{N}$ and for every $k \in \{1, \ldots, n\}$, let $\mathcal{I}_k \in \mathcal{J}$. Then, $\mathcal{U} = \{(1, \mathcal{I}_1), \ldots, (n, \mathcal{I}_n)\}$ is a set of $n$* module instances *of $\mathcal{J}$. The set $\mathcal{U}$ is called the* module uses*.*

Note that the interfaces $\mathcal{I}_k$ are not required to be different since the same module may be used multiple times in another module definition. Therefore, in order to be able to distinguish different instances of the same module, a different number is associated with each of them. This is the case in the introductory example of Sect. 2, where three copies of the Channel module are used.

**Definition 18 (Module definition).** *A* module definition $\mathcal{D} = (\mathcal{I}, \Sigma, \mathcal{U}, (si_k)_{k=1}^n, (pi_k)_{k=1}^n, (ti_k)_{k=1}^n, (so_k)_{k=1}^n, (po_k)_{k=1}^n, (to_k)_{k=1}^n, \mathcal{A})$ *for interface $\mathcal{I}$ over some module interfaces $\mathcal{J}$, consists of the following:*

1. *its own* interface $\mathcal{I} = ((SIG_I, P_I, T_I), (SIG_O, P_O, T_O), sort_{IO})$,
2. *an* algebraic net scheme $\Sigma = (N, SIG, X, sort, l, c, m)$ *with signature* $SIG = (S, O)$ *and net* $N = (P, T, F)$,
3. *a set of* module instances $\mathcal{U} = \{(1, \mathcal{I}_1), \ldots, (n, \mathcal{I}_n)\}$ *of* $\mathcal{J}$,
4. *for each* $k \in \{1, \ldots, n\}$,
   (a) *a signature homomorphism* $si_k : SIG_I^{\mathcal{I}_k} \to SIG$,
   (b) *an injective signature homomorphism* $so_k : SIG_O^{\mathcal{I}_k} \to SIG \setminus SIG_I$,
   (c) *a mapping* $pi_k : P_I^{\mathcal{I}_k} \to P$,
   (d) *an injective mapping* $po_k : P_O^{\mathcal{I}_k} \to P \setminus P_I$,
   (e) *a mapping* $ti_k : T_I^{\mathcal{I}_k} \to T$,

*(f) an injective mapping* $to_k : T_O^{\mathcal{I}_k} \to T \setminus T_I$,
*such that the co-domains of all homomorphisms* $so_k$ *are pairwise disjoint, the co-domains of all mappings* $po_k$ *are pairwise disjoint, and the co-domains of all mappings* $to_k$ *are pairwise disjoint. Moreover, for every* $k$ *and for every* $p \in P_I^{\mathcal{I}_k}$*, we have* $si_k(sort_{IO}^{\mathcal{I}_k}(p)) = sort(pi_k(p))$*, and for every* $p \in P_O^{\mathcal{I}_k}$*, we have* $so_k(sort_{IO}^{\mathcal{I}_k}(p)) = sort(po_k(p))$*, such that* $SIG_D = (S', O')$ *with* $S' = S \setminus (S^{SIG_I} \cup \bigcup_{k=1}^{n} so_k(S^{SIG_O^{\mathcal{I}_k}}))$ *and* $O' = O \setminus (O^{SIG_I} \cup \bigcup_{k=1}^{n} so_k(O^{SIG_O^{\mathcal{I}_k}}))$ *is a signature, and*

5. $\mathcal{A}$ *is a* $SIG_D$*-algebra.*

Basically, the module definition consists of an algebraic net scheme $\Sigma$, where the homomorphisms and mappings (see condition 4) from the interfaces of the used module instances $\mathcal{U}$ to $\Sigma$ indicate how the instances of the modules are embedded into $\Sigma$. This concerns the embedding of places and transitions as well as the use of the different symbols of the signatures. Note that if an export symbol of a module instance is mapped to a symbol of $\Sigma$, this symbol will get its meaning from this module instance. Therefore, condition 4 requires that these mappings do not overlap. The meaning of the symbols of the import signature will be defined when the module is used; therefore, the module definition itself does not need to give a definition to these symbols. Therefore, the algebra $\mathcal{A}$ does not need to assign a meaning for the symbols coming from the import signature of the defined module or from the export symbols of the used modules. The remaining part of the signature, denoted by $SIG_D$ in the above definition, must be a signature and $\mathcal{A}$ must be a $SIG_D$-algebra (condition 5).

## 5.2 Denoted Implementation

In this section, we will define the module implementation inferred from a module definition, i.e., based on other modules. Let us consider a module definition

$$\mathcal{D} = (\mathcal{I}, \Sigma, \mathcal{U}, (si_k)_{k=1}^n, (pi_k)_{k=1}^n, (ti_k)_{k=1}^n, (so_k)_{k=1}^n, (po_k)_{k=1}^n, (to_k)_{k=1}^n, \mathcal{A})$$

as defined in Def. 18 using module instances $\mathcal{U} = \{(1, \mathcal{I}_1), \ldots, (n, \mathcal{I}_n)\}$. In order to define the module implementation, the implementations of the used modules must be known. Let us assume that for each $k \in \{1, \ldots, n\}$, $\mathcal{M}_k = (\mathcal{I}_k, \Sigma_k, \mathcal{A}_k)$ is an implementation for interface $\mathcal{I}_k$. The basic idea of the module defined by $\mathcal{D}$ is to make a disjoint union of all signatures and nets of the implementations and the module definition itself, and to transform the arc, place, and transition labels accordingly. However, some parts need to be identified, as defined by the homomorphism between the signatures and the mappings from the interface places and transitions to the places and transitions of the module definition.

We start with defining the signature of the module implementation, which will be denoted with $\widehat{SIG}$. First, we summarise the available signatures:

1. The signature $SIG = (S, O)$ from the module definition.
2. For every use of a module $(k, \mathcal{I}_k)$, there are two disjoint signatures $SIG_I^{\mathcal{I}_k}$ and $SIG_O^{\mathcal{I}_k}$. We define $SIG^{\mathcal{I}_k} = SIG_I^{\mathcal{I}_k} \cup SIG_O^{\mathcal{I}_k}$. Moreover, there is a signature homomorphism $f_k : SIG^{\mathcal{I}_k} \to SIG$ defined by $f_k = so_k \cup si_k$.

3. For every use of a module $(k, \mathcal{I}_k)$, the implementation $\mathcal{M}_k = (\mathcal{I}_k, \Sigma_k, \mathcal{A}_k)$ has a signature $SIG_k$ and variables $X_k$. By definition $SIG^{\mathcal{I}_k} \subseteq SIG_k$ holds.

**Definition 19 (Signature $\widehat{SIG}$ and homomorphisms $\sigma_k$)**
*For each $k \in \{1, \ldots, n\}$, the mapping $\sigma_k$ is a signature homomorphism from $SIG_k$ to $\widehat{SIG}$, defined as follows: for every $x \in SIG_k$ for which $f_k(x)$ is defined: $\sigma_k(x) = f_k(x)$; for every other symbol $x \in SIG_k$ : $\sigma_k(x) = (k, x)$. We define $\widehat{SIG} = (\widehat{S}, \widehat{O})$ by $\widehat{S} = S \cup \bigcup_{k=1}^{n} \sigma_k(S^{SIG_k})$ and $\widehat{O} = O \cup \bigcup_{k=1}^{n} \sigma_k(O^{SIG_k})$, where the arities carry over by interpreting $\sigma_k$ as a signature homomorphism.*

Note that, in the definition of $\sigma_k(x)$, the pair $(k, x)$ is used to make this symbol of $SIG_k$ different from all the other symbols. The signature $\widehat{SIG}$ will be the signature of the defined module implementation. The signature homomorphisms $\sigma_k$ relate the signatures of the implementations to $\widehat{SIG}$; they will be used to transfer the labels from the different module implementations to the defined module implementation.

The variables from the different modules are made disjoint in the same way.

**Definition 20 (Variables $\widehat{X}$).** *For every $k \in \{1, \ldots, n\}$, the mapping $\xi_k$ is defined by: $\xi_k(x) = (k, x)$ for every variable $x \in X_k$. The set of all variables is defined by $\widehat{X} = X \cup \bigcup_{k=1}^{n} \xi_k(X_k)$.*

The meaning of the non-imported symbols of $\widehat{SIG}$ is defined by an $\widehat{SIG} \setminus SIG_I$-algebra $\widehat{\mathcal{A}}$. Basically, this meaning carries over from the other algebras via the respective homomorphisms.

**Definition 21 (Algebra $\widehat{\mathcal{A}}$).** *The $\widehat{SIG} \setminus SIG_I$-algebra $\widehat{\mathcal{A}}$ associated with $\mathcal{A}$ is defined as follows: If $\mathcal{A}(x)$ is defined, then $\widehat{\mathcal{A}}(x) = \mathcal{A}(x)$; if $\mathcal{A}_k(x)$ is defined, then $\widehat{\mathcal{A}}(\sigma_k(x)) = \mathcal{A}_k(x)$.*

By the conditions imposed on the algebras and the signature homomorphisms, this definition of $\widehat{\mathcal{A}}$ is unique and it is a $\widehat{SIG} \setminus SIG_I$-algebra.

For experts, $\widehat{SIG}$ and $\widehat{\mathcal{A}}$ are pushout constructions in an appropriate category; but the categorical constructions are beyond the scope of this paper.

Next, we define the places and transitions of the module implementation, which are basically a disjoint union of all the places and transitions of the used module implementations and the places and transitions of the module definition itself. The places and transitions identified by the mappings from the import and export interfaces will be merged. First, we summarise what we already know:

1. The net $N = (P, T, F)$ of the module definition. The set of all nodes of that net is $Z = P \cup T$.
2. For every use of a module $(k, \mathcal{I}_k)$, let $Z^{\mathcal{I}_k}$ be the set of nodes of the interface and let $Z_k$ be the set of nodes of the implementation, $Z^{\mathcal{I}_k} \subseteq Z_k$. There is a mapping $g_k : Z^{\mathcal{I}_k} \to Z$, which is defined by $g_k = pi_k \cup po_k \cup ti_k \cup to_k$.

**Definition 22 (Places $\widehat{P}$ and transitions $\widehat{T}$).** *For every $k \in \{1, \ldots n\}$, a mapping $e_k$ is defined as follows: $e_k(x) = g_k(x)$ for every $x \in Z^{\mathcal{I}_k}$, and $e_k(x) =$*

*$(k, x)$ for every $x \in Z_k \setminus Z^{\mathcal{I}_k}$. The set of places of the module implementation defined by the module definition is defined by $\widehat{P} = P \cup \bigcup_{k=1}^{n} e_k(P_k)$ and the set of transitions is defined by $\widehat{T} = T \cup \bigcup_{k=1}^{n} e_k(T_k)$.*

Now we have all the ingredients for defining the module implementation. Basically, the mappings of the module instances carry over from the module implementations via the homomorphism:

**Definition 23 (Defined module implementation)**
*Let $\mathcal{D} = (\mathcal{I}, \Sigma, \mathcal{U}, (si_k)_{k=1}^{n}, (pi_k)_{k=1}^{n}, (ti_k)_{k=1}^{n}, (so_k)_{k=1}^{n}, (po_k)_{k=1}^{n}, (to_k)_{k=1}^{n}, \mathcal{A})$ be a module definition with module uses $\mathcal{U} = \{(1, \mathcal{I}_1), \ldots, (n, \mathcal{I}_n)\}$ and module implementations $\mathcal{M}_k = (\mathcal{I}_k, \Sigma_k, \mathcal{A}_k)$ for each $k$.*

*The module implementation defined by $\mathcal{D}$ is $\widehat{\mathcal{M}} = (\mathcal{I}, \widehat{\Sigma}, \widehat{\mathcal{A}})$ where $\widehat{\Sigma} = (\widehat{N}, \widehat{SIG}, \widehat{X}, \widehat{sort}, \widehat{l}, \widehat{c}, \widehat{m})$ with $\widehat{N} = (\widehat{P}, \widehat{T}, \widehat{F})$ such that*
*$\widehat{F} = F \cup \bigcup_{k=1}^{n} \{(e_k(x), e_k(y)) \mid (x, y) \in A_k\}$.*

*The mappings $\widehat{l}$, $\widehat{sort}$, $\widehat{c}$, and $\widehat{m}$ are defined as follows:*

– *$\widehat{l}(f) = l(f)$ for every arc $f \in F$ and $\widehat{l}(f) = \overline{\sigma_k \cup \xi_k}(l_k(f))$ for every arc $f \in F_k$.*
– *$\widehat{sort}(p) = sort(p)$ for every place $p \in P$ and $\widehat{sort}(e_k(p)) = \sigma_k(sort_k(p))$ for every place $p \in P_k$.*
– *for every transition $t \in T$, for which there exists no $k$ with $t \in e_k(T_O^k)$, we define $\widehat{c}(t) = c(t)$; for every transition $t \in T^k \setminus T_I^k$ we define $\widehat{c}(e_k(t)) = \overline{\sigma_k \cup \xi_k}(c_k(t))$.*
– *for every place $p \in P$, for which there exists no $k$ with $p \in e_k(P_O^k)$, we define $\widehat{m}(p) = m(p)$; for every place $p \in P^k \setminus P_I^k$ we define $\widehat{m}(e_k(p)) = \overline{\sigma_k \cup \xi_k}(m_k(p))$.*

As mentioned earlier, $\widehat{\mathcal{A}}$ is a $\widehat{SIG} \setminus SIG_I$-algebra. By the conditions imposed on the module definitions, $\widehat{l}$, $\widehat{sort}$, $\widehat{c}$, and $\widehat{m}$ are properly defined. Altogether, the defined module implementation is uniquely defined:

**Theorem 1.** *For an interface $\mathcal{I} = ((SIG_I, P_I, T_I), (SIG_O, P_O, T_O), sort_{IO}^{\mathcal{I}})$ and a module definition $\mathcal{D}$ for $\mathcal{I}$ with module uses $\mathcal{U} = \{(1, \mathcal{I}_1), \ldots, (n, \mathcal{I}_n)\}$ and module implementations $\mathcal{M}_k$, $\widehat{\mathcal{M}} = (\mathcal{I}, \widehat{\Sigma}, \widehat{\mathcal{A}})$ is a uniquely defined module implementation. If $SIG_I$ is empty, then $(\widehat{\Sigma}, \widehat{\mathcal{A}})$ is an algebraic net.*

## 6 Example

Here, we present an example of a railway case study described in a modular way in [12]. It is now slightly changed so as to be consistent with our notations.

The example models a toy railway composed of several track sections, as shown in Fig. 5, either connected directly or via a switch. Several trains can circulate at the same time, and the routing policy of trains should ensure that there is no collision and the system is always running. The modular design of

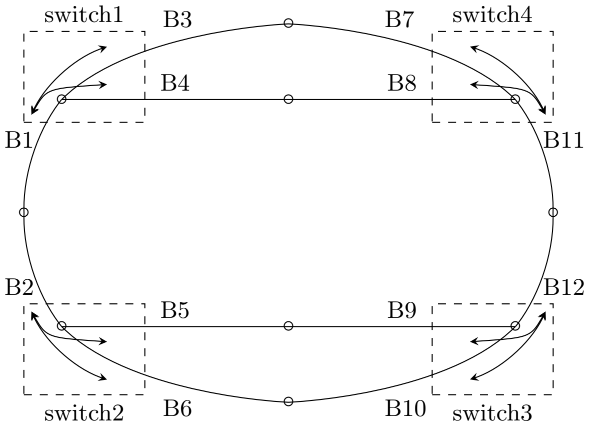
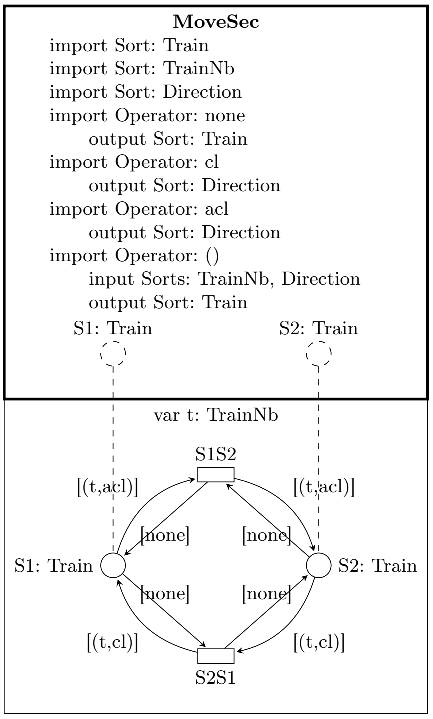
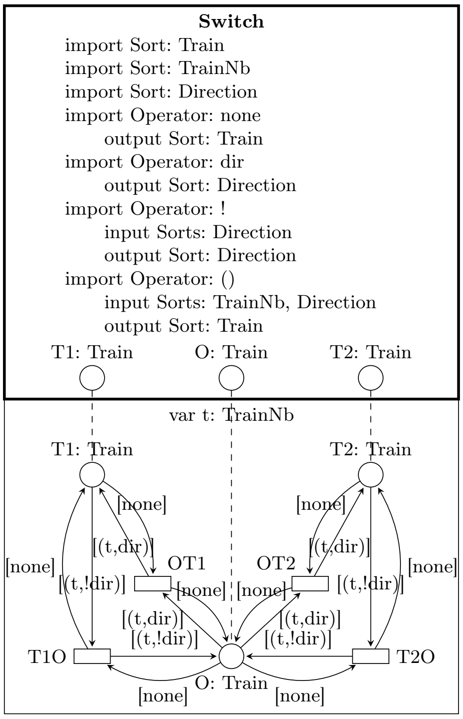


**Fig. 5.** The tracks of the model railway



(a) The MoveSec module



(b) The Switch module

**Fig. 6.** The `MoveSec` module

such a system was the scope of [12] and lead to identifying 2 modules: **MoveSec** models the moves between two directly connected tracks while **Switch** is a switch connecting three track sections. In both modules, each place corresponds to a track section, which may or may not be occupied by a train. The transitions reflect the possible moves. These two modules are depicted in Fig. 6 and are used by a top-level module, which captures the whole system, in Fig. 7.

We have chosen to define the track sections within the **Switch** modules since the switch is the most elaborate part of the system. Therefore the places in module **Switch** exports its places. Conversely, module **MoveSec** imports its places, as they are defined elsewhere.

Other choices in this particular example could have been made for import and export places. For instance, the tracks of a toy railway are asymmetric since for connecting pieces, one side of a track gets inside (the other side of) another track. This easily fits an imported place and an exported place scheme for the **MoveSec** module. But this also leads to two types of switch modules: one exporting places `T1` and `T2` and importing `O`, and the other doing the converse.

The choice we made illustrates parameterisation of modules. Note that module **Switch** imports a direction operator **dir**, which allows for using the same module to represent all switches, even though they operate in symmetrical ways. The

**Declarations**

| **Sorts** | **Operators** |
|---|---|
| Train | none: Train |
| TrainNb | cl: Direction |
| Direction | acl: Direction |
| | !: Direction,Direction |
| | (): TrainNb, Direction, Train |

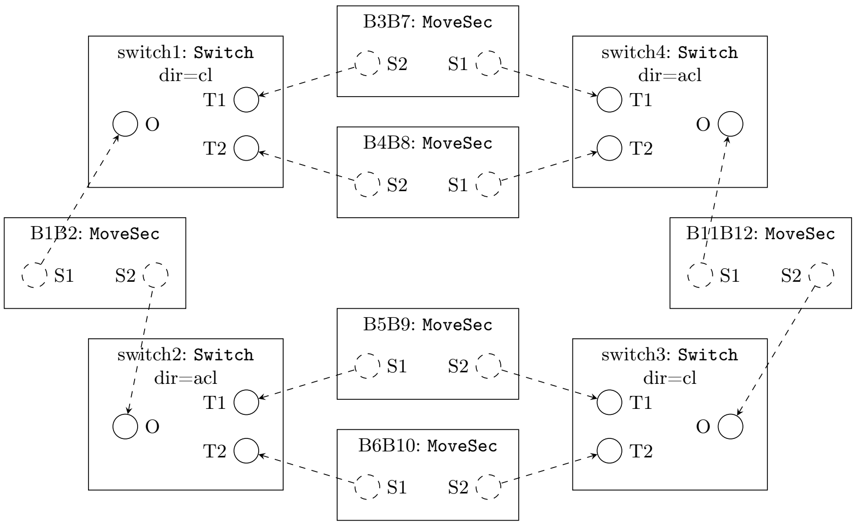


**Fig. 7.** The top-level net model of the model railway

operator is instantiated when connecting the **Switch** module, as shown in Fig. 7. It then takes value `cl` for switches `switch1` and `switch3`, and value `acl` for switches `switch2` and `switch4`.

Finally, the sorts and operators are defined in the top-level module and can be used consistently by all modules. This can be considered as a *global* definition. A `Train` on a track section is identified by a `TrainNb`, and a can move in a given `Direction`. The operators consist of 3 constants: `none` indicating that no train is on a track section, `cl` and `acl` giving the possible directions of trains (clockwise and anticlockwise). The unary operator `!` is intended to change a direction into the opposite one. Finally, `()` forms a pair with a train identity and a direction in which the train moves, detailing a train present on a track section.

## 7 Discussion and Extensions

In this section, we briefly discuss our module concept and some issues that should be considered for the work on Part 3 of the ISO/IEC-15909 standard.

### 7.1 Abstraction and Refinement

One of the main objectives of using modular design is to handle abstraction and refinement features. Our proposal fits with such a scheme by separating the module definition which lies at an abstract level and the module implementation. Moreover, modules can import constructs from others and provide constructs to be used by others.

Refinement can be pursued further, by detailing the functioning of a module through other new modules. To cope with such a process, it will be most helpful to provide a *hierarchy of modules*, showing how they are embedded in one another. The current module definitions allow us to build modules in a hierarchical way. However, for practical use, a designer should be provided with a view of the hierarchy (as in e.g., Hierarchical CPNs).

The example of Sect. 6 also shows that *parameterisation* of modules is possible. This is a key feature for reuse of modules in different contexts.

### 7.2 Aggregation of Label Information

In our formalisation, some annotations of places and transitions are ignored. For example, the initial marking of a place is always taken from the module where the place is actually defined. If a module imports a place, the module can define an initial marking. But this marking is irrelevant since it will come from wherever the imported place is defined. The same holds for the transition condition.

For the transition condition, it might make sense to use a conjunction of all transition conditions attached to the transition. As concerns the initial marking, it might make sense to use the sum of all initial markings. Since we started from the modular PNML semantics, we did not include that here.

Since such an *aggregation mechanism* seems to be reasonable in at least some cases, aggregation should be considered for the upcoming standard. However, this will introduce some technical difficulties. Not all annotations can be aggregated in a reasonable way: clearly the aggregation function would require an associative and commutative operation with a neutral element for making the aggregation independent from a specific order. Even if the operation is associative and commutative, its syntactical representation is not. Therefore, there would not be a canonical syntactical representation for the defined module implementation.

The aggregation mechanism could be even more advanced. For example, the defining module could provide the operation that is used for the aggregation. This way, it would be up to the defining module to decide whether and how particular labels of modules using it should be aggregated. What is reasonable, necessary, easily usable, and semantically sound is subject to future research.

### 7.3 Export of Variables

In our formalisation, modules can export and import only sort and operation symbols. It does not allow for exporting variables. In the case of synchronous communication via merging of transitions, it might, however, make sense to use a common variable for such transitions to exchange values between different partners during a synchronisation. Therefore, it might be worthwhile to also export and import some of the variables along with a transition.

A formalisation, however, requires that variables are defined locally to a transition as for example proposed by Schmidt [13]. The formalisation is a bit more technical, but we believe that this concept should be included in the standard.

### 7.4 Node Connection Policies

In our definition of export and import nodes, the other modules could connect to that node as to any other node. In some cases, some uses might not be intended at all. In our introductory example from Fig. 1, it does not make much sense for a module using the Channel module to add a token to export place p2. Though adding a token does not do much harm here, the module might want to restrict the use of this place so that tokens can only be removed from that place. Right now such a restriction cannot be enforced and would just be a textual recommendation of the use of a node.

It would be nice if a module could provide some composition policies that state in which way a node may be used, in order to define and to enforce communication paradigms. What exactly should be expressible by such policies and how a language for expressing such policies should look like, requires further investigation.

### 7.5 Generators

The key mechanism for having the module concept work is the generator. This way, it is possible to construct standard sorts out of existing sorts without even knowing the underlying algebra yet.

Up to now, there is only one fixed generator, which supports the standard generic constructs on sorts like multisets or products over sorts. It is not yet possible to define user-defined generic constructs. Of course, it would be useful to allow the extension of this generator within a module definition, so that a module could define new generic constructs. To this end, we could use existing theory from algebraic specifications. The question, however, is how much expressivity is needed and worth the effort to be included in the standard.

The idea of generators could also serve a different purpose: As we have seen, we used the generator for defining the built-in sorts and the standard constructs. Actually, many variants of high-level Petri nets differ only in these standard sorts and constructs. One example are well-formed nets [14], which are currently included as a special version in Part 1 of ISO/IEC 15909 (renamed symmetric nets). Generators could ease the definition of sub-classes of high-level Petri nets.

## 8 Related Work

Many modular constructs have been proposed in the literature. Our aim is to propose a framework capturing most of these mechanisms. In this section, we show how such mechanisms are dealt with.

Our approach extends the work in [5] by providing a formal and flexible definition. The communication mechanisms proposed in [15] are place fusion and transition fusion. They are easily handled by place and transition import/export features. The main difference with our proposal is the asymmetry between importing and exporting, whereas plain fusion is symmetric. But this is no restriction.

One of the earliest and most widespread modular approach is Hierarchical Coloured Petri Nets [16] and their implementation within CPNTools [6]. They also use the concept of port places, which can be defined as input, output or both. The structuring of nets is presented via a hierarchy of modules. Such nets also use the place fusion concept, which is captured by our proposal.

## 9 Conclusion

In this paper, we have shown that there is a formal foundation for the concepts of modular PNML. We also identified some issues that should be considered and resolved in the standardisation of the module concept in Part 3 of ISO/IEC 15909. All kinds of proposals, suggestions, and concerns are most welcome—as is any active participation in the standardisation process.

*Acknowledgements.* We would like to thank Anne Haxthausen and Hubert Baumeister for some helpful discussions on the category theory constructions behind the concepts of this paper. We hope that we, eventually, will write a joint paper from that perspective.

## References

1. ISO/IEC: Software and Systems Engineering – High-level Petri Nets, Part 1: Concepts, Definitions and Graphical Notation, International Standard ISO/IEC 15909 (2004)
2. Kindler, E., Weber, M.: A universal module concept for Petri nets – an implementation-oriented approach. Informatik-Bericht 150, Humboldt-Universität zu Berlin, Institut für Informatik (2001)
3. Weber, M., Kindler, E.: The Petri Net Markup Language. In: Ehrig, H., Reisig, W., Rozenberg, G., Weber, H. (eds.) Petri Net Technology for Communication-Based Systems. LNCS, vol. 2472, pp. 124–144. Springer, Heidelberg (2003)
4. ISO/JTC1/SC7/WG19: Software and Systems Engineering – High-level Petri Nets, Part 2: Transfer Format. FDIS 15909-2 (under ballot), v. 1.3.6, ISO/IEC (2008)
5. Kindler, E.: Modular PNML revisited: Some ideas for strict typing. In: Proc. AWPN 2007, Koblenz, Germany (2007)
6. Jensen, K., Kristensen, L.M., Wells, L.: Coloured Petri nets and CPN tools for modelling and validation of concurrent systems. Journal of Software Tools for Technology Transfer 9(3-4), 213–254 (2007)
7. Ehrig, H., Mahr, B.: Fundamentals of Algebraic Specifications 1, Equations and Initial Semantics. EATCS Monographs on Theoretical Computer Science, vol. 6. Springer, Heidelberg (1985)
8. Berthomieu, B., Choquet, N., Colin, C., Loyer, B., Martin, J., Mauboussin, A.: Abstract Data Nets combining Petri nets and abstract data types for high level specification of distributed systems. In: Proceedings of VII European Workshop on Application and Theory of Petri Nets (1986)
9. Vautherin, J.: Parallel systems specifications with coloured Petri nets and algebraic specifications. In: Rozenberg, G. (ed.) APN 1987. LNCS, vol. 266, pp. 293–308. Springer, Heidelberg (1987)
10. Billington, J.: Many-sorted high-level nets. In: Proceedings of the 3rd International Workshop on Petri Nets and Performance Models, pp. 166–179. IEEE Computer Society Press, Los Alamitos (1989)
11. Reisig, W.: Petri nets and algebraic specifications. Theoretical Computer Science 80, 1–34 (1991)
12. Choppy, C., Petrucci, L., Reggio, G.: A modelling approach with coloured Petri nets. In: Kordon, F., Vardanega, T. (eds.) Ada-Europe 2008. LNCS, vol. 5026, pp. 73–86. Springer, Heidelberg (2008)
13. Schmidt, K.: Verification of siphons and traps for algebraic Petri nets. In: Azéma, P., Balbo, G. (eds.) ICATPN 1997. LNCS, vol. 1248, pp. 427–446. Springer, Heidelberg (1997)
14. Chiola, G., Dutheillet, C., Franceschinis, G., Haddad, S.: On well-formed coloured nets and their symbolic reachability graph. In: Jensen, K., Rozenberg, G. (eds.) Petri Nets: Theory and Application, pp. 373–396. Springer, Heidelberg (1991)
15. Christensen, S., Petrucci, L.: Modular analysis of Petri nets. The Computer Journal 43(3), 224–242 (2000)
16. Jensen, K.: Coloured Petri Nets: Basic concepts, analysis methods and practical use. Basic concepts. Monographs in Theoretical Computer Science, vol. 1. Springer, Heidelberg (1992)

# Decidability Results for Restricted Models of Petri Nets with Name Creation and Replication[⋆]

Fernando Rosa-Velardo and David de Frutos-Escrig

Dpto. de Sistemas Informáticos y Computación
Universidad Complutense de Madrid
{fernandorosa,defrutos}@sip.ucm.es

**Abstract.** In previous works we defined $\nu$-APNs, an extension of P/T nets with the capability of creating and managing pure names. We proved that, though reachability is undecidable, coverability remains decidable for them. We also extended P/T nets with the capability of nets to replicate themselves, creating a new component, initially marked in some fixed way, obtaining g-RN systems. We proved that these two extensions of P/T nets are equivalent, so that g-RN systems have undecidable reachability and decidable coverability. Finally, for the class of the so called $\nu$-RN systems, P/T nets with both name creation and replication, we proved that they are Turing complete, so that also coverability turns out to be undecidable. In this paper we study how can we restrict the models of $\nu$-APNs (and, therefore, g-RN systems) and $\nu$-RN systems in order to keep decidability of reachability and coverability, respectively. We prove that if we forbid synchronizations between the different components in a g-RN system, then reachability is still decidable. The proof is done by reducing it to reachability in a class of multiset rewriting systems, similar to Recursive Petri Nets. Analogously, if we forbid name communication between the different components in a $\nu$-RN system, or restrict communication to happen only for a given finite set of names, we obtain decidability of coverability.

## 1 Introduction

Pure names are identifiers with no relation between them other than equality [11]. They were first mentioned by Needham, who said that pure names are *nothing but a bit pattern that is an identifier, and is only useful for comparing for identity with other bit patterns* [17]. Names are relevant to mobility and security because they can be used to represent channels [16], keys [1] or ambients [5].

In previous works we have studied a very simple extension of P/T nets, that we called $\nu$-APNs [18, 19]. Tokens in $\nu$-APNs are pure names, that can be created fresh, moved along the net and used to restrict the firing of transitions with name matching. $\nu$-APNs have been recently used in [7] to model complex choreographies proposed in [6].

[⋆] Work partially supported by the Spanish projects DESAFIOS TIN2006-15660-C02-02, WEST TIN2006-15578-C02-01 and PROMESAS-CAM S-0505/TIC/0407.

In [13] the author proves undecidability of reachability for every object-oriented Petri net formalism. For that purpose, Minimal OO-nets are defined, as a minimal model of nets having objects as tokens, assuming that, at least, each object has a name. Though our $\nu$-APNs were thought of in a different context, they essentially correspond to the minimal OO-nets of [13].

Another model based on Petri nets that has names as tokens are *Data Nets* [14]. In Data Nets, tokens are not pure in general, but taken from a linearly-ordered infinite domain. Names can be created, but they cannot be guaranteed to be fresh.

Other similar models include Nested Petri Nets [15] that, following the nets-within-nets paradigm, have nets as tokens (object nets), that can evolve autonomously (object-autonomous steps), move along the system net (transport steps), synchronize with each other (horizontal synchronization steps) or synchronize with the system net (vertical synchronization steps). Nested nets are more expressive than $\nu$-APNs. Indeed, it is possible to simulate every $\nu$-APN by means of a Nested Petri Net which uses only object-autonomous and horizontal synchronization steps. Therefore, undecidability results can be transferred from Nested Petri Nets to $\nu$-APNs, but this is not the case for decidability results.

In the field of mobility, particularly in that of mobile agent systems, components usually have the capacity to replicate themselves, that is, the capacity of creating a new copy of themselves. In previous works we also extended P/T nets with a simple primitive that creates a new net, marked in some fixed way. We called g-RN to this extension, where the "g" stands for "garbage", since we also consider a garbage collection mechanism that removes empty nets, those without tokens, that are assumed to be blocked. Therefore, the number of components in a system cannot only grow when a new replication is executed, but also decrease when a component becomes garbage.

We know that reachability in $\nu$-APNs is undecidable [13], though coverability is still decidable [18]. Moreover, in [19] we proved that $\nu$-APNs and g-RN systems are equivalent, in a sense that preserves both reachability and coverability, so that we also know that reachability is undecidable for g-RN systems, but coverability is decidable for them.

Finally, also in [19] we extended P/T names both with name creation and replication, obtaining $\nu$-RN systems, and proving that, although both extensions were equivalent, when we consider them simultaneously we achieved Turing-completeness. In particular, coverability is undecidable for them.

In this paper we study how both models, g-RN systems (or equivalently $\nu$-APNs) and $\nu$-RNs can be restricted in order to keep decidability of reachability and coverability, respectively. We will prove that reachability is decidable for the class of g-RN systems without synchronizations. The proof is done by first reducing it to reachability in a multiset rewriting system with conditional rewrite rules, where the conditions are reachability problems in ordinary P/T nets. This technique is somewhat similar to the model of Recursive Petri Nets [12], in which some transitions (the so called *abstract* transitions) are not atomic. They first remove tokens from preconditions, but do not put them in postconditions until a new component (a child thread created by the abstract transition, initially marked in some fixed way) fires a *final* transition. However, there are important differences between these models, that do not allow to reduce reachability in those rewriting systems to reachability for RPNs.

For the model of $\nu$-RN systems, that encompasses both name creation and replication, we prove that by forbidding name communication between components, although maintaining synchronizations, coverability remains decidable. If communication is allowed, but restricted to happen for names in a given finite set, then we also prove decidability of coverability. We prove that, with these restrictions, $\nu$-RNs are a well-structured system [10], for which coverability is decidable.

## 2 Preliminaries

Given an arbitrary set $A$, we will denote by $\mathcal{MS}(A)$ the set of multisets of $A$, that is, the set of mappings $m : A \to \mathbb{N}$. We denote by $S(m)$ the support of $m$, that is, the set $\{a \in A \mid m(a) > 0\}$. A multiset $m$ is finite if $S(m)$ is a finite set, in which case we denote by $|m| = \sum\limits_{a\in S(m)} m(a)$ the cardinality of $m$. Given two multisets $m_1, m_2 \in \mathcal{MS}(A)$ we denote by $m_1 + m_2$ the multiset defined by $(m_1 + m_2)(a) = m_1(a) + m_2(a)$. We will write $m_1 \subseteq m_2$ if $m_1(a) \leq m_2(a)$ for every $a \in A$. In this case, we can define $m_2 - m_1$, given by $(m_2 - m_1)(a) = m_2(a) - m_1(a)$. We will denote by $\sum$ the extended multiset sum operator and by $\emptyset \in \mathcal{MS}(A)$ the multiset $\emptyset(a) = 0$, for every $a \in A$. If $f : A \to B$ and $m \in \mathcal{MS}(A)$, then we define $f(m) \in \mathcal{MS}(B)$ by $f(m)(b) = \sum\limits_{f(a)=b} m(a)$. We will also use set notation to specify multisets, as it is standard.

Every partial order $\leq$ defined in $A$ induces a partial order $\sqsubseteq$ in $\mathcal{MS}(A)$, given by $\{a_1, \ldots, a_n\} \sqsubseteq \{b_1, \ldots, b_m\}$ if there is some $h : \{1, \ldots, n\} \to \{1, \ldots, m\}$ injective such that $a_i \leq b_{h(i)}$ for all $i$. We write $s < s'$ if $s \leq s'$ and $s' \not\leq s$ (analogously, we write $\sqsubset$ for $\sqsubseteq$). A partial order $\leq$ is a well-quasi order (wqo) if for every infinite chain $s_0, s_1, \ldots$ there are $i$ and $j$ with $i < j$ such that $s_i \leq s_j$. Equivalently, it is a wqo if every infinite sequence has an increasing subsequence. It is a well known fact that the multiset order induced by a wqo is also a wqo.

Along the paper we assert several times that a model $\mathbf{M}'$ simulates another model $\mathbf{M}$. By that we mean that for every system $\mathcal{N}$ in $\mathbf{M}$ there is $\mathcal{N}' = F(\mathcal{N})$ in $\mathbf{M}'$, where $F$ is a computable function, such that the transition systems generated by the semantics of $\mathcal{N}$ and $\mathcal{N}'$ are isomorphic. Therefore, reachability in $\mathcal{N}$ and $\mathcal{N}'$ are equivalent. Moreover, the isomorphisms preserve the natural orders in each of the models, so that coverability is also equivalent.

In order to set notations, we next define P/T nets.

**Definition 1.** *A P/T net is a tuple $N = (P, T, F)$ where $P$ and $T$ are disjoint finite sets and $F \subseteq (P \times T) \cup (T \times P)$. A marking of $N$ is any $M \in \mathcal{MS}(P)$.*

As usual, we denote by $t^\bullet$ and ${}^\bullet t$ the set of postconditions and preconditions of $t$, respectively, that is, $t^\bullet = \{p \mid (t, p) \in F\}$ and ${}^\bullet t = \{p \mid (p, t) \in F\}$.

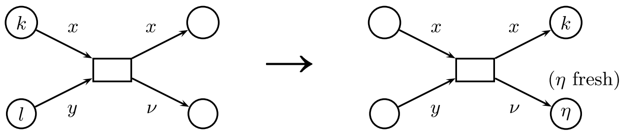


**Fig. 1.** A simple $\nu$-APN

**Definition 2.** *Let $N$ be a P/T net and $M$ a marking of $N$. A transition $t$ is enabled at marking $M$ if ${}^\bullet t \subseteq M$. The reached state of $N$ after the firing of $t$ is $M' = (M - {}^\bullet t) + t^\bullet$.*

We will write $M \xrightarrow{t} M'$ if $M'$ is the reached marking after the firing of $t$ at marking $M$, $M \longrightarrow M'$ if there is some $t$ such that $M \xrightarrow{t} M'$, and denote by $\longrightarrow^*$ the reflexive and transitive closure of $\longrightarrow$.

## 3 Name Creation and Replication

In this section we briefly present $\nu$-APNs and g-RN systems. For more details see [19]. In $\nu$-APNs tokens are names taken from an infinite set $Id$. In order to handle names, we need matching variables labelling the arcs of the nets, taken from a set $Var$. Moreover, we add a primitive capable of creating fresh names, formalized by means of a special variable $\nu \in Var$.

**Definition 3.** *A $\nu$-APN is a tuple $N = (P, T, F)$, where $P$ and $T$ are finite disjoint sets, and $F : (P \times T) \cup (T \times P) \rightarrow Var$ is a partial function such that for every $(p, t)$ in the domain of $F$, $F(p, t) \neq \nu$.*

We denote by $Var(t)$ the set of variables labelling arcs adjacent to $t$.

**Definition 4.** *A marking of a $\nu$-APN $N = (P, T, F)$ is any $M : P \rightarrow \mathcal{MS}(Id)$. We define $S(M) = \bigcup_{p \in P} S(M(p))$.*

The set $S(M) \subset Id$ is the set of all the names appearing in some place according to $M$.

Transitions are fired with respect to a mode, that chooses which tokens are taken from preconditions and which are put in postconditions. Given a transition $t$ of a net $N$, a mode of $t$ is a mapping $\sigma : Var(t) \rightarrow Id$, that instantiates each variable involved in the firing of $t$ to an identifier. We will use $\sigma, \sigma', \sigma_1 \ldots$ to range over modes. In the following definition we assume that $\{\sigma(F(t, p))\} = \emptyset$ and $\{\sigma(F(p, t)) = \emptyset\}$ whenever $F(t, p)$ and $F(p, t)$ are not defined, respectively.

**Definition 5.** *Let $N$ be a $\nu$-APN, $M$ a marking of $N$, $t$ a transition of $N$ and $\sigma$ a mode of $t$. We say $t$ is enabled with mode $\sigma$ if $\sigma(\nu) \notin S(M)$ and for all $p \in {}^\bullet t$, $\sigma(F(p, t)) \in M(p)$. The reached state of $N$ after the firing of $t$ with mode $\sigma$ is the marking $M'$, given by*

$$M'(p) = (M(p) - \{\sigma(F(p, t))\}) + \{\sigma(F(t, p))\} \quad \forall p \in P$$

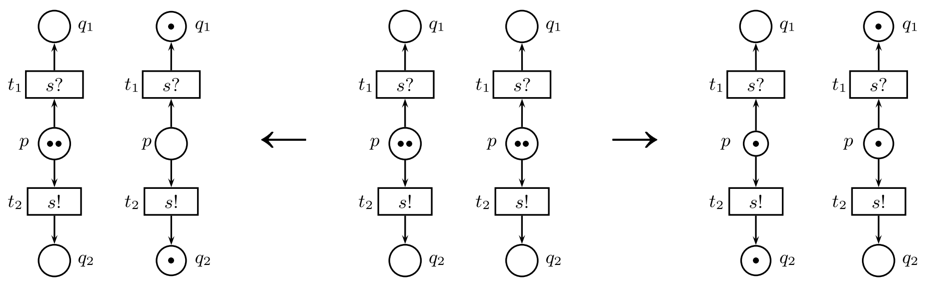


**Fig. 2.** g-RN system with marking $\mathcal{M} = \{\{p,p\},\{p,p\}\}$ and two possible firings

We will write $M \xrightarrow{t(\sigma)} M'$ if $M'$ is reached from $M$ when $t$ is fired with mode $\sigma$. Analogously as for P/T nets, we also have the relations $\longrightarrow$ and $\longrightarrow^*$. Fig. 1 depicts a simple example of a $\nu$-APN and the firing of its only transition.

In order to capture the intuition that the names in *Id* are pure, we work modulo $\equiv_\alpha$, which allows consistent renaming of names in markings. Accordingly, the order $\sqsubseteq_\alpha$ that induces coverability for $\nu$-APNs is defined as follows: $M \sqsubseteq_\alpha M'$ if there is an injection $\iota : S(M) \to S(M')$ such that for every place $p \in P$, $\iota(M(p)) \subseteq M'(p)$.

We now give a brief insight of g-RN systems, the extension of P/T nets with replication. In a g-RN system nets have two types of transitions: synchronizing transitions and replicating transitions. Synchronizing transitions are those which are meant to be fired synchronously. For that purpose we will consider a set $\mathcal{S}$ of service names, endowed with a function $\mathit{arity} : \mathcal{S} \to \mathbb{N}$ and we take the set of synchronizing labels $\mathit{Sync} = \{s(i) \mid s \in \mathcal{S},\ 1 \leq i \leq \mathit{arity}(s)\}$. If $\mathit{arity}(s) = 2$ then we will write $s?$ and $s!$ instead of $s(1)$ and $s(2)$, respectively, that can be interpreted as the offer and request of the service $s$. We denote by $\mathcal{A}$ the set of labels $s$ with arity one, and identify $s \in \mathcal{A}$ with $s(1)$. Then, a synchronous firing can happen whenever $n$ compatible transitions (having labels $s(1), \ldots, s(n)$ for some $s \in \mathcal{S}$ with arity $n$) are enabled. In that case they can all be fired simultaneously, following the ordinary token game (see Fig. 2). Replicating transitions are labelled with a marking, and when fired produce a new component, initially marked as indicated by the replicating transition (see Fig. 3).

Therefore, markings of g-RN systems are multisets of markings of their components. We will identify a component by its current marking (out of repetitions), and talk about component $M$. We also consider a very simple garbage collection mechanism that removes empty components from markings.

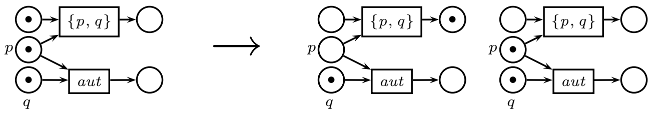


**Fig. 3.** g-RN system firing a replication transition

**Definition 6.** *A g-RN system is a labelled Petri net $N = (P, T, F, \lambda)$ where:*

- *$P$ and $T$ are finite disjoint sets of places and transitions, respectively,*
- *$F \subseteq (P \times T) \cup (T \times P)$ is the set of arcs of the net,*
- *$\lambda$ is a function from $T$ to the set of labels $Sync \cup \mathcal{MS}(P)$.*

*A component of $N$ is any $M \in \mathcal{MS}(P)$, and a marking of $N$ is a finite multiset of components of $N$.*

A component $M$ is an ordinary marking of $N$ if considered as an ordinary P/T net. We will use $\mathcal{M}$, $\mathcal{M}'$, $\mathcal{M}_1$,... to range over markings. We assume that every transition has a precondition, so that every firing needs the presence of a token. Therefore, empty components cannot fire any transition, so that they can be considered as garbage. We identify markings up to $\equiv$, the least congruence such that $\mathcal{M} \equiv \mathcal{M} + \{\emptyset\}$. We denote by $Init_N$ or simply $Init$ when there is no confusion, the set of components that appear as labels of replicating transitions, that is, $Init_N = \{\lambda(t) \mid t \in T,\ \lambda(t) \in \mathcal{MS}(P)\}$.

The general definition of enabling and firing of transitions in g-RN system is a bit technical, because modes must specify which components are going to fire each of the transitions, and these components are not necessarily different. Again, the interested reader is referred to [19] for further details. However, in this work we will mostly be interested in those g-RN that cannot synchronize, that is, those in which transitions are either replicating transitions (labelled with a component) or autonomous transitions (labelled with some element of $\mathcal{A}$).

Therefore, next we present the simplified definition of enabling and firing of transitions for the considered subclass of g-RN systems.

**Definition 7.** *Let $N$ be a g-RN system without synchronizations, $t$ a transition of $N$ and $\mathcal{M}$ a marking of $N$. We say $t$ is enabled for a component $M \in \mathcal{M}$ if ${}^\bullet t \subseteq M$. The reached marking after the firing of $t$ for $M$ is $\mathcal{M}' = (\mathcal{M} - \{M\}) + \{M', \overline{M}\}$, where:*

- $M' = (M - {}^\bullet t) + t^\bullet$, *and*
- $\overline{M} = \begin{cases} \emptyset, \textit{ if } \lambda(t) \in \mathcal{A}, \\ \lambda(t), \textit{ otherwise.} \end{cases}$

Therefore, when $t$ is autonomous, the firing of $t$ consists on the ordinary firing of transition $t$, so that $M$ evolves to $M'$. When $t$ is a replicating transition, the new component $\lambda(t)$ is also added to the marking. Moreover, notice that since we are working modulo $\equiv$, every time a component becomes empty, it is automatically removed from the current marking.

In [19] we proved that $\nu$-APNs and g-RN systems (with synchronizations) are equivalent, in the sense that they can simulate each other, so that the decidability/undecidability results from $\nu$-APNs can be transferred to g-RN systems. In particular, we know that both $\nu$-APNs and g-RN systems have undecidable reachability, and both have decidable coverability.

Finally, $\nu$-RN systems can be defined as g-RN systems with names as tokens or, equivalently, as $\nu$-APNs with replication transitions. A marking of a $\nu$-RN system is a multiset of components, where now components are not multisets of places, but mappings $M : P \to \mathcal{MS}(Id)$. In [19] we proved that $\nu$-RN systems are Turing complete. In particular, both reachability and coverability are undecidable for them.

## 4 Decidability of Reachability for g-RN Systems without Synchronizations

Next let us prove that the subclass of g-RN systems that cannot synchronize have decidable reachability. First let us introduce some notations that we will use throughout this section, that deal with the behaviour of a component when considering it in isolation, that is, without considering it as part of a system. We will write $M \xrightarrow{\mathcal{M}} M'$ to denote the fact that $M'$ can be reached from $M$ by firing a *sequence* of transitions whose set of replicating transitions produces the new components in $\mathcal{M}$. Analogously, we will also write $M \xrightarrow{\mathcal{M}^{\geq}} M'$, when $M'$ can be reached from $M$ by at least producing the new components in $\mathcal{M}$. Since reachability is decidable for ordinary P/T nets [9], we immediately obtain the following decidability results.

**Lemma 1.** *Given $M_1$, $M_2$ and $\mathcal{M}$, it is decidable whether $M_1 \xrightarrow{\mathcal{M}} M_2$.*

*Proof.* Deciding $M_1 \xrightarrow{\mathcal{M}} M_2$ amounts to deciding whether $M_2$ is reachable from $M_1$ having fired some transitions (the ones labelled with components in $\mathcal{M}$) a certain number of times. As it is standard, we add a postcondition $p_M$ to each transition $t$ with $\lambda(t) = M \in \mathit{Init}$. Then, $M_1 \xrightarrow{\mathcal{M}} M_2$ if and only if $M_1 \longrightarrow^* M_2 + \overline{M}$, where $\overline{M}(p_M) = \mathcal{M}(M)$.

In order to decide, $M_1 \xrightarrow{\mathcal{M}^{\geq}} M_2$ we also add the places $p_M$ together with a new place *ok*, and we use the standard technique of reducing coverability to reachability, but only applied to the places $p_M$. We add a transition[1] with $\{p_M \mid M \in \mathcal{M}\}$ as precondition and *ok* as postcondition. Then $M_1 \xrightarrow{\mathcal{M}^{\geq}} M_2$ if and only if the submarking $M_2 + \{ok\}$ (without considering the places $p_M$) is reachable.

This section is devoted to proving that given $\mathcal{M}_0$ and $\mathcal{M}_f$, markings of a g-RN system $N$ without synchronizations, we can decide whether $\mathcal{M}_0 \to^* \mathcal{M}_f$. Let us denote by $\mathcal{R}(\mathcal{M}_0, \mathcal{M}_f)$ or just $\mathcal{R}$ when there is no confusion, the set of components appearing in the initial or final marking, or in some replicating transition, $\mathcal{R}(\mathcal{M}_0, \mathcal{M}_f) = S(\mathcal{M}_0) \cup S(\mathcal{M}_f) \cup \mathit{Init}_N$.

Consider any trace reaching $\mathcal{M}_f$ from such $\mathcal{M}_0$. Every component that appears in any marking of that trace evolves in its own, because there are no

[1] We are now assuming that arcs have weights to keep the ideas clear. Weights could have actually been considered, without affecting any of the results presented.

synchronizations. Since the final marking is eventually reached, all of those components either evolve to the empty component (possibly creating on their way other components) or to some of the components in the final marking (again, possibly creating other components). In other words, we are not interested in the full behavior of components, but only in those aspects: whether they evolve to the empty marking, whether they evolve to some component in the final marking and, in both cases, what components it creates.

Therefore, in order to simplify our study by focusing on those aspects, we will not work directly over the reachability graph generated by the g-RN system, but over that of the transition system that we introduce next, that takes accurately into account the previous considerations.

**Definition 8.** *Let $N = (P, T, F, \lambda)$ be a g-RN system without synchronizations, $\mathcal{M}_0$ and $\mathcal{M}_f$ two markings of $N$. Let us define the transition system $l(N) = (S, \mapsto)$, given by:*

- *$S = \mathcal{MS}(\mathcal{R}(\mathcal{M}_0, \mathcal{M}_f))/_{\equiv}$,*
- *$\mapsto$ is the least congruence (with respect to multiset addition) such that*

$$\frac{M \xrightarrow{\mathcal{M}} M'}{\{M\} \mapsto \{M'\} + \mathcal{M}}$$

Therefore, each step $\mathcal{M} \mapsto \mathcal{M}'$ represents part of the life of a component $M$ in $\mathcal{M}$, that either disappears if $M' = \emptyset$, or evolves to a component in $\mathcal{R}$. The behavior of $N$ that we are interested on is reflected in $l(N)$, as we prove in the next result.

**Proposition 1.** *For any $\mathcal{M}_1$ and $\mathcal{M}_2$ in $\mathcal{R}$, $\mathcal{M}_1 \rightarrow^* \mathcal{M}_2$ in $N \Leftrightarrow \mathcal{M}_1 \mapsto^* \mathcal{M}_2$ in $l(N)$.*

*Proof.* We prove that $\mathcal{M}_1 \rightarrow^* \mathcal{M}_2$ implies $\mathcal{M}_1 \mapsto^* \mathcal{M}_2$ (the converse implication is trivial by definition of $\mapsto$). The proof is by induction on the number of created components in the trace. If no component is created, then $\mathcal{M}_1 = \{M_1, \ldots, M_n\}$, $\mathcal{M}_2 = \{M'_1, \ldots, M'_n\}$ and $M_i \xrightarrow{\emptyset} M'_i$ for $i = 1, \ldots, n$. Then, for all $i$, we can derive $\{M_i\} \mapsto \{M'_i\}$, and because it is a congruence, $\{M_1, \ldots, M_n\} \mapsto^* \{M'_1, \ldots, M'_n\}$.

Let us now suppose that some component is created in the trace. In that case, there is some component that is created last, by some other component $\overline{M}$. This component was either in the initial marking or it was created by some other component, so in any case $\overline{M} \in \mathcal{R}$. Let $\mathcal{M}$ be the multiset of all the components created by $\overline{M}$. Since no more components are created after those in $\mathcal{M}$, $\mathcal{M}$ evolves to some $\mathcal{M}' \subseteq \mathcal{M}_f$, and every $M \in \mathcal{M}$ satisfies $M \xrightarrow{\emptyset} M'$ for some $M' \in S(\mathcal{M}_2)$ or for $M' = \emptyset$. Then, we can derive that $\{M\} \mapsto \{M'\}$ and, as in the base case, $\mathcal{M} \mapsto^* \mathcal{M}'$.

Now we have to distinguish between two cases: the one in which $\overline{M}$ evolves to the empty marking, $\overline{M} \xrightarrow{\mathcal{M}} \emptyset$ (so that $\{\overline{M}\} \mapsto \mathcal{M} \mapsto^* \mathcal{M}'$), and the one in which it evolves to some component in the final marking, $\overline{M} \xrightarrow{\mathcal{M}} M_f$ (so that $\{\overline{M}\} \mapsto \mathcal{M} + \{M_f\} \mapsto^* \mathcal{M}' + \{M_f\}$). In the first case, we can reorder the trace so that

$$\mathcal{M}_1 \to^* \mathcal{M}_2 - \mathcal{M}' + \{\overline{M}\} \to^* \mathcal{M}_2$$

The induction hypothesis tells us that $\mathcal{M}_1 \mapsto^* \mathcal{M}_2 - \mathcal{M}' + \{\overline{M}\} \mapsto^* \mathcal{M}_2 - \mathcal{M}' + \mathcal{M}' = \mathcal{M}_2$. Analogously, we obtain $\mathcal{M}_1 \mapsto^* \mathcal{M}_2$ in the second case.

From now on, we will study the reachability problem for the transition systems $l(N)$. The main problem when we try to devise an algorithm to decide reachability in g-RN systems is that of components that may evolve to the empty marking, possibly by creating other components that could also eventually disappear. In order to handle this difficulty we will define the following order, that takes into account those markings.

**Definition 9.** *We write $\mathcal{M}_1 \sqsubseteq_\emptyset \mathcal{M}_2$ whenever $\mathcal{M}_2 = \mathcal{M}_1 + \mathcal{M}$ and $\mathcal{M} \mapsto^* \emptyset$.*

That is, $\mathcal{M}_1 \sqsubseteq_\emptyset \mathcal{M}_2$ if $\mathcal{M}_1 \subseteq \mathcal{M}_2$ and $\mathcal{M}_2 - \mathcal{M}_1 \mapsto^* \emptyset$. In the first place, the defined relation is reflexive, transitive and anti-symmetric.

**Lemma 2.** *$\sqsubseteq_\emptyset$ is a partial order.*

Moreover, given $\mathcal{M}_1$ and $\mathcal{M}_2$, there is a procedure to effectively determine whether $\mathcal{M}_1 \sqsubseteq_\emptyset \mathcal{M}_2$. In order to see it, let us first see two results.

**Lemma 3.** *Given $M$, the set of all minimal $\mathcal{M}$ (with respect to multiset inclusion) such that $M \xrightarrow{\mathcal{M}} \emptyset$ is computable.*

*Proof.* Actually, we prove the more general result of computing all minimal $\mathcal{M}$ greater than a given $\overline{\mathcal{M}}$ such that $M \xrightarrow{\mathcal{M}} \emptyset$, so that we obtain the result by taking the particular case in which $\overline{\mathcal{M}} = \emptyset$. If $M \not\to^* \emptyset$ (as marking of a P/T net) then there is no such $\mathcal{M}$, and the set we are computing is empty. Let us suppose that $M \to^* \emptyset$. We proceed by induction on $n$, the number of components in *Init*. If $n = 0$ then it is the case that $M \xrightarrow{\emptyset} \emptyset$, so that the only marking to consider is the empty one (which is minimal).

Let us now consider the inductive case, that is, $\mathit{Init} = \{M_1, \dots, M_n\}$ for some $n > 0$. We know that $M \to^* \emptyset$, so that we know that there is at least one $\mathcal{M}_\emptyset$ such that $M \xrightarrow{\mathcal{M}_\emptyset} \emptyset$. Then, we can do a breadth-first search in the lattice of markings (see Fig. 4) to compute one minimal $\mathcal{M}_\emptyset$ greater than $\overline{\mathcal{M}}$ such that $M \xrightarrow{\mathcal{M}_\emptyset} \emptyset$.

Now we need to compute the rest of the minimal markings, though we do not need to search among those greater than $\mathcal{M}_\emptyset$ (with respect to multiset inclusion) because we know that any solution greater than $\mathcal{M}_\emptyset$ would not be minimal. Let us denote by $k_i$ the number of times that the component $M_i \in \mathcal{R}$ appears in $\mathcal{M}_\emptyset$, that is, $k_i = \mathcal{M}_\emptyset(M_i)$. For each $i \in \{1, \dots, n\}$ and all $j \in \{0, \dots, k_i - 1\}$

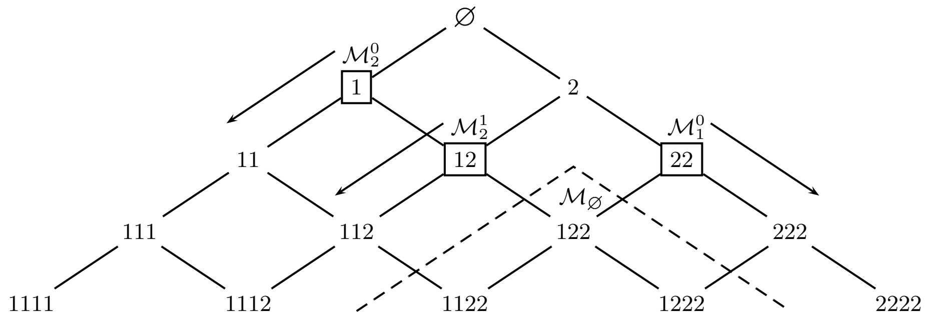


**Fig. 4.** Computation of all minimal $\mathcal{M}_\emptyset$ when $|Init| = 2$

let $\mathcal{M}_i^j$ be the marking in which the $i$-th component $M_i$ of *Init* appears $j$ times, and the $l$-th component $M_l$ appears $k_l$ times, for all $l \neq i$, that is

$$\mathcal{M}_i^j(M_l) = \begin{cases} j \text{ if } l = i, \\ k_l \text{ if } l \neq i \end{cases}$$

Now for each $i$ and $j$, let us see that we only need to look for all minimal $\mathcal{M}'$ greater than $\mathcal{M}_i^j$ and whose number of components $M_i$ does not increase. Indeed, let $i$ and $j$ such that $j < k_i - 1$ and $\mathcal{M}$ such that $\mathcal{M}_i^j \subset \mathcal{M}$ with $\mathcal{M}(M_i) = j + 1$. Since $\mathcal{M}_i^j \subset \mathcal{M}$, $\mathcal{M}(M_l) \geq \mathcal{M}_i^j(M_l) = k_l = \mathcal{M}_i^{j+1}(M_l)$, and $\mathcal{M}(M_i) = j + 1 = \mathcal{M}_i^{j+1}(M_i)$. Then we have that $\mathcal{M}_i^{j+1} \subseteq \mathcal{M}$. Similarly, we can see that for $j = k_i - 1$, any $\mathcal{M}$ greater than $\mathcal{M}_i^j$ such that $\mathcal{M}(M_i) = j + 1$ satisfies that $\mathcal{M}_\emptyset \subseteq \mathcal{M}$.

Then, for every $\mathcal{M}_i^j$ we can "block" the firing of the replicating transitions that create $M_i$, and apply the induction hypothesis to compute the set $Min_i^j$ of all minimal markings greater than $\mathcal{M}_i^j$ that comply with the thesis. Now we can compute the set we are looking for as $\{\mathcal{M}_\emptyset\} \cup \bigcup Min_i^j$.

In Fig. 4 you can see an example of the reasoning followed in the previous proof, in the restricted case in which there are two different replicating transitions, that is, when $|Init| = 2$. In it, the first four levels of the lattice of all multisets with elements in *Init* is depicted, denoting by 1 the component $M_1$ and by 2 the component $M_2$, so that we write for instance 1122 to represent the multiset $\{M_1, M_1, M_2, M_2\}$. In Fig. 4 it is assumed that the first marking that we find when we search the lattice for a marking $\mathcal{M}_\emptyset$ such that $M \xrightarrow{\mathcal{M}_\emptyset} \emptyset$ is 122, so that $k_1 = 1$ and $k_2 = 2$. In that case, we do not need to keep searching among those markings greater than 122, those inside the dashed line. In order to keep searching among the markings that are not greater than $\mathcal{M}_\emptyset$ the proof of the previous result builds $\mathcal{M}_1^0$ (because $k_1 = 1$) and both $\mathcal{M}_2^0$ and $\mathcal{M}_2^1$ (because $k_2 = 2$), which correspond to the markings inside boxes in the picture. As you can see, now it is enough to keep searching by following the arrows, as proved in the last part of the previous proof. For instance, the marking 1112 is greater than

$\mathcal{M}_2^0$ if we allow 2 to be created, but is also greater than $\mathcal{M}_2^1$ without allowing creation of component 2. Moreover, any marking greater than $\mathcal{M}_2^1$ that creates component 2 is also greater than $\mathcal{M}_\emptyset$.

Next, two simple lemmas that we will need to prove that the order $\sqsubseteq_\emptyset$ is decidable.

**Lemma 4.** *$\mathcal{M} \mapsto^* \emptyset$ if and only if for all $M \in S(\mathcal{M})$, $\{M\} \mapsto^* \emptyset$.*

**Lemma 5.** *If $\mathcal{M} \subseteq \mathcal{M}'$ and $\mathcal{M}' \mapsto^* \emptyset$ then $\mathcal{M} \mapsto^* \emptyset$.*

*Proof.* If $\mathcal{M} \not\mapsto^* \emptyset$ then, by the previous lemma there is $M \in \mathcal{M}$ such that $\{M\} \not\mapsto^* \emptyset$. Since $\mathcal{M} \subseteq \mathcal{M}'$ then $M \in \mathcal{M}'$, so that again by the previous result, $\mathcal{M}' \not\mapsto^* \emptyset$.

The following result, that will allow us to conclude that the defined order is decidable, is a weak form of the decidability result we are looking for.

**Proposition 2.** *Given a marking $\mathcal{M}$, it is decidable whether $\mathcal{M} \mapsto^* \emptyset$.*

*Proof.* By Lemma 4, it is enough to decide whether $\{M\} \mapsto^* \emptyset$ for a given component $M$. We proceed by induction on the number of components in *Init*. If there are no components then $M$ cannot replicate, and it is enough to decide whether $M \rightarrow^* \emptyset$, which is decidable. Let us see the inductive case.

If $M \not\rightarrow^* \emptyset$ then clearly $\{M\} \not\mapsto^* \emptyset$. Otherwise, by Lemma 3 we can consider all minimal $\mathcal{M}$ such that $M \xrightarrow{\mathcal{M}} \emptyset$. We have to decide whether at least one of those $\mathcal{M}$ satisfies $\mathcal{M} \rightarrow^* \emptyset$. Notice that, thanks to Lemma 5, it is enough to consider minimal markings (the empty marking can be reached if and only if it can be reached from the minimal ones). Notice also that, because we are beginning the trace in $M$, it is enough to consider traces that do not create marking $M$, so that we can remove $M$ from *Init*. Therefore, we can apply the induction hypothesis and we can conclude.

**Corollary 1.** *$\sqsubseteq_\emptyset$ is a decidable partial order.*

Though Proposition 2 is only a step away from the result we are looking for, decidability of general reachability, it does not look immediate to generalize the previous result to the general one, essentially because we do not have a result analogous to Lemma 5 in the general case. Instead, we will adapt the widely used technique of well quasi-orders (wqo) for our purposes. In general, the technique is used to prove decidability of the so called control reachability [3], that in our setting amounts to coverability. However, the coverability problem induced by $\sqsubseteq_\emptyset$ is just reachability. Indeed, a marking $\mathcal{M}'$ such that $\mathcal{M} \sqsubseteq_\emptyset \mathcal{M}'$ is reachable if and only if $\mathcal{M}$ is reachable, because $\mathcal{M}' = \mathcal{M} + \overline{\mathcal{M}}$ with $\overline{\mathcal{M}} \rightarrow^* \emptyset$ and, therefore, $\mathcal{M}' \rightarrow^* \mathcal{M}$.

We cannot use the technique directly, because the order we have defined is not a wqo. A quasi-order is wqo if every sequence of states has "comparable" elements. Our order would be a wqo if, for every sequence of markings $(\mathcal{M}_i)_{i=0}^\infty$ there were indices $i < j$ such that $\mathcal{M}_i \sqsubseteq_\emptyset \mathcal{M}_j$. However, if $M$ is a component such that $M \not\to^* \emptyset$ then the sequence $\{M\}$, $\{M, M\}$, $\{M, M, M\}$, ... does not satisfy that condition.

However, the order is enough to decide reachability. The general technique is based on a backwards reachability algorithm. The property of wqo assures termination of the algorithm. We will prove that, even though our order does not satisfy the property, the algorithm still halts.

In general, given a set $J$, we denote by $\mathcal{C}(J)$ the upward closed set induced by $J$, that is, $\mathcal{C}(J) = \{\mathcal{M} \mid \mathcal{M}' \sqsubseteq_\emptyset \mathcal{M},\ \mathcal{M}' \in J\}$. Moreover, let us denote by $\mathit{pre}(\mathcal{M}) = \{\mathcal{M}' \mid \mathcal{M}' \mapsto \mathcal{M}\}$ and extend it pointwise to sets of markings.

Then, let us take the following sequence of sets:

- $I_0 = \mathcal{C}(\{\mathcal{M}_f\})$,
- $I_{n+1} = I_0 \cup \mathit{pre}(I_n)$.

This produces an increasing chain $I_0 \subseteq I_1 \subseteq I_2 \subseteq \cdots$. For each $n \geq 0$, $I_n$ is the set of markings from which the final marking can be covered in at most $n$ steps. Therefore, $\mathcal{M}_f$ can be covered if and only if $\mathcal{M}_0$ is in some of those sets, which by the previous comments amounts to saying that $\mathcal{M}_f$ can be reached if and only if $\mathcal{M}_0 \in I_i$ for some $i \geq 0$. Now, in order to obtain an effective algorithm, we need a way to represent and compute those infinite sets, together with a guarantee of termination, namely that the previous chain stabilizes.

In the first place, since $\mapsto$ is a congruence with respect to multiset inclusion, the following is a straightforward result.

**Lemma 6.** *The relation $\mapsto$ is monotonic with respect to $\sqsubseteq_\emptyset$.*

This tells us that every set $I_i$ is an ideal, that is, an upward closed set. We need a finite representation of those $I_i$. For that purpose, given an ideal $I$ we define $\min(I)$ as the set satisfying:

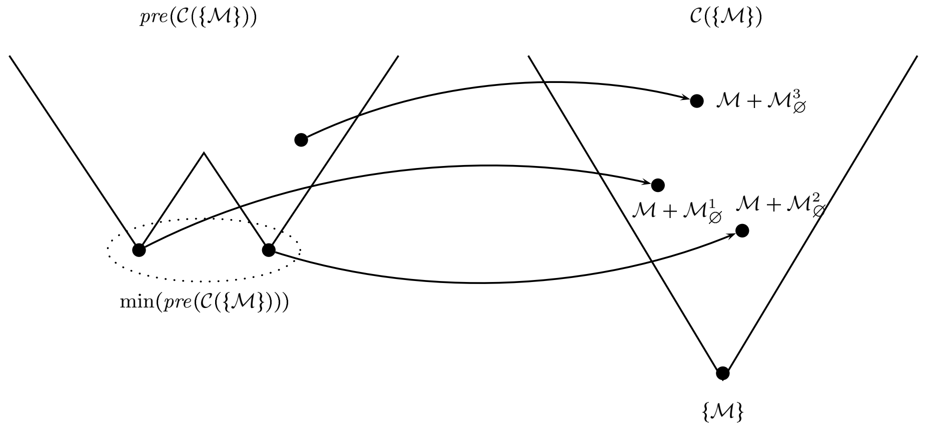
1. $\min(I) \subseteq I$,
2. If $\mathcal{M}, \mathcal{M}' \in \min(I)$ and $\mathcal{M} \sqsubseteq_\emptyset \mathcal{M}'$ then $\mathcal{M} = \mathcal{M}'$,
3. For every $\mathcal{M} \in I$ there is $\mathcal{M}' \in \min(I)$ such that $\mathcal{M}' \sqsubseteq_\emptyset \mathcal{M}$.

It is straightforward to see that $\min(I)$ is well defined, because there is only one set that can satisfy the previous conditions (essentially, thanks to the antisymmetry of $\sqsubseteq_\emptyset$). If $\sqsubseteq_\emptyset$ were a wqo, then we would immediately know that $\min(I)$ is finite for every ideal $I$. However, as we said before, it is not a wqo. Actually, for $J = \{\{M\}, \{M, M\}, \{M, M, M\}, \ldots\}$ with $M \not\to^* \emptyset$, $\min(\mathcal{C}(J)) = J$, which is not finite.

Nevertheless, we can prove that it is finite for every $I_i$. Let us first see the following lemmas.

**Lemma 7.** *Given $M$, $M'$ and $\mathcal{M}$, the set of all minimal $\mathcal{M}_\emptyset$ (with respect to multiset inclusion) such that $M \xrightarrow{\mathcal{M}+\mathcal{M}_\emptyset} M'$ and $\mathcal{M}_\emptyset \to^* \emptyset$ is computable.*

*Proof.* Very similar to the proof of Lemma 3.



$pre(\mathcal{C}(\{\mathcal{M}\}))$
$\mathcal{C}(\{\mathcal{M}\})$
$\mathcal{M}+\mathcal{M}_\emptyset^3$
$\mathcal{M}+\mathcal{M}_\emptyset^1$
$\mathcal{M}+\mathcal{M}_\emptyset^2$
$\min(pre(\mathcal{C}(\{\mathcal{M}\})))$
$\{\mathcal{M}\}$

**Fig. 5.** Computation of $\min(pre(\mathcal{C}(\{\mathcal{M}\})))$

Now we see how we can compute the predecessor function.

**Lemma 8.** *For every* $\mathcal{M}$*, the set* $\min(pre(\mathcal{C}(\{\mathcal{M}\})))$ *is finite and computable.*

*Proof.* For all $\mathcal{M}' + \{M_f\} \subseteq \mathcal{M}$ with $M_f \in S(\mathcal{M}_f) \cup \{\emptyset\}$, let us consider all the steps in which some component evolves to $M_f$, creating on its way at least the components in $\mathcal{M}'$, together with some others, that can necessarily evolve to the empty marking.

For all $M \in \mathcal{R}$, let us see if $M$ can be that component. If $M \xrightarrow{\mathcal{M}'^{\geq}} M_f$ then, by Lemma 7, we can compute all minimal $\mathcal{M}_\emptyset$ such that $M \xrightarrow{\mathcal{M}'+\mathcal{M}_\emptyset} M_f$ and $\mathcal{M}_\emptyset \rightarrow^* \emptyset$. Then we can add to the set of predecessors the marking $\mathcal{M} - \mathcal{M}' - \{M_f\} + \{M\}$. Notice that thanks to Lemma 5 it is enough to consider only minimal markings $\mathcal{M}_\emptyset$.

We have described a finite procedure, yielding finitely many markings in the set of predecessors. This finite set could be not minimal, but we can always minimize this finite set to compute the set we are interested in.

Fig. 5 can give you some insight about the proof of the previous result. A marking $\mathcal{M}$ induces an upwards closed set, the cone in the right handside of Fig. 5. We want to compute (a finite representation of) the set of the predecessors of the markings in that cone, that have the form $\mathcal{M} + \mathcal{M}_\emptyset$ with $\mathcal{M} \rightarrow^* \emptyset$. The proof of the previous result factorises (thanks to Lemma 7) all the ways in which such markings can be reached, yielding finitely many markings $\mathcal{M}'$ such that $\mathcal{M}' \rightarrow \mathcal{M} + \mathcal{M}_\emptyset$. Therefore, every marking $\mathcal{M}' + \mathcal{M}'_\emptyset$ in the left handside cones can reach in one step the cone in the right.

**Proposition 3.** *For every* $i \geq 0$, $\min(I_i)$ *is finite and computable.*

*Proof.* We proceed by induction on $i$. Trivially, $\min(I_0) = \min(\mathcal{C}(\{\mathcal{M}_f\})) = \{\mathcal{M}_f\}$. Let us suppose that $\min(I_i)$ is finite and computable and let us see that $\min(I_{i+1})$ is also finite and computable.

We need to compute $\min(I_{i+1})$, which can be done as in [3] by computing

$$\min\Big(\min(I_0) \cup \bigcup_{\mathcal{M}\in\min(I_i)} \min(pre(\mathcal{C}(\{\mathcal{M}\})))\Big)$$

The induction hypothesis tell us that $\min(I_i)$ is finite, so that the union is a finite one. Moreover, the previous result tells us that each $\min(pre(\mathcal{C}(\{\mathcal{M}\})))$ is finite and computable, so that we can conclude.

Therefore, we can compute the sequence

$$\min(I_0), \min(I_1), \min(I_2), \min(I_3), \cdots$$

This sequence can be used to represent the previous chain of ideals because for every ideal, $\mathcal{C}(\min(I)) = I$. Now let us see that the latter stabilizes, and therefore, also the former.

**Proposition 4.** *There is $k$ such that for every $j \geq k$, $I_k = I_j$.*

*Proof.* Let us suppose that it does not stabilize, that is, $I_0 \subset I_1 \subset I_2 \subset I_3 \subset \cdots$ Then, for every $i \geq 1$ there is $\mathcal{M}_i \in I_i \setminus I_{i-1}$. By construction, since every $I_i$ is upward closed, for every $i < j$ $\mathcal{M}_i \not\sqsubseteq_\emptyset \mathcal{M}_j$. Since the multiset inclusion order is a wqo, there is a subsequence (that we denote in the same way, for clarity of notations) such that $\mathcal{M}_1 \subset \mathcal{M}_2 \subset \mathcal{M}_3 \subset \cdots$

Then, for every $i > 1$, $\overline{\mathcal{M}}_i = \mathcal{M}_i - \mathcal{M}_{i-1}$ is such that $\overline{\mathcal{M}}_i \not\rightarrow^* \emptyset$ (otherwise, we would have $\mathcal{M}_{i-1} \sqsubseteq_\emptyset \mathcal{M}_i$). In that case, every marking reachable from $\overline{\mathcal{M}}_i$ has at least one component. Therefore, since $\mathcal{M}_n = \mathcal{M}_0 + \overline{\mathcal{M}}_1 + \ldots + \overline{\mathcal{M}}_n$, every marking reachable from $\mathcal{M}_n$ has at least $n$ components, which for $n$ greater than $|\mathcal{M}_f|$ is a contradiction, because from every $\mathcal{M}_n$ the marking $\mathcal{M}_f$ should be reachable.

Then, in order to decide reachability we must compute the previous sequence (its finite representation) until it stabilizes in an ideal $I_k$, and see whether $\mathcal{M}_0 \in I_k = \mathcal{C}(\min(I_k))$, that is, if $\mathcal{M} \sqsubseteq_\emptyset \mathcal{M}_0$ for some $\mathcal{M} \in \min(I_k)$.

**Corollary 2.** *The coverability notion induced by $\sqsubseteq_\emptyset$ in $l(N)$ is decidable.*

Since reachability and coverability (induced by $\sqsubseteq_\emptyset$) are equivalent, and Prop. 1 holds, we have the result we were looking for.

**Proposition 5.** *Reachability for g-RN systems without synchronizations is decidable.*

As we have said, we have proved decidability of reachability by following the technique proposed in [3], although our order is not a wqo. However, in a strict sense, we are not extending the technique. Actually, we could work with an order similar to $\sqsubseteq_\emptyset$ that were a wqo. We can classify components in $\mathcal{R}$ in those that perpetuate their offspring (that is, those $M$ such that $\{M\} \not\mapsto \emptyset$) and those that do not (that is, those $M$ such that $\{M\} \mapsto \emptyset$). Let us denote by $\mathcal{P} \subseteq \mathcal{R}$

the set of those that cannot evolve to the empty marking. As we have seen in the proof of Prop. 4, any marking containing $n$ components in $\mathcal{P}$ can only evolve to markings with at least $n$ components. Thus, any marking with more than $n = |\mathcal{M}_f|$ components in $\mathcal{P}$ cannot reach $\mathcal{M}_f$. If we denote by $\mathcal{P}_n$ the set of markings with more than $n$ components in $\mathcal{P}$, then we could have defined $\sqsubseteq'_\emptyset = \sqsubseteq_\emptyset \cup \mathcal{P}_n \times \mathcal{P}_n$. Intuitively, we are identifying all the markings in $\mathcal{P}_n$. The order $\sqsubseteq'_\emptyset$ is a wqo, and it induces the same coverability problem as $\sqsubseteq_\emptyset$. However, we have preferred to present our proofs without that technicality, thus keeping them more intuitive.

We can obtain an analogous result for $\nu$-APNs thanks to the equivalence between g-RN systems and $\nu$-APNs proved in [19]. g-RN systems can simulate $\nu$-APNs, in the sense that for every $\nu$-APN $N$ there is a g-RN system $F(N)$ such that the transitions systems generated by $N$ and $F(N)$ are isomorphic, and that isomorphism is monotonic, so that reachability and coverability are both preserved. Moreover, $F$ itself is an isomorphism. The simulation consists on considering a different component to represent each different name. When different names can interact in the firing of a transition, in the simulation different components synchronize. When all the variables adjacent to a transition $t$ are the same (that is, $|\mathit{Var}(t)| = 1$), then only one name is involved in its firing. If we denote by $\nu_=$-APNs the subclass of $\nu$-APNs such that every transition $t$ satisfies $|\mathit{Var}(t)| = 1$, or $|\mathit{Var}(t)| = 2$ with $\nu \in \mathit{Var}(t)$, it is straightforward to see that $\nu_=$-APNs are the counterpart of g-RN systems without synchronizations.

**Proposition 6.** *If $N$ is a $\nu_=$-APN then $F(N)$ is a g-RN system without synchronizations.*

**Corollary 3.** *Reachability is decidable for the class of $\nu_=$-APNs.*

For each g-RN system without synchronizations, we have defined a multiset rewriting system $l(N)$. This rewrite system is not a P/T net, though we are rewriting multisets in a monotonic way. The reason is that the rewritings are conditional ones, where the condition is reachability in an ordinary Petri net. This reminds of Recursive Petri Nets (RPN) [12]. RPNs have two special types of transitions: abstract and final transitions. The firing of abstract transitions is not atomic. They remove tokens from their preconditions, but instead of adding tokens to postconditions they create a new thread, starting on a marking associated to the transition. Tokens are added to postconditions when the child thread finishes, which happens when it fires a final transition.

We could try to simulate $l(N)$ by using an RPN, immediately obtaining the decidability of reachability as a corollary of the analogous result for RPNs [12]. For each rule

$$\frac{M \xrightarrow{\mathcal{M}} M'}{\{M\} \mapsto \mathcal{M} + \{M'\}}$$

we could consider creating a child thread starting in $M$. However, this "simulation" would not be correct for two reasons. In the first place, we are interested in reaching marking $M'$, but the capability of firing a final transition (the ending condition of a child thread in RPNs) is a coverability condition.

The second reason is that, though we are writing a single rule for all such $M$, $M'$ and $\mathcal{M}$, although $M$ and $M'$ are taken from a finite set, $\mathcal{M}$ is taken from the infinite set $\mathcal{MS}(\mathcal{R})$. We could use a different transition for every two components $M$ and $M'$, but not for every $\mathcal{M}'$. The technique of considering only minimal such markings would not be valid in this case either. Therefore, the simulation using RPNs (or a similar model) must use a single transition for every $\mathcal{M}$ such that $M \xrightarrow{\mathcal{M}} M'$. The most intuitive way that we can think of to achieve it, is to allow child threads to communicate some results to their parent thread. In this way, if the child thread communicates how many times a transition has been fired (that, of course, can be controlled by the number of tokens in some special places) we would have a faithful simulation of the application of the rewrite rule. However, we think that any general model (allowing synchronization) with these features has undecidable reachability.

## 5 Decidability of Coverability for $\nu$-RN Systems with Restricted Communication

In the previous section we have restricted g-RN systems, for which reachability is undecidable, in order to keep its decidability. Our goal now is to do the same thing for $\nu$-RN systems (Petri Nets extended simultaneously both with names and replication). As we proved in [19], $\nu$-RN systems are Turing complete and, in particular, coverability is undecidable for them.

We could think that we also need to forbid synchronizations in order to keep decidability of coverability for $\nu$-RN systems. However, as we prove next, it is enough to restrict communications. A communication happens whenever there is a variable labelling an output arc of a transition, and an input arc of a different compatible transition (see Fig. 6).

In a first approach, we will forbid all name communications between different components. Therefore, components will still be able to synchronize between them, as long as no name moves from one component to another.

The natural order in $\nu$-RN systems, that induces coverability, is defined by $\mathcal{M}_1 = \{M_1, \ldots, M_n\} \sqsubseteq \mathcal{M}_2 = \{M'_1, \ldots, M'_m\}$ if there are two injections $h : \{1, \ldots, n\} \to \{1, \ldots, m\}$ and $\iota : Id \to Id$ such that for every $i \in \{1, \ldots, n\}$, $\iota(M_i(p)) \subseteq M'_{h(i)}(p)$ for all $p$. $h$ has the role of mapping components of $\mathcal{M}_1$ to components of $\mathcal{M}_2$, while $\iota$ maps names in $\mathcal{M}_1$ to names in $\mathcal{M}_2$.

Let us denote by $\sqsubseteq\!\!\!\!\equiv$ the multiset order induced by $\sqsubseteq_\alpha$. According to the definition of multiset order, $\{M_1, \ldots, M_n\} \sqsubseteq\!\!\!\!\equiv \{M'_1, \ldots, M'_m\}$ if there is an injection $h : \{1, \ldots, n\} \to \{1, \ldots, m\}$ such that $M_i \sqsubseteq_\alpha M'_{h(i)}$. In this case, for each $i$

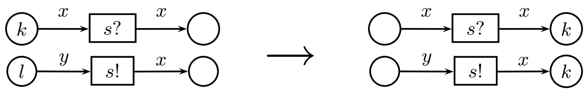


**Fig. 6.** A simple $\nu$-RN system with communication

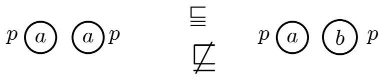


**Fig. 7.** $\nu$-RNs related by $\sqsubseteq\!\!\!\!\equiv$ but not by $\sqsubseteq$

there is an injection $\iota_i$ such that $\iota_i(M_i) \subseteq M'_{h(i)}$. Notice that in the case of $\sqsubseteq$, the mapping $\iota$ that renames names must be the same for all the components, and now we are allowing different mappings $\iota_i$. In other words, for $\sqsubseteq$ names are global, but for $\sqsubseteq\!\!\!\!\equiv$, they are local to components.

The orders $\sqsubseteq$ and $\sqsubseteq\!\!\!\!\equiv$ are different. Indeed, for the simple case in which the net has a single places $p$, it is enough to consider $\mathcal{M} = \{M_1, M_2\}$ and $\mathcal{M}' = \{M'_1, M'_2\}$ with $M_1(p) = M_2(p) = M'_1(p) = \{a\}$ and $M'_2(p) = \{b\}$. Clearly, they satisfy $\mathcal{M}_1 \sqsubseteq\!\!\!\!\equiv \mathcal{M}_2$, but not $\mathcal{M}_1 \sqsubseteq \mathcal{M}_2$ (see Fig. 7). However, we can prove the following relation between them.

**Proposition 7.** *Let $\mathcal{M} = \{M_1, \ldots, M_n\}$ and $\mathcal{M}' = \{M'_1, \ldots, M'_m\}$ be two markings of a $\nu$-RN system.*

- *If $\mathcal{M} \sqsubseteq \mathcal{M}'$ then $\mathcal{M} \sqsubseteq\!\!\!\!\equiv \mathcal{M}'$.*
- *If $S(M_i) \cap S(M_j) = S(M'_i) \cap S(M'_j) = \emptyset \;\; \forall i \neq j$ then $\mathcal{M} \sqsubseteq \mathcal{M}' \Leftrightarrow \mathcal{M} \sqsubseteq\!\!\!\!\equiv \mathcal{M}'$.*

*Proof*

- If $\mathcal{M} \sqsubseteq \mathcal{M}'$ then there are two injections $h : \{1, \ldots, n\} \to \{1, \ldots, m\}$ and $\iota : Id \to Id$ such that $\iota(M_i(p)) \sqsubseteq M'_{h(i)}(p)$ for every $p$ and for all $i \in \{1, \ldots, n\}$. In particular, by definition of $\sqsubseteq_\alpha$ we have that for each $i$, $M_i \sqsubseteq_\alpha M'_{h(i)}$. By definition of multiset order we can conclude that $\mathcal{M} \sqsubseteq\!\!\!\!\equiv \mathcal{M}'$.
- Thanks to the previous item, it is enough to prove that if $\mathcal{M} \sqsubseteq\!\!\!\!\equiv \mathcal{M}'$ then $\mathcal{M} \sqsubseteq \mathcal{M}'$. By definition of $\sqsubseteq\!\!\!\!\equiv$ there is an injection $h : \{1, \ldots, n\} \to \{1, \ldots, m\}$ such that for all $i$, $M_i \sqsubseteq_\alpha M'_{h(i)}$. By definition of $\sqsubseteq_\alpha$, for each $i$ there is an injection $\iota_i : S(M_i) \to S(M_{h(i)})$ such that $\iota_i(M_i(p)) \subseteq M'_{h(i)}(p)$ for all $p$. Since we are assuming that all the components in $\mathcal{M}$ have disjoint namespaces (so that the domains of the $\iota_i$s are disjoint), we can safely define $\iota : Id \to Id$ by $\iota(a) = \iota_i(a)$ whenever $a \in S(M_i)$, which is an injection (in its domain) because each $\iota_i$ is injective and the components in $\mathcal{M}'$ also have disjoint namespaces. Then we have that $\iota(M_i(p)) = \iota_i(M_i(p)) \subseteq M'_{h(i)}(p)$ for all $p$, and we can conclude that $\mathcal{M} \sqsubseteq \mathcal{M}'$.

We saw that the converse of the previous result cannot be true. Since $\sqsubseteq$ considers names to be global, while $\sqsubseteq\!\!\!\!\equiv$ considers them to be local, we can informally state the previous result as follows: If each component has its own namespace, then global and local names are the same thing. Components that cannot communicate have its own namespace, if this is true in the initial marking. We call $\nu$-lRN systems to this class of $\nu$-RN systems, though we omit their formal definition due to lack of space, as well as the details of the following result.

**Proposition 8.** *Coverability is decidable for $\nu$-lRN systems.*

*Proof (sketch).* Once again, we use the wqo technique, that is, that $\nu$-RN systems without communications are well structured systems. In the first place, we have to see that $\sqsubseteq$ is a wqo. Since $\sqsubseteqq$ is a wqo (it is the multiset order induced by a wqo), thanks to the previous result it is enough to prove that all reachable markings have disjoint namespaces. Since components do not communicate names, if they initially have disjoint namespaces then they will always have disjoint namespaces. Therefore, for every reachable marking the orders $\sqsubseteq$ and $\sqsubseteqq$ are the same, and therefore, $\sqsubseteq$ (which induces coverability) is a wqo.

The proofs of monotonicity and computable predecessors are similar to the analogous ones for $\nu$-APNs, which can be seen in [18].

Notice that, unlike in the previous section, where we forbid all synchronizations between components, now we are only forbidding communications between them. This means that several components can synchronize, as long as they are anonymous synchronizations (that is, a component can synchronize with any component that is willing to do so, and the result of that synchronization is the same whichever that component was).

We could also allow a finite amount of names in a common namespace without affecting the decidability result. Let $Id_c$ be the finite set of names allowed in the common namespace. If all the names appearing in more than one component in the initial marking is taken from $Id_c$, and communications are forced to happen with names in $Id_c$, then the previous decidability result can be easily extended to cope with this finite amount of names. If we denote by $\nu$-RN($Id_c$) this class of $\nu$-RN systems, we have the following result, only sketched due to lack of space.

**Proposition 9.** *Coverability is decidable for $\nu$-RN($Id_c$) systems for $Id_c$ finite.*

*Proof (sketch).* Given a $\nu$-RN($Id_c$) system $N$, it is straightforward to build the $\nu$-lRN system $N^*$ that behaves as $N$. $N^*$ should have a storage of the names in $Id_c$ and replace communications with synchronizations. In that case, the names of $Id_c$ in each component can be safely renamed so that they satisfy the restrictions of $\nu$-lRN systems. Then, we can conclude thanks to the previous result.

## 6 Conclusions and Future Work

We have restricted the models of g-RN systems and $\nu$-RN systems presented in [19] to obtain decidability results that do not hold in the unrestricted models. More precisely, reachability, which is undecidable for g-RN systems, has been proved to be decidable in the subclass of g-RN systems in which we do not allow synchronizations between the different components that compose a system.

This decidability result is interesting in itself. Moreover, the proof has been carried out by reducing the problem to reachability in a multiset rewriting system with conditional rules, in which the conditions are reachability problems in ordinary P/T nets. As we mentioned at the end of Sect. 4, the rewritings systems

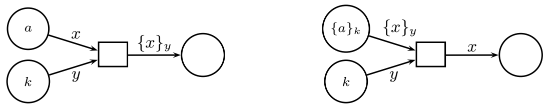


**Fig. 8.** Ciphering and deciphering

$l(N)$ that we have used are quite similar to the model of Recursive Petri Nets (RPN), thus bringing close two apparently quite different models. However, it seems that RPNs are not enough to capture the behavior of $l(N)$. We have used these rewrite systems only as a technicality for our purposes, but perhaps it would be interesting to study which is the minimal extension (or modification) of RPNs that suffices to capture the behavior of $l(N)$, and so that reachability is still decidable. As we said, the main difference is that child threads should have some *result places*, associated to other places in the father thread, so that when the former finished, the latter could receive the results obtained by its child. In our setting, the result places would be places added in an ad hoc way, that count how many times each of the replicating transitions have been fired.

Many of the models described in this paper and in [19] are well structured, for which coverability is decidable. Since in most of them reachability is undecidable, we need a finer way to compare the expressive power of these models. In [4] a comparison between well-structured systems is done. The comparison criterion is weak trace equivalence, with coverability as accepting condition for traces. We plan to place $\nu$-APNs and the related models that appear in this paper inside the hierarchy obtained in [4]. For instance, it seems very intuitive that Lossy Channel Systems [2] are incomparable to $\nu$-APNs, because $\nu$-APNs cannot have a FIFO-like behavior.

The comparison is achieved by seeing those systems as subclasses of MSR(C) [8], which is a model based on multiset rewriting. Therefore, it would also be interesting to see how the rewrite systems $l(N)$ fit inside the hierarchy.

As we have said, in [19] we proved Turing completeness of $\nu$-RN systems, together with a way to map name creation to replication and viceversa. We plan to study how this mapping can be achieved when we are dealing with tuples of pure names as tuples.

It would be interesting to see if we can use pairs in a restricted way without reaching Turing-completeness. Moreover, pairs can be used to represent cryptographic primitives. For instance, we can write $(a, k)$ as $\{a\}_k$ to represent the message $a$ ciphered under key $k$. Then, transitions like the ones in Figure 8 simulate the ciphering and deciphering of $\{a\}_k$. Following the previous ideas, it seems that when each key can only be used to cipher finitely many messages, then the $\nu$-RN system counterpart is actually a $\nu$-RN($Id_c$), so that coverability would remain decidable.

## Acknowledgments

The authors would like to thank the anonymous referees for their valuable comments.

## References

[1] Abadi, M., Gordon, A.D.: A Calculus for Cryptographic Protocols: The Spi Calculus. In: Fourth ACM Conference on Computer and Communications Security, pp. 36–47. ACM Press, New York (1997)

[2] Abdulla, P.A., Jonsson, B.: Verifying Programs with Unreliable Channels. Information and Computation 127(2), 91–101 (1996)

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

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

[5] Cardelli, L., Gordon, A.D.: Mobile Ambients. In: Nivat, M. (ed.) FOSSACS 1998. LNCS, vol. 1378, pp. 140–155. Springer, Heidelberg (1998)

[6] Decker, G., Puhlmann, F.: Extending BPMN for Modelling Complex Choreographies. In: Meersman, R., Tari, Z. (eds.) OTM 2007, Part I. LNCS, vol. 4803, pp. 24–40. Springer, Heidelberg (2007)

[7] Decker, G., Weske, M.: Instance Isolation Analysis for Service-Oriented Architectures. In: Proceedings of the 2008 IEEE International Conference on Services (SCC 2008), pp. 249–256. IEEE Computer Society, Los Alamitos (2008)

[8] Delzanno, G.: An overview of MSR(C): A CLP-based Framework for the Symbolic Verification of Parameterized Concurrent Systems. In: 11th Int. Workshop on Functional and Logic Programming, WFLP 2002. Electronic Notes in Theoretical Computer Science, vol. 76. Elsevier, Amsterdam (2002)

[9] Esparza, J., Nielsen, M.: Decidability issues for Petri Nets - a survey. Bulletin of the EATCS 52, 244–262 (1994)

[10] Finkel, A., Schnoebelen, P.: Well-Structured Transition Systems Everywhere! Theoretical Computer Science 256(1-2), 63–92 (2001)

[11] Gordon, A.: Notes on Nominal Calculi for Security and Mobility. In: Focardi, R., Gorrieri, R. (eds.) FOSAD 2000. LNCS, vol. 2171, pp. 262–330. Springer, Heidelberg (2001)

[12] Haddad, S., Poitrenaud, D.: Recursive Petri Nets. Acta Informatica 44(7-8), 463–508 (2007)

[13] Kummer, O.: Undecidability in object-oriented Petri nets. Petri Net Newsletter 59, 18–23 (2000)

[14] Lazic, R., Newcomb, T.C., Ouaknine, J., Roscoe, A.W., Worrell, J.: Nets with Tokens Which Carry Data. Fundamenta Informaticae 88(3), 251–274 (2008)

[15] Lomazova, I.: Nested Petri nets - a formalism for specification and verification of multi-agent distributed systems. Fundamenta Informaticae 43(1-4), 195–214 (2000)

[16] Milner, R., Parrow, J., Walker, D.: A Calculus of Mobile Processes, I. Information and Computation 100(1), 1–40 (1992)

[17] Needham, R.M.: Names. Distributed Systems, pp. 89–101. Addison-Wesley, Reading (1989)

[18] Rosa-Velardo, F., de Frutos-Escrig, D., Marroquín-Alonso, O.: On the expressiveness of Mobile Synchronizing Petri Nets. In: 3rd Int. Workshop on Security Issues in Concurrency, SecCo 2005. ENTCS, vol. 180(1), pp. 77–94. Elsevier, Amsterdam (2007)

[19] Rosa-Velardo, F., de Frutos-Escrig, D.: Name Creation vs. Replication in Petri Net Systems. Fundamenta Informaticae 88(3), 329–356 (2008); special issue on Selected Papers from ATPN 2007. IOS Press (2008)

# Pomset Languages of Finite Step Transition Systems

Jean Fanchon[1,2] and Rémi Morin[3,4,⋆]

[1] CNRS, LAAS, 7 avenue du colonel Roche, F-31077 Toulouse, France
[2] Université de Toulouse, UPS, INSA, INP, ISAE, LAAS, F-31077 Toulouse, France
[3] Laboratoire d'Informatique Fondamentale de Marseille, LIF, CNRS, UMR 6166
[4] Aix-Marseille Université, 163, avenue de Luminy, Case 901 F-13288 Marseille, France

**Abstract.** Step transition systems form a powerful model to describe the concurrent behaviors of distributed or parallel systems. They offer also a general framework for the study of marking graphs of Petri nets [22]. In this paper we investigate a natural labeled partial order semantics for step transition systems. As opposed to [19] we allow for autoconcurrency by considering steps that are multisets of actions. First we prove that the languages of step transition systems are precisely the width-bounded languages that are *step-closed* and *quasi-consistent*. Extending results from [19] we focus next on *finite* step transition systems and characterize their languages in the line of Buchi's theorem. Our main result present six equivalent conditions in terms of *regularity* and *MSO-definability* for a set of labeled partial orders to be recognized by some finite step transition system.

## 1 Introduction

Partially ordered multisets, also called pomsets, labeled partial orders or partial words, constitute one of the most basic models of concurrency [23]. Process algebras like CCS and TCSP, and system models such as Petri nets have been given pomset semantics for many years, and several pomset algebras have been designed. Nevertheless the study of pomset languages from the point of view of recognizability, regularity or logical definability still offers many interesting problems to investigate.

One of the most important and usefull results in language theory is a theorem due to Büchi [3] which states the equivalence between the definability of a set of words by a formula of MSO logic and its algebraic recognizability by a monoid morphism, or equivalently its recognizability by a finite word automaton. This result has been widely used in model checking: Most temporal logics, in particular LTL, are subsumed by MSO logic, and effective translations from LTL formulae to automata have been designed. The equivalence between MSO logic and algebraic recognizability has also been proved for trees [7,24]. As a consequence, similarly to words, definable sets of trees are characterized by tree automata which provides an effective decision procedure.

A key difficulty one encounters in the case of pomsets as opposed to words is the lack of a finite set of operators generating all of them [12]. However various restricted frameworks have been defined and investigated with an algebraic approach, and in many

⋆ Supported by the ANR project CHOCO.

cases, Kleene-like and Büchi-like results have been established. In particular the equivalence between MSO definability and algebraic recognizability has been proved for subsets of Mazurkiewicz traces [5,25], finitely generated subsets of message sequence charts [21], the algebra of series-parallel pomsets [20,18], the concurrency monoids of stably concurrent automata [8], etc.

For larger classes of pomsets, as an alternative approach of recognizability, different accepting finite devices for pomset languages have been investigated such as, e.g., graph acceptors applied to pomsets [25], asynchronous automata applied to restricted classes of pomsets without autoconcurrency [9], and message-passing automata [2]. In these works the recognizable languages have no characterization in an algebraic framework.

In this paper we investigate a partial order semantics for step transition systems with autoconcurrency. The latter are simply automata where transitions carry multisets of actions. As shown by [22] this model is a nice framework for the study the marking graphs of Place/Transition Petri nets. We introduce first a new and somewhat abstract way to define the pomsets accepted by some step transition system. We show however that this new approach coincides with the notion of firing pomset from [26,19]. This connection leads us to a characterization of the pomset languages that are recognized by step transition systems which is much simpler than the one obtained in [19] when no autoconcurrency occurs: More precisely we prove that a pomset language is recognized by some deterministic step transition system if and only if it is *step-closed* and *quasi-consistent* (Theorem 15).

Next we focus on *finite* deterministic step transition systems. In Section 2 we establish first a connection with the notion of *regularity* from [11]: A step-closed and quasi-consistent pomset language is regular if and only if it is recognized by some finite step transition system (Theorem 22). Then we introduce in Section 3 the notion of MSO definable sets and present various equivalences in the line of Buchi's theorem. Our main result gives several characterizations of the step-closed and quasi-consistent languages that can be recognized by some finite deterministic step transition system (Theorem 48). In particular we extend to the setting of autoconcurrency the main result from [19]: A pomset language that is step-closed and quasi-consistent is recognized by some finite deterministic step transition system if and only if *its basis is definable in MSO logic and also prime-bounded*, a notion borrowed from [17]. This relies essentially on two technical ingredients: We show how to generalize to the setting of autoconcurrency the fact that pomset languages of finite deterministic step transition systems are prime-bounded; we explain also how to apply the technique of chain partitions from [4] in order to build an MSO sentence from a regular, quasi-consistent and step-closed pomset language.

***Preliminaries.*** Throughout the paper we fix some *finite* alphabet $\Sigma$. A *labeled partial order* (lpo) over $\Sigma$ is a triple $t = (E, \preccurlyeq, \xi)$ where $(E, \preccurlyeq)$ is a finite partial order and $\xi$ is a mapping from $E$ to $\Sigma$. This structure can be seen as an abstraction of an execution of a concurrent system [5,23]. In this view, the elements $e$ of $E$ are *events* and their label $\xi(e)$ describes the action that is performed in the system by the event $e \in E$. Furthermore, the order $\preccurlyeq$ describes the dependence between events. In particular, if two events are concurrent, they can be executed in any order or even in parallel. A *pomset* is the isomorphic class of an lpo. We denote by $\mathbb{P}(\Sigma)$ the class of all pomsets over

$\Sigma$. A pomset $t = (E, \preccurlyeq, \xi)$ is *without autoconcurrency* if no action can be performed concurrently with itself, that is: $\xi(x) = \xi(y)$ implies $x \preccurlyeq y$ or $y \preccurlyeq x$ for all $x, y \in E$.

An *order extension* of a pomset $t = (E, \preccurlyeq, \xi)$ is a pomset $t' = (E, \preccurlyeq', \xi)$ such that $\preccurlyeq \subseteq \preccurlyeq'$. We denote by $\mathrm{OE}(t)$ the set of order extensions of $t$ and we put $\mathrm{OE}(\mathcal{L}) = \bigcup_{t \in \mathcal{L}} \mathrm{OE}(t)$ for any pomset language $\mathcal{L} \subseteq \mathbb{P}(\Sigma)$. Clearly $\mathcal{L} \subseteq \mathrm{OE}(\mathcal{L})$. We say that a pomset language is *weak* if it is closed for order extensions, that is, if $\mathcal{L} = \mathrm{OE}(\mathcal{L})$. A *linear extension* of $t$ is an order extension that is linearly ordered. It corresponds to a sequential view of the concurrent execution $t$. Linear extensions of a pomset $t$ over $\Sigma$ can naturally be regarded as words over $\Sigma$. By $\mathrm{LE}(t) \subseteq \Sigma^\star$, we denote the set of linear extensions of a pomset $t$ over $\Sigma$. For any subset of pomsets $\mathcal{L} \subseteq \mathbb{P}(\Sigma)$, we put $\mathrm{LE}(\mathcal{L}) = \bigcup_{t \in \mathcal{L}} \mathrm{LE}(t)$.

Let $t = (E, \preccurlyeq, \xi)$ be a pomset and $x, y \in E$. The elements $x$ and $y$ are *concurrent* or *incomparable* (denoted $x \text{ co } y$) if $\neg(x \preccurlyeq y) \wedge \neg(y \preccurlyeq x)$. Now $y$ *covers* $x$ (denoted $x \prec\!\!\cdot\, y$) if $x \prec y$ and $x \prec z \preccurlyeq y$ implies $z = y$. An *ideal* of a pomset $t = (E, \preccurlyeq, \xi)$ is a subset $H \subseteq E$ such that $x \in H \wedge y \preccurlyeq x \Rightarrow y \in H$. The restriction $t' = (H, \preccurlyeq \cap (H \times H), \xi \cap (H \times \Sigma))$ is then called a *prefix* of $t$ and we write $t' \leqslant t$. For all $z \in E$, we denote by $\downarrow\! z$ the ideal of events below $z$, i.e. $\downarrow\! z = \{y \in E \mid y \preccurlyeq z\}$.

For any pomset language $\mathcal{L} \subseteq \mathbb{P}(\Sigma)$, $\mathrm{Pref}(\mathcal{L})$ denotes the set of prefixes of pomsets from $\mathcal{L}$. Clearly $\mathcal{L} \subseteq \mathrm{Pref}(\mathcal{L})$. The language $\mathcal{L}$ is called *prefix-closed* if $\mathcal{L} = \mathrm{Pref}(\mathcal{L})$. Let $t_1 = (E_1, \preccurlyeq_1, \xi_1)$ be a pomset over $\Sigma$. The *residual* $\mathcal{L} \setminus t_1$ consists of all pomsets $t_2 = (E_2, \preccurlyeq_2, \xi_2)$ such that there exists some pomset $t = (E, \preccurlyeq, \xi)$ in $\mathcal{L}$ satisfying the following conditions:

1. $E = E_1 \cup E_2$, $E_1 \cap E_2 = \emptyset$, and $E_1$ is an ideal of $t$,
2. $t_1$ is the restriction of $t$ to events in $E_1$, and
3. $t_2$ is the restriction of $t$ to events in $E_2$.

The most basic operation on pomsets is certainly the *strong concatenation*. Given two pomsets $t_1 = (E_1, \preccurlyeq_1, \xi_1)$ and $t_2 = (E_2, \preccurlyeq_2, \xi_2)$ over $\Sigma$, we denote by $t_1 \cdot t_2$ the pomset that puts each event of $t_2$ after all events of $t_1$, i.e. $t_1 \cdot t_2 = (E_1 \uplus E_2, \preccurlyeq_1 \cup \preccurlyeq_2 \cup (E_1 \times E_2), \xi_1 \cup \xi_2)$. A *step sequence* is a pomset where the concurrency relation between events is transitive: Any event belongs to a single maximal set of pairwise concurrent events (a step). The set of all step sequences over $\Sigma$ is denoted $\mathbb{S}(\Sigma)$. Thus $\mathbb{S}(\Sigma) = \{(E, \preccurlyeq, \xi) \in \mathbb{P}(\Sigma) \mid \forall e, e', e'' \in E : e \text{ co } e' \wedge e' \text{ co } e'' \wedge e \neq e'' \Rightarrow e \text{ co } e''\}$. Clearly the set of all step sequences is closed for strong concatenation. Pomsets with empty ordering (besides equality), i.e. pomsets $(E, \mathrm{Id}_E, \xi)$, are particular step pomsets. Note that $\mathbb{S}(\Sigma)$ is the submonoid of $\mathbb{P}(\Sigma)$ generated by strong concatenation of pomsets with empty ordering. A subset of step sequences $K \subseteq \mathbb{S}(\Sigma)$ is called a step language. Now a multiset over $\Sigma$ is a mapping $m : \Sigma \to \mathbb{N}$. Given two multisets $m_1$ and $m_2$ we write $m_1 \leqslant m_2$ if $m_1(a) \leqslant m_2(a)$ for all $a \in \Sigma$. In that case $m_2 \setminus m_1$ denote the multiset such that $m_2 \setminus m_1(a) = m_2(a) - m_1(a)$ for each $a$. We denote by $M(\Sigma)$ the set of all multisets over $\Sigma$. Clearly any step sequence can be identified with a sequence of non-empty multisets. Conversely any sequence of multisets $u \in M(\Sigma)^\star$ can be regarded as a step sequence. We shall use these correspondances implicitly at some places in the sequel of this paper.

Let $t$ be a pomset over $\Sigma$. We denote by $\mathrm{SE}(t)$ the set of step extensions $\mathrm{SE}(t) = \mathrm{OE}(t) \cap \mathbb{S}(\Sigma)$. For any pomset language $\mathcal{L} \subseteq \mathbb{P}(\Sigma)$ we put $\mathrm{SE}(\mathcal{L}) = \bigcup_{t \in \mathcal{L}} \mathrm{SE}(t)$. Noteworthy we have $\mathrm{SE}(\mathrm{SE}(\mathcal{L})) = \mathrm{SE}(\mathcal{L})$.

## 2 Pomset Languages of Step Transition Systems

In this section we investigate a pomset semantics for step transition systems which turns out to correspond to the notion of firing pomsets or enabled partial words [26,19].

### 2.1 Step Transition Systems and Their Languages

Step transition systems were used by Mukund [22] in order to extend the synthesis problem of elementary Petri nets [10] to the more general setting of Place/Transition nets. These automata accept not only words, that is: sequences of actions, but also sequences of multisets, such as step firing sequences of a Place/Transition Petri net. We slightly extend this model by providing step transition systems with some accepting states.

**Definition 1.** *A* step transition system *(for short, an STS) over the alphabet $\Sigma$ is a structure $\mathcal{A} = (Q, \imath, \Sigma, \longrightarrow, F)$ where $Q$ is a set of states, $\imath \in Q$ is an initial state, $F \subseteq Q$ is the subset of final states and $\longrightarrow \subseteq Q \times M(\Sigma) \times Q$ is a set of labeled transitions such that*

$\mathsf{STS}_1$*: $\forall q_1, q_2 \in Q$: $q_1 \xrightarrow{\emptyset} q_2 \Leftrightarrow q_1 = q_2$;*
$\mathsf{STS}_2$*: $\forall q_1, q_2 \in Q$, $\forall p' \leqslant p \in M(\Sigma)$: $q_1 \xrightarrow{p} q_2 \Rightarrow \exists q_3 \in Q$, $q_1 \xrightarrow{p'} q_3 \xrightarrow{p \backslash p'} q_2$.*

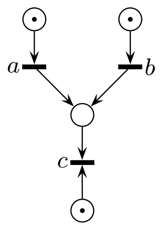
*The step transition system $\mathcal{A}$ is* finite *if $Q$ is finite and moreover the size of all steps in transitions from $\mathcal{A}$ is bounded, i.e. there is a finite number of transitions.*

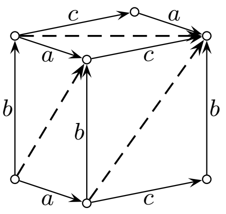
As usual, for any step sequences $u = a_1 ... a_n \in M(\Sigma)^\star$, we write $q \xrightarrow{u} q'$ if there are states $q_0$,..., $q_n$ such that $q_0 = q$, $q_n = q'$ and for each $i \in [1, n]$, $q_{i-1} \xrightarrow{a_i} q_i$. The *step language of* $\mathcal{A}$ $\mathrm{SL}(\mathcal{A}) \subseteq M(\Sigma)^\star$ collects all step sequences $u$ such that $\imath \xrightarrow{u} q_f$ for some $q_f \in F$. Observe that for any transition $q \xrightarrow{u} q'$ and any step sequence $v \in \mathrm{SE}(u)$, we have $q \xrightarrow{v} q'$. This shows that $\mathrm{SE}(\mathrm{SL}(\mathcal{A})) = \mathrm{SL}(\mathcal{A})$.

**Definition 2.** *The pomset language $\mathcal{L}(\mathcal{A})$ of a step transition system $\mathcal{A}$ consists of all pomsets $t \in \mathbb{P}(\Sigma)$ such that* $\mathrm{SE}(t) \subseteq \mathrm{SL}(\mathcal{A})$.

Thus a pomset $t$ is accepted by some step transition system $\mathcal{A}$ if all step extensions of $t$ are step sequences of $\mathcal{A}$. We will show in Theorem 14 that this definition coincides with the approach adopted in [19], when we focus on deterministic step transition systems without autoconcurrency.

**Example 3.** Consider the Petri net from Fig. 1. Its marking graph is the step transition system depicted in Fig. 2 and two of its pomsets are drawn in Fig. 3.





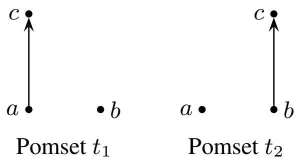


**Fig. 1.** A Petri net... **Fig. 2.** ... its marking graph... **Fig. 3.** ... and two of its firing pomsets

Let $\mathcal{L} \subseteq \mathbb{P}(\Sigma)$. The *step closure* $\langle \mathcal{L} \rangle_{SE} \subseteq \mathbb{P}(\Sigma)$ of $\mathcal{L}$ collects all pomsets whose step extensions are step extensions from $\mathcal{L}$: $\langle \mathcal{L} \rangle_{SE} = \{t \in \mathbb{P}(\Sigma) \mid \mathrm{SE}(t) \subseteq \mathrm{SE}(\mathcal{L})\}$. It is clear that $\langle \mathcal{L} \rangle_{SE} = \langle \mathrm{SE}(\mathcal{L}) \rangle_{SE}$ because $\mathrm{SE}(\mathrm{SE}(\mathcal{L})) = \mathrm{SE}(\mathcal{L})$. Moreover $\mathcal{L} \subseteq \langle \mathcal{L} \rangle_{SE}$. We borrow now from [11] the notion of step-closed languages (which were called *compatible* in [11, Def. 1.5]).

**Definition 4.** *A pomset language $\mathcal{L} \subseteq \mathbb{P}(\Sigma)$ is* step-closed *if $\mathcal{L} = \langle \mathcal{L} \rangle_{SE}$.*

It is easy to check that any step-closed pomset language $\mathcal{L}$ is weak, i.e. $\mathcal{L} = \mathrm{OE}(\mathcal{L})$. Note also that for any step transition system $\mathcal{A}$, we have $\mathcal{L}(\mathcal{A}) = \langle \mathrm{SL}(\mathcal{A}) \rangle_{SE}$ and $\mathrm{SE}(\mathcal{L}(\mathcal{A})) = \mathrm{SL}(\mathcal{A})$. The next result shows that step-closed pomset languages are precisely the languages of step transition systems.

**Proposition 5.** *Let $\mathcal{L} \subseteq \mathbb{P}(\Sigma)$ be a pomset language. Then $\mathcal{L}$ is the pomset language of some step transition system if and only if $\mathcal{L}$ is step-closed.*

**Proof.** If $\mathcal{L}$ is the language of an STS $\mathcal{A}$ then $\mathcal{L} = \langle \mathrm{SL}(\mathcal{A}) \rangle_{SE}$ because $\mathrm{SE}(\mathrm{SL}(\mathcal{A})) = \mathrm{SL}(\mathcal{A})$. It follows that $\mathrm{SE}(\mathcal{L}) = \mathrm{SL}(\mathcal{A})$ hence $\mathcal{L} = \langle \mathrm{SE}(\mathcal{L}) \rangle_{SE} = \langle \mathcal{L} \rangle_{SE}$. Assume now that $\mathcal{L}$ is step-closed. We have $\mathcal{L} = \langle \mathcal{L} \rangle_{SE} = \langle \mathrm{SE}(\mathcal{L}) \rangle_{SE}$. We can build an STS such that $\mathrm{SL}(\mathcal{A}) = \mathrm{SE}(\mathcal{L})$. Then $\mathcal{L}(\mathcal{A}) = \langle \mathrm{SL}(\mathcal{A}) \rangle_{SE} = \langle \mathrm{SE}(\mathcal{L}) \rangle_{SE} = \mathcal{L}$. ■

### 2.2 Pomset Languages of Deterministic Step Transition Systems

In this paper we are mainly interested in properties of pomset languages described by *deterministic* step transition systems. The latter include the marking graph of unlabeled Petri nets.

**Definition 6.** *A step transition system $\mathcal{A}$ is* deterministic *if for all states $q_1, q_2, q_3 \in Q$ and for all steps $m \in M(\Sigma)$ we have: $q_1 \xrightarrow{m} q_2 \wedge q_1 \xrightarrow{m} q_3$ implies $q_2 = q_3$.*

If $\mathcal{A}$ is deterministic then for all states $q, q', q'' \in Q$ and all step sequences $u, v \in M(\Sigma)^\star$ such that $q \xrightarrow{u} q'$ and $q \xrightarrow{v} q''$ we have $\mathrm{LE}(u) \cap \mathrm{LE}(v) \neq \emptyset \Rightarrow q' = q''$.

In order to characterize the pomset languages of *deterministic* step transition systems, we need to introduce the following new condition.

**Definition 7.** *A pomset language $\mathcal{L} \subseteq \mathbb{P}(\Sigma)$ is* quasi-consistent *if for any two prefixes $t, t' \in \mathrm{Pref}(\mathcal{L})$ we have $\mathrm{LE}(t) \cap \mathrm{LE}(t') \neq \emptyset$ implies $\mathcal{L} \setminus t = \mathcal{L} \setminus t'$.*

In other words a pomset language is quasi-consistent if all prefixes that share a common linear extension have a common residual. This condition resembles the definition of a consistent pomset language [1] which requires that $\mathrm{LE}(t) \cap \mathrm{LE}(t') \neq \emptyset$ implies $t = t'$ for any two prefixes of $\mathcal{L}$. However the pomset language of the Petri net from Example 3 is quasi-consistent but not consistent.

The next useful lemma characterizes the subsets of step sequences that are the step languages of deterministic step transition systems.

**Lemma 8.** *Let $\mathcal{L} \subseteq \mathbb{S}(\Sigma)$ be a step language. Then $\mathcal{L}$ is the step language of some deterministic step transition system if and only if $\mathcal{L}$ is quasi-consistent and* $\mathrm{SE}(\mathcal{L}) \subseteq \mathcal{L}$.

**Proof.** Let $\mathcal{L}$ be the step language of a deterministic step transition system $\mathcal{A}$. We have already noticed that $\mathrm{SE}(\mathcal{L}) = \mathcal{L}$. Let $u, v \in \mathrm{Pref}(\mathcal{L})$ and $q, q'$ be the two states such that $\imath \xrightarrow{u} q$ and $\imath \xrightarrow{u} q'$. If $\mathrm{LE}(u) \cap \mathrm{LE}(v) \neq \emptyset$ then $q = q'$ because $\mathcal{A}$ is deterministic. Therefore $\mathcal{L} \setminus u = \mathcal{L} \setminus v$. Thus $\mathcal{L}$ is quasi-consistent.

Assume now that the step language $\mathcal{L} \subseteq \mathbb{S}(\Sigma)$ is quasi-consistent and $\mathrm{SE}(\mathcal{L}) \subseteq \mathcal{L}$. Then $\mathrm{SE}(\mathcal{L}) = \mathcal{L}$. We consider the equivalence relation $\simeq$ over $\mathcal{L}$ such that $u \simeq v$ iff $\mathcal{L} \setminus u = \mathcal{L} \setminus v$. For any pair $u, v \in M(\Sigma)^\star$ and any step $m \in M(\Sigma)$, if $u \simeq v$ then $u \cdot m \simeq v \cdot m$. We denote by $[u]$ the equivalence class of $u$ w.r.t. $\simeq$. We consider the structure $\mathcal{A}$ that consists of the quotient $M(\Sigma)^\star / \simeq$ as set of states, the triples $[u] \xrightarrow{m} [u \cdot m]$ as transition relation, the equivalence class of the empty pomset $[\varepsilon]$ as initial state and the set of equivalence classes $\mathcal{L}/\simeq$ as set of final states. Then $\mathcal{A}$ is a deterministic step transition system: If $m = m_1 \oplus m_2$ then we have $\mathrm{LE}(m) \cap \mathrm{LE}(m_1 \cdot m_2) \neq \emptyset$. Since $\mathcal{L}$ is quasi-consistent, $u \cdot m \simeq u \cdot m_1 \cdot m_2$. Finally it is clear that $\mathrm{SL}(\mathcal{A}) = \mathcal{L}$. ■

**Remark 9.** A step language $\mathcal{L} \subseteq \mathbb{S}(\Sigma)$ satisfies the requirement $\mathrm{SE}(\mathcal{L}) \subseteq \mathcal{L}$ iff for any step sequences $w, w' \in \mathbb{S}(\Sigma)$ and for any multisets $m_1, m_2 \in M(\Sigma)$, we have: $w \cdot (m_1 \oplus m_2) \cdot w' \in \mathcal{L}$ implies $w \cdot m_1 \cdot m_2 \cdot w' \in \mathcal{L}$.

The same criterium of quasi-consistency will be used to characterize the pomset languages of deterministic step transition systems in Theorem 15 —together with the basic requirement that it is step-closed (Prop. 5). Half of this result follows actually from Lemma 8 as expressed by the next statement.

**Corollary 10.** *If a pomset language $\mathcal{L}$ is step-closed and quasi-consistent then it is the pomset language of some deterministic step transition system.*

**Proof.** Let $K = \mathrm{SE}(\mathcal{L})$. We have $\mathrm{SE}(K) = K$ and moreover $\mathcal{L} = \langle K \rangle_{SE}$ because $\mathcal{L}$ is step-closed. By Lemma 8, it is sufficient to show that $K$ is quasi-consistent. Let $u, v \in \mathrm{Pref}(K)$ such that $\mathrm{LE}(u) \cap \mathrm{LE}(v) \neq \emptyset$. Since $\mathcal{L}$ is step-closed, it is weak and thus $K \subseteq \mathcal{L}$. Thus $u, v \in \mathrm{Pref}(\mathcal{L})$ and $\mathcal{L} \setminus u = \mathcal{L} \setminus v$ because $\mathcal{L}$ is quasi-consistent. Let $w \in \mathbb{P}(\Sigma)$. If $w \in K \setminus u$ then $w \in \mathcal{L} \setminus u$ hence $v \cdot w \in \mathcal{L}$ because $\mathcal{L}$ is weak. It follows that $v \cdot w \in K$ because $K = \mathcal{L} \cap \mathbb{S}(\Sigma)$. Thus $K \setminus u \subseteq K \setminus v$. By symmetry we get $K \setminus u = K \setminus v$. Thus $K$ is quasi-consistent. ■

## 2.3 Comparisons to Similar Approaches from the Literature

At this point it is necessary to compare Definition 2 to other pomset semantics from the literature, in particular [19] and [26]. To this aim we use the notion of T-pomsets.

A *T-pomset* $u \cdot m \in \Sigma^\star \cdot M(\Sigma)$ a simply a sequence of actions $u \in \Sigma^\star$ followed by a step $m \in M(\Sigma)$. We denote by $\mathbb{T}(\Sigma)$ the class of all T-pomsets over $\Sigma$. A T-pomset language $\mathcal{L} \subseteq \mathbb{T}(\Sigma)$ is a set of T-pomsets. The set of *T-extensions* of a pomset $t$ collects the order extensions of $t$ that are T-pomsets: $\mathrm{TE}(t) = \mathrm{OE}(t) \cap \mathbb{T}(\Sigma)$. Moreover for any pomset language $\mathcal{L}$ we put $\mathrm{TE}(\mathcal{L}) = \bigcup_{t \in \mathcal{L}} \mathrm{TE}(t)$. Thus we have $\mathrm{TE}(\mathcal{L}) = \mathrm{OE}(\mathcal{L}) \cap \mathbb{T}(\Sigma)$.

Let $\mathcal{A}$ be a fixed *deterministic* step transition system. We consider now the collection of T-pomsets $T(\mathcal{A}) = \mathrm{Pref}(\mathrm{SL}(\mathcal{A})) \cap \mathbb{T}(\Sigma)$. Thus a T-pomset $u \cdot m$ belongs to $T(\mathcal{A})$ if there are three states $q, q', q'' \in Q$ with $q'' \in F$ such that $\imath \xrightarrow{u} q \xrightarrow{m} q' \xrightarrow{v} q''$ for some word $v \in \Sigma^\star$. It is easy to check that if $u \cdot m \in T(\mathcal{A})$, $v \in \Sigma^\star$ and $v_1, v_2 \in \mathrm{LE}(m)$ then we have

$\mathsf{Cpl}_1$: $m' \leqslant m$ implies $u \cdot m' \in T(\mathcal{A})$
$\mathsf{Cpl}_2$: $m' \leqslant m \wedge v \in \mathrm{LE}(m')$ implies $u \cdot v \cdot (m \setminus m') \in T(\mathcal{A})$
$\mathsf{Cpl}_3$: $u \cdot v_1 \cdot v \cdot m' \in T(\mathcal{A})$ implies $u \cdot v_2 \cdot v \cdot m' \in T(\mathcal{A})$

because $\mathcal{A}$ is deterministic. This shows that $T(\mathcal{A})$ corresponds to the notion of a complete local independance relation [19, Def. 1.2]. Now we can borrow the notion of process from [19, Def. 2.3] as follows.

**Definition 11.** *A* process *of the deterministic step transition system $\mathcal{A}$ is a pomset $t = (E, \preccurlyeq, \xi)$ such that for all prefixes $t' = (E', \preccurlyeq', \xi')$ of $t$, and for all linear extensions $u \in \mathrm{LE}(t')$, we have $(u, \xi(\min_{\preccurlyeq}(E \setminus E'))) \in T(\mathcal{A})$.*

*A process of $\mathcal{L}$ is* final *if each of its linear extensions belongs to* $\mathrm{SL}(\mathcal{A})$. *We denote by $\mathfrak{p}(\mathcal{A})$ the class of all processes of $\mathcal{A}$ and by $\mathfrak{p}_f(\mathcal{A})$ the class of all final processes of $\mathcal{A}$.*

Let $t = (E, \preccurlyeq, \xi)$ be a process and let $t'$ be a prefix of $t$. Since $t$ is meant to describe a concurrent execution of $\mathcal{A}$, $t'$ corresponds to a partial execution and to complete it, the system has to perform $E \setminus E'$. As the minimal events of this remainder are mutually incomparable, the pomset does not restrict the order in which they shall be performed. Therefore, the system should be able to execute them in parallel, which means that the step $\xi(\min(E \setminus E'))$ should be independent after performing $t'$ or any of its linear extensions. This is precisely the requirement in the above definition. Note that the set $\mathfrak{p}(\mathcal{A})$ is in particular closed under prefixes and order extensions. A process is final if all its linear extensions lead from the initial state to some final state.

Equivalently we can characterize the class of final processes as follows: $\mathfrak{p}_f(\mathcal{A})$ collects all pomsets $t \in \mathbb{P}(\Sigma)$ such that $\mathrm{TE}(\mathrm{Pref}(t)) \subseteq T(\mathcal{A})$ and $\mathrm{LE}(t) \subseteq \mathrm{SL}(\mathcal{A})$. We want to show now that $\mathfrak{p}_f(\mathcal{A})$ coincides with $\mathcal{L}(\mathcal{A})$ as stated in Theorem 14. Observe first that $\mathrm{SE}(\mathrm{Pref}(\mathcal{L}(\mathcal{A}))) = \mathrm{Pref}(\mathrm{SE}(\mathcal{L}(\mathcal{A}))) = \mathrm{Pref}(\mathrm{SL}(\mathcal{A}))$.

**Proposition 12.** $\mathrm{Pref}(\mathcal{L}(\mathcal{A}))$ *is step-closed.*

**Proof.** Let $t \in \mathbb{P}(\Sigma)$ be such that $\mathrm{SE}(t) \subseteq \mathrm{SE}(\mathrm{Pref}(\mathcal{L}(\mathcal{A})))$. We have to show that $t \in \mathrm{Pref}(\mathcal{L}(\mathcal{A}))$. Since $\mathrm{SE}(t) \subseteq \mathrm{Pref}(\mathrm{SL}(\mathcal{A}))$, there exists a state $q$ such that for any step sequence $u \in \mathrm{SE}(t)$, we have $\imath \xrightarrow{u} q$ because $\mathcal{A}$ is deterministic. Moreover there is some final state $q_f \in F$ and some $w \in \Sigma^\star$ such that $q \xrightarrow{w} q_f$. Then $\mathrm{SE}(t) \cdot w \subseteq \mathrm{SL}(\mathcal{A})$. Since $\mathrm{SE}(t \cdot w) = \mathrm{SE}(t) \cdot w$, we get $t \cdot w \in \mathcal{L}(\mathcal{A})$ and $t \in \mathrm{Pref}(\mathcal{L}(\mathcal{A}))$. ■

**Proposition 13.** *For all* $t \in \mathbb{P}(\Sigma)$, $t \in \mathrm{Pref}(\mathcal{L}(\mathcal{A}))$ *iff* $\mathrm{TE}(\mathrm{Pref}(t)) \subseteq T(\mathcal{A})$.

**Proof.** Let $K = \mathrm{SL}(\mathcal{A})$. Assume first that $t \in \mathrm{Pref}(\mathcal{L}(\mathcal{A}))$. We have $\mathrm{SE}(t) \subseteq \mathrm{SE}(\mathrm{Pref}(\mathcal{L}(\mathcal{A}))) = \mathrm{Pref}(K)$. Now $\mathrm{TE}(\mathrm{Pref}(t)) \subseteq \mathrm{SE}(\mathrm{Pref}(t)) = \mathrm{Pref}(\mathrm{SE}(t)) \subseteq \mathrm{Pref}(K)$. It follows that $\mathrm{TE}(\mathrm{Pref}(t)) \subseteq \mathrm{Pref}(K) \cap \mathbb{T}(\Sigma) = T(\mathcal{A})$.

Conversely we show that all pomsets $t$ such that $\mathrm{TE}(\mathrm{Pref}(t)) \subseteq T(\mathcal{A})$ belong to $\mathrm{Pref}(\mathcal{L}(\mathcal{A}))$. We proceed by induction over the size of $t$. The claim is trivial if $t$ is empty. Induction step: Let $t$ be such that $\mathrm{TE}(\mathrm{Pref}(t)) \subseteq T(\mathcal{A})$. Recall now that $\mathrm{SE}(\mathrm{Pref}(\mathcal{L}(\mathcal{A}))) = \mathrm{Pref}(K)$. By Proposition 12 above, $\mathrm{Pref}(\mathcal{L}(\mathcal{A}))$ is step-closed. So we just have to show that each step sequence $u \in \mathrm{SE}(t)$ belongs to $\mathrm{Pref}(K)$. Assume $u = u'.m$ with $u' \in \mathbb{S}(\Sigma)$ and $m \in M(\Sigma)$. Let $t'$ be a prefix of $t$ such that $u' \in \mathrm{SE}(t')$. Then $\mathrm{TE}(\mathrm{Pref}(t')) \subseteq \mathrm{TE}(\mathrm{Pref}(t)) \subseteq T(\mathcal{A})$. By induction hypothesis we know that $t' \in \mathrm{Pref}(\mathcal{L}(\mathcal{A}))$. Since $u' \in \mathrm{SE}(t')$ we get $u' \in \mathrm{SE}(\mathrm{Pref}(\mathcal{L}(\mathcal{A})))$, i.e. $u' \in \mathrm{Pref}(K)$. For any $s \in \mathrm{LE}(u')$, $s \cdot m \in \mathrm{TE}(t)$ hence $s \cdot m \in T(\mathcal{A})$. By definition we have $T(\mathcal{A}) \subseteq \mathrm{Pref}(K)$ hence $s \cdot m \in \mathrm{Pref}(K)$. By Lemma 8 we know that $K$ is quasi-consistent. Since $s \in \mathrm{LE}(u')$, this implies that $K \setminus s = K \setminus u'$. Now $s \cdot m \in \mathrm{Pref}(K)$ hence $u' \cdot m \in \mathrm{Pref}(K)$, i.e. $u \in \mathrm{Pref}(K)$. ■

We can now prove that the pomset semantics adopted in Def. 2 coincides with the notion of final processes from Def. 11.

**Theorem 14.** *For any deterministic step transition system* $\mathcal{A}$, $\mathcal{L}(\mathcal{A}) = \mathfrak{p}_f(\mathcal{A})$.

**Proof.** For any $t \in \mathcal{L}(\mathcal{A})$, we have

$$\mathrm{TE}(\mathrm{Pref}(t)) \subseteq \mathrm{SE}(\mathrm{Pref}(t)) \subseteq \mathrm{Pref}(\mathrm{SE}(t)) \subseteq \mathrm{Pref}(\mathrm{SL}(\mathcal{A}))$$

hence $\mathrm{TE}(\mathrm{Pref}(t)) \subseteq \mathrm{Pref}(\mathrm{SL}(\mathcal{A})) \cap \mathbb{T}(\Sigma) = T(\mathcal{A})$. Moreover $\mathrm{SE}(t) \subseteq \mathrm{SL}(\mathcal{A})$ hence $\mathrm{LE}(t) \subseteq \mathrm{SL}(\mathcal{A})$. Thus $t \in \mathfrak{p}_f(\mathcal{A})$.

Conversely assume now that $t \in \mathfrak{p}_f(\mathcal{A})$. Then $\mathrm{TE}(\mathrm{Pref}(t)) \subseteq T(\mathcal{A})$, i.e. $t \in \mathrm{Pref}(\mathcal{L}(\mathcal{A}))$ by Prop. 13 and moreover $\mathrm{LE}(t) \subseteq \mathrm{SL}(\mathcal{A})$. This implies that for all $s \in \mathrm{LE}(t)$ there exists some $q_s \in F$ such that $\imath \xrightarrow{s} q_s$. Consider now some $u \in \mathrm{SE}(t)$. Since $t \in \mathrm{Pref}(\mathcal{L}(\mathcal{A}))$, we have $u \in \mathrm{SE}(\mathrm{Pref}(\mathcal{L}(\mathcal{A}))) = \mathrm{Pref}(\mathrm{SE}(\mathcal{L}(\mathcal{A}))) = \mathrm{Pref}(\mathrm{SL}(\mathcal{A}))$ so there exists some state $q_u$ such that $\imath \xrightarrow{u} q_u$. Let $s \in \mathrm{LE}(u)$. Then on one hand $s \in \mathrm{LE}(t)$ and on the other hand $\imath \xrightarrow{s} q_u$. It follows that $q_u \in F$ and $u \in \mathrm{SL}(\mathcal{A})$. Thus $\mathrm{SE}(t) \subseteq \mathrm{SL}(\mathcal{A})$, i.e. $t \in \mathcal{L}(\mathcal{A})$. ■

However for non-deterministic step transition systems, this identity fails in general.

## 2.4 Expressive Power of Deterministic Step Transition Systems

To conclude this section we present a characterization of the pomset languages of deterministic STSs that is actually the converse property of Cor. 10. The formulation of this result is much simpler than the axiomatization established in [19, Def. 2.10 and Th. 2.16]. This simplification is a key step for the sequel of this paper.

**Theorem 15.** *A pomset language is recognized by some deterministic step transition system if and only if it is step-closed and quasi-consistent.*

In the rest of this section we fix some *deterministic* STS $\mathcal{A}$. Since the pomset language of $\mathcal{A}$ is step-closed (Prop. 5) the proof of Theorem 15 follows from Proposition 18.

We mimic [19] and introduce now the trace relation $\sim$ over $\Sigma^\star$ as the least equivalence relation such that

– $\forall u, u' \in \Sigma^\star, \forall a \in \Sigma$: $u \sim v \Rightarrow u.a \sim v.a$
– $\forall u \cdot m \in T(\mathcal{A}), \forall m' \leqslant m, \forall v, v' \in \mathrm{LE}(m')$: $u.v \sim u.v'$

Since $\mathcal{A}$ is deterministic, it is easy to check that

– For any words $u, v \in \Sigma^\star$, $\imath \xrightarrow{u} q \wedge u \sim v$ implies $\imath \xrightarrow{v} q$.
– For all $u, v \in \Sigma^\star$ and all $m \in M(\Sigma)$: $u \cdot m \in T(\mathcal{A}) \wedge u \sim v$ implies $v \cdot m \in T(\mathcal{A})$

The next two results were established in [19] for deterministic step transition systems *without autoconcurrency*. But their proofs can be easily extended to the context of autoconcurrency. Due to Theorem 14 they can be rephrased as follows.

**Lemma 16.** *[19, Lemma 2.5] Let* $t \in \mathrm{Pref}(\mathcal{L}(\mathcal{A}))$. *For all words* $u, v \in \mathrm{LE}(t)$ *we have* $u \sim v$.

**Lemma 17.** *[19, Lemma 2.6] Let* $t = (E, \preccurlyeq, \xi)$ *be a pomset from* $\mathbb{P}(\Sigma)$. *Then* $t \in \mathrm{Pref}(\mathcal{L}(\mathcal{A}))$ *if and only if for any prefix* $t' = (E', \preccurlyeq', \xi')$ *of* $t$*, there exists at least one linear extension* $u \in \mathrm{LE}(t')$ *such that* $u.\xi(\min_{\preccurlyeq}(E \setminus E')) \in T(\mathcal{A})$.

This lemma will be useful in the next section in order to characterize in a logical way which pomset languages correspond to some deterministic STS. For now we derive from Lemma 16 the announced result which concludes the proof of Theorem 15.

**Proposition 18.** $\mathcal{L}(\mathcal{A})$ *is quasi-consistent.*

**Proof.** Let $t, t'$ be two pomsets from $\mathrm{Pref}(\mathcal{L}(\mathcal{A}))$ such that $\mathrm{LE}(t) \cap \mathrm{LE}(t') \neq \emptyset$. We have to show that $\mathcal{L}(\mathcal{A}) \setminus t = \mathcal{L}(\mathcal{A}) \setminus t'$. Let $r \in \mathcal{L}(\mathcal{A}) \setminus t$. Then $\mathrm{SE}(t) \cdot \mathrm{SE}(r) \subseteq \mathrm{SE}(\mathcal{L}(\mathcal{A}))$. By Lemma 16, all linear extensions of $t$ (resp. $t'$) are $\sim$-equivalent. Since $\mathrm{LE}(t) \cap \mathrm{LE}(t') \neq \emptyset$, we get that all linear extensions of $t$ and $t'$ are $\sim$-equivalent. Since $\mathcal{A}$ is deterministic, there is a state $q$ such that for any word $s \in \mathrm{LE}(t) \cup \mathrm{LE}(t')$, we have $\imath \xrightarrow{s} q$. Then $\imath \xrightarrow{u} q$ for any step sequence $u \in \mathrm{SE}(t) \cup \mathrm{SE}(t')$. Since $\mathrm{SE}(t) \cdot \mathrm{SE}(r) \subseteq \mathrm{SE}(\mathcal{L}(\mathcal{A}))$ and $\mathrm{SE}(\mathcal{L}(\mathcal{A})) = \mathrm{SL}(\mathcal{A})$ we get $\mathrm{SE}(t') \cdot \mathrm{SE}(r) \subseteq \mathrm{SE}(\mathcal{L}(\mathcal{A}))$, i.e. $\mathrm{SE}(t' \cdot r) \subseteq \mathrm{SE}(\mathcal{L}(\mathcal{A}))$. Since $\mathcal{L}(\mathcal{A})$ is step-closed, this implies that $t' \cdot r \in \mathcal{L}(\mathcal{A})$ hence $r \in \mathcal{L}(\mathcal{A}) \setminus t'$. Thus we have $\mathcal{L}(\mathcal{A}) \setminus t \subseteq \mathcal{L}(\mathcal{A}) \setminus t'$. The result follows by symmetry. ■

## 3 Regular Pomset Languages

In language theory, a set of words $L \subseteq \Sigma^\star$ is called regular if it has finitely many residuals. For a given word $u$, the residual at $u$ consists of all words $v$ such that $u.v \in L$. In particular, if $u$ is not a prefix of some word from $L$ then the residual at $u$ is empty. By analogy with these classical definitions, we introduced in [11] the notion of a regular pomset language which extends the classical notion of regularity for word languages.

**Definition 19.** *Let* $\mathcal{L}$ *be a set of pomsets. Given two pomsets* $t$ *and* $t'$*, we put* $t \equiv^r t'$ *if* $\mathcal{L} \setminus t = \mathcal{L} \setminus t'$. *Then* $\mathcal{L}$ *is* regular *if the equivalence relation* $\equiv^r$ *is of finite index.*

We observed in [11] that this notion of regularity coincides with the usual notion of regularity for Mazurkiewicz traces [5], message sequence charts [14], consistent sets of pomsets [1], and more generally to local trace languages [19].

In this section we relate first this notion of regularity with the class of finite step transition systems (Theorems 21 and 22). Next we extend to the setting of autoconcurrency the main technical result from [19]: *The basis of the pomset language of a finite deterministic step transition system is prime-bounded* (Theorem 36).

## 3.1 Pomset Languages of Finite Step Transition Systems

Recall that a finite STS is *width-bounded*, i.e. it has bounded steps: There is some natural number $k \in \mathbb{N}$ such that the size of any multiset $m$ is at most $k$ if $m$ occurs in a transition from $\mathcal{A}$. We denote by $\mathbb{P}_{w<k}(\Sigma)$ the class of pomsets whose width is bounded by $k$, that is, whose steps are bounded by $k$. We start by the useful next result.

**Proposition 20.** *[11, Th. 3.5] Let $\mathcal{L}$ be a width-bounded and step-closed pomset language. Then $\mathcal{L}$ is regular if and only if* $\mathrm{SE}(\mathcal{L})$ *is regular.*

As a consequence the pomset language of any finite step transition system $\mathcal{A}$ is regular because $\mathrm{SE}(\mathcal{L}(\mathcal{A})) = \mathrm{SL}(\mathcal{A})$ and moreover $\mathrm{SL}(\mathcal{A})$ is obviously regular. It is also width-bounded. Actually regularity coincides with finite step transition systems as shown by the next result.

**Theorem 21.** *Let $\mathcal{L}$ be a step-closed pomset language. Then $\mathcal{L}$ is the language of some* finite *step transition system if and only if $\mathcal{L}$ is regular and width-bounded.*

**Proof.** If $\mathcal{L}$ is regular then $\mathrm{SE}(\mathcal{L})$ is regular (Prop. 20). Similarly to Prop 5, we can build a finite STS $\mathcal{A}$ such that $\mathrm{SL}(\mathcal{A}) = \mathrm{SE}(\mathcal{L})$. Then $\mathcal{L}(\mathcal{A}) = \langle \mathrm{SE}(\mathcal{L}) \rangle = \mathcal{L}$. ■

We focus now on pomset languages of *deterministic* step transition systems. By Theorem 15 these languages are step-closed and quasi-consistent. It is clear also that they are width-bounded because the step language of $\mathcal{A}$ is width-bounded. Moreover Theorem 21 asserts that they are regular. The next result characterizes the languages of deterministic finite step transition systems by establishing the converse property.

**Theorem 22.** *Let $\mathcal{L}$ be a step-closed pomset language. Then $\mathcal{L}$ is the language of some finite* deterministic *step transition system if and only if $\mathcal{L}$ is regular, width-bounded and quasi-consistent.*

**Proof.** Assume that $\mathcal{L}$ is step-closed, quasi-consistent, width-bounded and regular. Since $\mathcal{L}$ is step-closed and width-bounded, the step language $\mathrm{SE}(\mathcal{L})$ is regular (Prop. 20). Moreover $\mathrm{SE}(\mathcal{L})$ is quasi-consistent because $\mathcal{L}$ is quasi-consistent. Furthermore we know that $\mathrm{SE}(\mathrm{SE}(\mathcal{L})) = \mathrm{SE}(\mathcal{L})$. Therefore we can apply the construction of Lemma 8 and get a deterministic STS whose step language is $\mathrm{SE}(\mathcal{L})$. Moreover the set of states is the quotient $\mathrm{SE}(\mathcal{L})/{\simeq}$ which is finite because $\mathrm{SE}(\mathcal{L})$ is regular. ■

## 3.2 Basis of a Pomset Language

As far as the description of concurrency is concerned, some redundancy of information may appear in weak languages. In order to focus on restricted but representative parts of weak languages, we look at *basic* sets of pomsets.

**Definition 23.** *A language of pomsets $\mathcal{L}$ is* basic *if $t_1 \in \mathrm{OE}(t_2)$ implies $t_1 = t_2$, for all $t_1, t_2 \in \mathcal{L}$.*

We can check easily that two basic sets that have the same order extensions are equal: $\mathrm{OE}(\mathcal{L}) = \mathrm{OE}(\mathcal{L}')$ implies $\mathcal{L} = \mathcal{L}'$ for all basic sets of pomsets $\mathcal{L}$ and $\mathcal{L}'$. Therefore the map OE from basic sets of pomsets to weak sets of pomsets is one-to-one. Actually, we shall see that this map is also onto.

**Definition 24.** *Let $\mathcal{L}$ be a language of pomsets. The basis of $\mathcal{L}$ consists of all pomsets $t \in \mathcal{L}$ which are no order extension of some other pomsets of $\mathcal{L}$:*

$$\mathrm{B}(\mathcal{L}) = \{t \in \mathcal{L} \mid \forall t' \in \mathcal{L} : t \in \mathrm{OE}(t') \Rightarrow t = t'\}.$$

For any weak language $\mathcal{L}$, $\mathrm{B}(\mathcal{L})$ is a basic set of pomsets and $\mathrm{OE}(\mathrm{B}(\mathcal{L})) = \mathcal{L}$. Thus, we obtain *a one-to-one correspondence between weak languages and basic languages.* However, as already observed in [19], this duality does not preserve regularity in general. Still we have the next fact:

**Proposition 25.** *[11, Cor. 2.7] Let $\mathcal{L}$ be a weak pomset language. If $\mathrm{B}(\mathcal{L})$ is regular then $\mathcal{L}$ is regular.*

The next example shows that the converse property fails: It exhibits a weak language that is regular whereas its basis is not regular.

**Example 26.** We consider the subset $\mathcal{L}_0$ of all pomsets $t$ that consist of two rows of $a$-events and an additional $b$-event. The language $\mathcal{L}_1 \subset \mathcal{L}_0$ restricts to the pomsets such that either the $b$-event covers the first $a$ of each row, or covers the $k$-th $a$-event of one row and is covered by the $2k$-th $a$-event of the other row. Examples of pomsets from $\mathcal{L}_1$ are depicted in Fig. 4. We claim that $\mathcal{L}_1$ is basic and not regular. Let $\mathcal{L} = \mathrm{OE}(\mathcal{L}_1)$. Then $\mathcal{L}$ is weak and $\mathcal{L}_1$ is the basis of $\mathcal{L}$. However, we observe that $\equiv^r_{\mathcal{L}}$ has index 5: If $t$ is not a prefix of some $t' \in \mathcal{L}$, then $\mathcal{L} \setminus t = \emptyset$. Now $\mathcal{L} \setminus \varepsilon = \mathcal{L}$, $\mathcal{L} \setminus a = \mathrm{OE}(\mathcal{L} \setminus a)$, and moreover for all pomsets $t$ that are prefixes of $\mathcal{L}$ different from the empty pomset $\varepsilon$ and the singleton pomset $a$, the residual $\mathrm{OE}(\mathcal{L}) \setminus t$ depends only on whether $b$ occurs in $t$. To be more precise, let $\mathcal{L}_2$ be the language of all pomsets consisting of two rows of $a$. Then $\mathrm{OE}(\mathcal{L}) \setminus t$ equals $\mathrm{OE}(\mathcal{L}_2)$ if $b$ occurs in $t$ and $\mathrm{OE}(\mathcal{L}_0)$ otherwise.

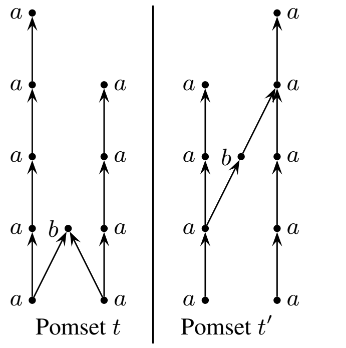
## 3.3 Prime-Bounded Pomset Languages

We need to recall now the notion of a prime-bounded pomset language.

**Definition 27.** *[17] Let $t = (E, \preccurlyeq, \xi)$ be a pomset and $k$ be a positive integer. A $k$-*chain covering *of $t$ is a family $(C_i)_{i \in [1,k]}$ of subsets of $E$ such that*

1. *each $C_i$ is a chain in $(E, \preccurlyeq)$, i.e. $(C_i, \preccurlyeq \cap (C_i \times C_i))$ is a linear order;*
2. *$E = \bigcup_{i \in [1,k]} C_i$;*
3. *$\forall x, y \in E$: $(x \prec\!\!\cdot\, y \Rightarrow \exists i \in [1,k] : \{x, y\} \subseteq C_i)$.*

For any $k \in \mathbb{N}$, $\mathbb{P}_k(\Sigma)$ denotes the class of pomsets over $\Sigma$ which admit a $k$-chain covering. A class of pomsets over $\Sigma$ is *prime-bounded* if it is included in some $\mathbb{P}_k(\Sigma)$.



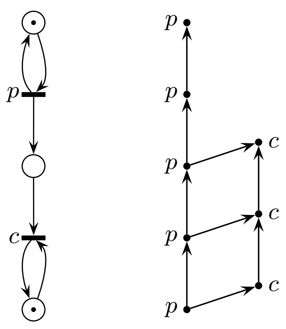


**Fig. 4.** Two pomsets from $\mathcal{L}_1$ in Ex. 26 **Fig. 5.** The Producer-Consumer and its basis

By Dilworth's Theorem [6], the first two requirements are equivalent to saying that $(E, \preccurlyeq)$ has width at most $k$. So the crucial part is the third requirement. It enforces that not only the partial order $(E, \preccurlyeq)$ is covered by the chains, but that in addition the end points of any prime interval belong to some common chain.

**Example 28.** We consider Producer-Consumer Petri net from Figure 5. A *broken ladder* is a pomset $t$ over $\Sigma = \{p, c\}$ that consists of a chain of $n$ production events (labeled by $p$) and a chain of $m$ consumption events (labeled by $c$) with $m \leqslant n$ and such that

– the $k^{th}$ consumption covers the $k^{th}$ production;
– no consumption is below any production.

An example of a broken ladder is described on Figure 5. The set $\mathcal{L}$ of all broken ladders is the basis of the pomset language corresponding to $\mathcal{N}$. Note that the set of broken ladders has width 2, but is not prime-bounded: The $k^{\text{th}}$ production and consumption have to belong to some chain that cannot contain the $l^{\text{th}}$ production for $l > k$.

Theorem 14 asserts that the pomset language considered in Def. 2 corresponds to the process semantics from Def. 11. Therefore we can rephrase the main technical result from [19] as follows.

**Lemma 29.** *[19, Lemma 3.15] Let $\mathcal{A}$ be a finite deterministic step transition system* without autoconcurrency. *Then the basis of $\mathcal{L}(\mathcal{A})$ is prime-bounded.*

In the rest of this section we extend this result to the general case of step transition systems with autoconcurrency.

## 3.4 Generalization to Step Transition Systems with Autoconcurrency

In the rest of this section we fix some $k \in \mathbb{N}$ and some pomset language $\mathcal{L} \subseteq \mathbb{P}(\Sigma)$. We assume that $\mathcal{L}$ is weak and $k$-width-bounded: $\mathcal{L} = \text{OE}(\mathcal{L})$ and $\mathcal{L} \subseteq \mathbb{P}_{w<k}(\Sigma)$.

Let $\Omega = \Sigma \times [1, k]$ be the set of pairs $(a, i)$ such that $a \in \Sigma$ and $1 \leqslant i \leqslant k$. We let $\pi : \Sigma \times [1, k] \to \Sigma$ denote the first projection. This projection extends to a mapping on pomsets $\pi : \mathbb{P}(\Sigma \times [1,k]) \to \mathbb{P}(\Sigma)$ such that for any pomset $t = (E, \preccurlyeq, \xi) \in \mathbb{P}(\Omega)$, we have $\pi(t) = (E, \preccurlyeq, \pi \circ \xi)$. We denote by $\mathbb{P}^{nac}(\Omega)$ the set of pomsets without autoconcurrency from $\mathbb{P}(\Omega)$ and consider the language $\mathcal{L}' = \pi^{-1}(\mathcal{L}) \cap \mathbb{P}^{nac}(\Omega)$. Thus $\mathcal{L}'$ collects the pre-images without autoconcurrency of all pomsets from $\mathcal{L}$.

**Lemma 30.** *We have* $\pi(\mathcal{L}') = \mathcal{L}$.

**Proof.** Let $t = (E, \preccurlyeq, \xi)$ be a pomset from $\mathcal{L}$. By Dilworth theorem [6], there exist $k$ paths in $t$ which together contain all events of $t$. From any such set of paths we can derive a chain partition of $E$, i.e. a set of $k$ mutually disjoint subsets which are totally ordered by $\preccurlyeq$ and which together contain all the events of $t$. Let $\{E_1, ..., E_k\}$ denote this partition of $E$. Then the pomset $t' = (E, \preccurlyeq, \xi')$ where for all $e \in E$, $\xi'(e) = (\xi(e), i)$ iff $e \in E_i$ is a pomset without autoconcurrency. Moreover it is clear that $\pi(t') = t$. It follows that $t' \in \mathcal{L}'$. ■

**Lemma 31.** *If $\mathcal{L}$ is step-closed then $\mathcal{L}'$ is step-closed.*

**Proof.** Let $t \in \mathbb{P}(\Omega)$ be such that $\mathrm{SE}(t) \subseteq \mathrm{SE}(\mathcal{L}')$. Since $\mathcal{L}' \subseteq \mathbb{P}^{nac}(\Omega)$ we have $t \in \mathbb{P}^{nac}(\Omega)$. On the other hand, we have $\pi(\mathrm{SE}(t)) = \mathrm{SE}(\pi(t))$ and $\pi(\mathrm{SE}(\mathcal{L}')) = \mathrm{SE}(\pi(\mathcal{L}')) = \mathrm{SE}(\mathcal{L})$ by Lemma 30. It follows that $\mathrm{SE}(\pi(t)) \subseteq \mathrm{SE}(\pi(\mathcal{L}')) = \mathrm{SE}(\mathcal{L})$. Since $\mathcal{L}$ is step-closed, we get $\pi(t) \in \mathcal{L}$. Hence $t \in \pi^{-1}(\mathcal{L}) \cap \mathbb{P}^{nac}(\Omega)$, i.e. $t \in \mathcal{L}'$. This shows that $\mathcal{L}'$ is step-closed. ■

**Lemma 32.** *If* $\mathrm{SE}(\mathcal{L})$ *is quasi-consistent then* $\mathrm{SE}(\mathcal{L}')$ *is quasi-consistent.*

**Proof.** Let $K = \mathrm{SE}(\mathcal{L})$ and $K' = \mathrm{SE}(\mathcal{L}')$. Since $\pi(\mathcal{L}') = \mathcal{L}$ we have $\pi(\mathrm{SE}(\mathcal{L}')) = \mathrm{SE}(\mathcal{L})$, i.e. $\pi(K') = K$. It follows that $\pi(\mathrm{Pref}(K')) = \mathrm{Pref}(K)$. Moreover $K \subseteq \mathcal{L}$ because $\mathcal{L}$ is weak.

Let $u, v \in \mathrm{Pref}(K')$ be such that $\mathrm{LE}(u) \cap \mathrm{LE}(v) \neq \emptyset$. We have to prove that $K' \backslash u = K' \backslash v$. We have $\pi(u) \in \mathrm{Pref}(K)$ and $\pi(v) \in \mathrm{Pref}(K)$. Moreover $\pi(\mathrm{LE}(u) \cap \mathrm{LE}(v)) \subseteq \mathrm{LE}(\pi(u)) \cap \mathrm{LE}(\pi(v)) \neq \emptyset$. Since $K$ is quasi-consistent, we get $K \setminus \pi(u) = K \setminus \pi(v)$.

Let $w \in K' \setminus u$. Then $w \in \mathbb{S}(\Omega)$ because $K' \subseteq \mathbb{S}(\Omega)$. Moreover $u \cdot w \in K'$ because $K' = \mathrm{SE}(\mathcal{L}')$. Therefore we have $\pi(u \cdot w) = \pi(u) \cdot \pi(w) \in K$. Since $K \setminus \pi(u) = K \setminus \pi(v)$, we get $\pi(v) \cdot \pi(w) = \pi(v \cdot w) \in K$. Now $v \cdot w \in \mathbb{S}(\Omega)$ and $\pi(v \cdot w) \in \mathcal{L}$ because $K \subseteq \mathcal{L}$. Moreover neither $v$ or $w$ shows some autoconcurrency. Hence $v \cdot w \in \mathcal{L}'$. It follows that $v \cdot w \in K'$. Thus $K' \setminus u \subseteq K' \setminus v$. By symmetry we get $K' \setminus u = K' \setminus v$. ■

The mapping $\pi$ maps step sequences from $M(\Omega)^\star$ onto step sequences from $M(\Sigma)^\star$. Moreover for any $t \in \mathbb{P}(\Omega)$, $t$ is a step sequence from $M(\Omega)^\star$ if and only if $\pi(t)$ is a step sequence from $M(\Sigma)^\star$. The induced mapping $\pi : M(\Omega)^\star \to M(\Sigma)^\star$ is a surjective monoid morphism for the strong concatenation: For any pair $u, v \in M(\Omega)^\star$ we have $\pi(u \cdot v) = \pi(u) \cdot \pi(v)$.

**Remark 33.** Let $\mathcal{L} \subseteq \mathbb{S}(\Omega)$ be a step language such that $\mathrm{SE}(\mathcal{L}) \subseteq \mathcal{L}$. Then $\mathcal{L}$ is regular if and only if $\mathcal{L}$ is recognizable in the monoid $M(\Omega)^\star$ —because $v \in \mathcal{L} \setminus u$ iff $u \cdot v \in \mathcal{L}$.

**Lemma 34.** *If* $\mathrm{SE}(\mathcal{L})$ *is regular then* $\mathrm{SE}(\mathcal{L}')$ *is regular.*

**Proof.** Let $\wp(\Omega)$ collect all subsets of $\Omega$. We check first that $\mathrm{SE}(\mathcal{L}') = \pi^{-1}(\mathrm{SE}(\mathcal{L})) \cap \wp(\Omega)^\star$. Consider first some $t \in \mathcal{L}'$ and some $u \in \mathrm{SE}(t)$. Then $\pi(u) \in \mathrm{SE}(\pi(t))$ and $\pi(t) \in \pi(\mathcal{L}') = \mathcal{L}$ by Lemma 30. Hence $u \in \pi^{-1}(\mathrm{SE}(\mathcal{L}))$. Now $u$ is a step sequence without autoconcurrency because $t \in \mathcal{L}'$, thus $u \in \pi^{-1}(\mathrm{SE}(\mathcal{L})) \cap \wp(\Omega)^\star$. Conversely, consider now some step sequence $u \in \wp(\Omega)^\star$ such that $\pi(u) \in \mathrm{SE}(\mathcal{L})$. Since $\mathcal{L}$ is weak, we have $\pi(u) \in \mathcal{L}$. Since $u$ shows no autoconcurrency, we get $u \in \pi^{-1}(\mathcal{L}) \cap \mathbb{P}^{nac}(\Omega)$, i.e. $u \in \mathcal{L}'$. Now $u$ is a step sequence, hence $u \in \mathrm{SE}(u)$: Thus $u \in \mathrm{SE}(\mathcal{L}')$.

The mapping $\pi : M(\Omega)^\star \to M(\Sigma)^\star$ is a monoid morphism. Therefore if $\mathrm{SE}(\mathcal{L})$ is recognizable in $M(\Sigma)^\star$ then its pre-image $\pi^{-1}(\mathrm{SE}(\mathcal{L}))$ is recognizable in $M(\Omega)^\star$. Furthermore $\wp(\Omega)^\star$ is a recognizable submonoid of $M(\Omega)^\star$ because for any $u \in M(\Omega)^\star$, $\wp(\Omega)^\star \setminus u$ is either empty or equal to $\wp(\Omega)^\star$. Since $\mathrm{SE}(\mathcal{L}') = \pi^{-1}(\mathrm{SE}(\mathcal{L})) \cap \wp(\Omega)^\star$, $\mathrm{SE}(\mathcal{L}')$ is the intersection of two recognizable languages, so it is recognizable, too. It follows now from Remark 33 that $\mathrm{SE}(\mathcal{L}')$ is regular. ∎

**Lemma 35.** *We have* $\mathrm{B}(\mathcal{L}) \subseteq \pi(\mathrm{B}(\mathcal{L}'))$.

**Proof.** Let $\mathcal{L}^\circ = \pi^{-1}(\mathrm{B}(\mathcal{L})) \cap \mathbb{P}^{nac}(\Omega)$. Clearly $\mathcal{L}^\circ \subseteq \mathcal{L}'$. By Lemma 30, we have $\mathrm{B}(\mathcal{L}) = \pi(\mathcal{L}^\circ)$. Furthermore we can check that $\mathcal{L}^\circ$ is included in the basis $\mathrm{B}(\mathcal{L}')$: For any pomset $t \in \mathcal{L}^\circ$, if $t \in \mathrm{OE}(t')$ with $t \neq t'$ for some $t' \in \mathcal{L}'$, then $\pi(t) \in \mathrm{OE}(\pi(t'))$ and $\pi(t) \neq \pi(t')$, hence $\pi(t) \notin \mathrm{B}(\mathcal{L})$: Contradiction. Thus $\mathcal{L}^\circ \subseteq \mathrm{B}(\mathcal{L}')$ and $\mathrm{B}(\mathcal{L}) = \pi(\mathcal{L}^\circ) \subseteq \pi(\mathrm{B}(\mathcal{L}'))$. ∎

We can now extend Lemma 29 to the setting of autoconcurrency.

**Theorem 36.** *Let $\mathcal{L}$ be a step-closed, quasi-consistent and width-bounded pomset language. If $\mathcal{L}$ is regular then its basis $\mathrm{B}(\mathcal{L})$ is prime-bounded.*

**Proof.** Since $\mathcal{L}$ is weak and width-bounded, we can apply the above construction of $\mathcal{L}'$ from $\mathcal{L}$. It is clear that $\mathcal{L}'$ is width-bounded. Since $\mathcal{L}$ is regular we know that $\mathrm{SE}(\mathcal{L})$ is regular by Prop. 20. Now Lemma 31 proves that $\mathcal{L}'$ is step-closed. Furthermore Lemma 34 asserts that $\mathrm{SE}(\mathcal{L}')$ is regular hence $\mathcal{L}'$ is regular (Prop. 20). On the other hand Lemma 32 shows that $\mathrm{SE}(\mathcal{L}')$ quasi-consistent. Since $\mathcal{L}'$ is step-closed this implies that $\mathcal{L}'$ is quasi-consistent. Theorem 22 ensures that $\mathcal{L}'$ is the language of a deterministic finite step transition system $\mathcal{A}$. We may assume that $\mathcal{A}$ shows no autoconcurrency at all because $\mathcal{L}(\mathcal{A}) = \mathcal{L}' \subseteq \mathbb{P}^{nac}(\Omega)$. By Lemma 29 the basis of $\mathcal{L}'$ is prime-bounded. The result follows then from Lemma 35. ∎

## 4 MSO-Definable Pomset Languages

In this section we want to characterize the pomset languages that are recognized by some finite deterministic step transition system in a logical way. Among other characterizations we show in Theorem 48 that these languages are exactly the pomset languages whose basis is prime-bounded and definable in Monadic Second-Order logic.

Admittedly this equivalence resembles the main result from [19]. However again we consider here step transition systems and pomsets with autoconcurrency. That is why we need to develop a new technique in order to handle concurrent events that carry the same action. In particular, we will explain why it is not possible to rely on the lexicographically least linear extension similarly to [5,19]. For that reason we make use of the notion of chain partition and related results borrowed from [4].

### 4.1 Monadic Second-Order Logic on Pomsets

Formulae of the MSO logic that we consider involve first-order variables $x, y, z...$ for events and second-order variables $X, Y, Z...$ for sets of events. They are built up from the atomic formulae $P_a(x)$ for $a \in \Sigma$ (which stands for "the event $x$ is labeled by the action $a$"), $x \preccurlyeq y$, and $x \in X$ by means of the boolean connectives $\neg, \vee, \wedge, \rightarrow, \leftrightarrow$ and quantifiers $\exists, \forall$ (both for first order and for set variables). We denote by $\mathrm{MSO}(\Sigma)$ the set of all formulae of MSO. Formulae without free variables are called sentences.

The satisfaction relation $\models$ between pomsets and sentences is defined canonically with the understanding that first order variables range over events of $E$ and second order variables over subsets of $E$. The set of pomsets which satisfy a sentence $\varphi$ is denoted by $\mathrm{Mod}(\varphi)$. We say that a set of pomsets $\mathcal{L}$ is MSO-*definable* if there exists a sentence $\varphi$ such that $\mathcal{L} = \mathrm{Mod}(\varphi)$.

**Example 37.** We continue Example 28. The language $\mathcal{L}$ of all broken ladders is MSO-definable by

$$\begin{aligned}
&\forall x, y : (P_p(x) \wedge P_p(y)) \vee (P_c(x) \wedge P_c(y)) \rightarrow (x \preccurlyeq y \vee y \preccurlyeq x)\\
&\forall x, y : P_p(y) \wedge x \preccurlyeq y \rightarrow P_p(x)\\
&\forall y : (P_c(y) \rightarrow \exists x(P_p(x) \wedge x \prec\!\cdot\, y))\\
&\forall x, z : ((P_p(x) \wedge P_c(z) \wedge x \preccurlyeq z) \rightarrow \exists y(P_c(y) \wedge x \prec\!\cdot\, y))
\end{aligned}$$

However $\mathrm{OE}(\mathcal{L})$ is not MSO-definable since $\mathrm{LE}(\mathcal{L})$ is not regular.

### 4.2 Monadic Second-Order Logic on Mazurkiewicz Traces

Let us recall some basic notion from Mazurkiewcz trace theory [5]. The concurrency of a distributed system is often represented by an *independence relation* over the alphabet of actions $\Sigma$, that is a binary, symmetric, and irreflexive relation $\| \subseteq \Sigma \times \Sigma$. The associated *trace equivalence* is the least congruence $\sim$ over $\Sigma^\star$ such that $\forall a, b \in \Sigma, a \| b \Rightarrow ab \sim ba$. A *trace* $[u]$ is the equivalence class of a word $u \in \Sigma^\star$. We denote by $\mathbb{M}(\Sigma, \|)$ the set of all traces over the independence alphabet $(\Sigma, \|)$. A *trace language* is a subset $\mathcal{L} \subseteq \mathbb{M}(\Sigma, \|)$. Let $u \in \Sigma^\star$; then the trace $[u]$ is precisely the set of linear extensions $\mathrm{LE}(t)$ of a unique pomset $t = (E, \preccurlyeq, \xi)$, that is, $[u] = \mathrm{LE}(t)$. Moreover $t$ satisfies the following properties:

$\mathsf{MP}_1$: For all events $e_1, e_2 \in E$ with $\xi(e_1) \not\| \xi(e_2)$, we have $e_1 \preccurlyeq e_2$ or $e_2 \preccurlyeq e_1$;
$\mathsf{MP}_2$: For all events $e_1, e_2 \in E$ with $e_1 \prec\!\cdot\, e_2$, we have $\xi(e_1) \not\| \xi(e_2)$.

Conversely the linear extensions of a pomset satisfying these two axioms form a trace of $\mathbb{M}(\Sigma, \|)$. Thus one usually identifies $\mathbb{M}(\Sigma, \|)$ with the class of pomsets satisfying $\mathsf{MP}_1$ and $\mathsf{MP}_2$. We have already mentioned that the notion of regularity adopted in this paper coincides with the usual definition for Mazurkiewicz trace languages, that is: A subset $\mathcal{L} \subseteq \mathbb{M}(\Sigma, \|)$ is regular if and only if $\mathrm{LE}(\mathcal{L})$ is a regular word language. An interesting property of Mazurkiewicz trace languages is the following.

**Theorem 38.** *[25] A set of Mazurkiewicz traces $\mathcal{L} \subseteq \mathbb{M}(\Sigma, \|)$ is regular if and only if it is MSO-definable.*

We recall now why prime-bounded languages are related to Mazurkiewicz trace languages. The class of pomsets $\mathbb{M}(\Sigma, \|)$ is prime-bounded by $\mathrm{Card}(\Sigma)^2$ whenever $\Sigma$ is finite. To see this, we simply consider for each pair of dependent actions $a \not\| b$ the set of events labeled by $a$ or $b$. Then, by Axiom $\mathsf{MP}_1$, this set forms a chain and, by Axiom $\mathsf{MP}_2$, the family of these chains forms a chain covering. Conversely, we shall explain below that any prime-bounded language $\mathcal{L} \subseteq \mathbb{P}(\Sigma)$ can be represented — up to a relabeling function — by a trace language over an appropriate finite independence alphabet by means of a construction borrowed from [17].

For each $k \in \mathbb{N}$ we define the independence alphabet $(\Gamma_k, \|)$ by $\Gamma_k = \Sigma \times (\wp([1,k]) \setminus \{\emptyset\})$ and $(a, M) \| (b, N)$ iff $M \cap N = \emptyset$ and $a \neq b$. Let $\pi_1 : \Gamma_k \to \Sigma$ be the projection to the first component. Then there is an obvious extension of $\pi_1$ to pomsets over $\Gamma_k$ defined by $\pi_1(E, \preccurlyeq, \xi) = (E, \preccurlyeq, \pi_1 \circ \xi)$. Now let $t = (E, \preccurlyeq, \xi)$ be a pomset over $\Sigma$ which admits $(C_i)_{1 \leqslant i \leqslant k}$ as $k$-chain covering. We define a new labeling function $\xi' : E \to \Gamma_k$ by $\xi'(e) = (\xi(e), \{i \in [1,k] \mid e \in C_i\})$. Since for any $e, f \in E$ with $e \mathrel{-\!\!\!<\!\!\cdot} f$, there exists $1 \leqslant i \leqslant k$ with $e, f \in C_i$, the new pomset $t' = (E, \preccurlyeq, \xi')$ belongs to $\mathbb{M}(\Gamma_k, \|)$ — that is, it satisfies Axioms $\mathsf{MP}_1$ and $\mathsf{MP}_2$.

**Corollary 39.** *Let $\mathcal{L} \subseteq \mathbb{P}(\Sigma)$ be a pomset language. If $\mathcal{L}$ is MSO-definable and prime-bounded then $\mathcal{L}$ is regular.*

**Proof.** Since $\mathcal{L}$ is prime-bounded we can build a finite independence alphabet $(\Gamma, \|)$ and a mapping $\pi : \Gamma \to \Sigma$ such that $\mathcal{L} \subseteq \pi(\mathbb{M}(\Gamma, \|))$. We put $\mathcal{L}' = \pi^{-1}(\mathcal{L}) \cap \mathbb{M}(\Gamma, \|)$. Then $\mathcal{L}'$ is MSO-definable because $\mathcal{L}$ is MSO-definable. It follows from Theorem 38 that $\mathcal{L}'$ is regular. Since $\mathcal{L} = \pi(\mathcal{L}')$ we get that $\mathcal{L}$ is regular. ■

## 4.3 Two Other Preliminary Observations

The first basic observation asserts that the basis of an MSO-definable pomset language is MSO-definable, too.

**Lemma 40.** *Let $\mathcal{L} \subseteq \mathbb{P}(\Sigma)$ be a pomset language. If $\mathcal{L}$ is MSO-definable then $\mathrm{B}(\mathcal{L})$ is MSO-definable, too.*

**Proof.** Let $\kappa$ be a sentence for $\mathcal{L}$. We define by induction the transformation $\alpha \mapsto \alpha^{e,f}$ of MSO-formulae (where $e$ and $f$ are two new free variables) by replacing $x \preccurlyeq y$ in $\alpha$ by $x \preccurlyeq y \wedge (x \neq e \vee y \neq f)$. Then we claim that the basis of $\mathcal{L}$ is defined by

$$\kappa' := \kappa \wedge \forall e, f : e \mathrel{-\!\!\!<\!\!\cdot} f \to \neg \kappa^{e,f}$$

To see that $\kappa'$ defines $\mathrm{B}(\mathcal{L})$, consider a pomset $t = (E, \preccurlyeq, \xi)$ from $\mathcal{L}$ and a prime interval $e \mathrel{-\!\!\!<\!\!\cdot} f$ in $t$. Then $t^{e,f} = (E, \preccurlyeq \setminus \{(e,f)\}, \xi)$ is still a pomset over $\Sigma$. Moreover, $t$ is in the basis of $\mathcal{L}$ if and only if for any $e \mathrel{-\!\!\!<\!\!\cdot} f$, $t^{e,f}$ is not in $\mathcal{L}$. ■

Note that the converse property may fail in general: Example 37 exhibits a weak language $\mathrm{OE}(\mathcal{L})$ that is not MSO-definable although its basis $\mathcal{L}$ is MSO-definable.

Step languages can be regarded as words over the alphabet of all multisets. By Buchi's theorem, any MSO-definable word language is regular. With no surprise we can easily establish a similar result for step languages.

**Corollary 41.** *Let $\mathcal{L} \subseteq \mathbb{S}(\Sigma)$ be a width-bounded step language. If $\mathcal{L}$ is MSO-definable then $\mathcal{L}$ is regular.*

**Proof.** It is clear that $\mathcal{L}$ is prime-bounded because it consists of sequences of bounded steps. The result follows then directly from Cor. 39. ■

The next example proves that this property may fail for general pomset languages: There are regular pomset languages that are not MSO-definable.

**Example 42.** Let us consider again the pomset language $\mathcal{L}$ from Example 26. Then $\mathcal{L}$ is regular and weak whereas its basis $\mathrm{B}(\mathcal{L})$ is prime-bounded but not regular. By Cor. 39, $\mathrm{B}(\mathcal{L})$ is not MSO-definable. It follows from Lemma 40 that $\mathcal{L}$ is not MSO-definable.

This example shows that regular pomset languages are not MSO-definable in general. However our goal now is to establish this property for the restricted class of pomset languages generated by finite deterministic step transition systems, that is, for regular, width-bounded, step-closed and quasi-consistent pomset languages, as stated in Prop. 47.

### 4.4 Linear Extensions and Chain Partitions

Let $k$ be an integer and $\mathbb{P}_{w<k}(\Sigma)$ be the set of pomsets width-bounded by $k$. By Dilworth theorem [6], a pomset $t = (E, \preccurlyeq, \xi)$ belongs to $\mathbb{P}_{w<k}(\Sigma)$ iff there exists a set of $k$ paths in $t$ which together contain all events of $t$. From any such set we can derive a chain partition of $E$, i.e. a set of mutually disjoint subsets each being totally ordered by $\preccurlyeq$ and which together contain all events of $t$.

In the sequel we shall make use of the following open formula:

$$ChainPart(X_1...X_k) := \forall x, y. \begin{bmatrix} \bigvee_{i\in[1,k]}(x \in X_i) \wedge \bigwedge_{i\neq j}(x \in X_i \rightarrow x \notin X_j)\wedge \\ \bigwedge_{i\in[1,k]}(x \in X_i \wedge y \in X_i) \rightarrow (x \leqslant y \vee y \leqslant x) \end{bmatrix}$$

**Proposition 43.** *A pomset $t \in \mathbb{P}(\Sigma)$ is $k$-width-bounded if and only if*

$$t \models \exists X_1...X_k.ChainPart(X_1...X_k).$$

Given a chain partition $D_1, ..., D_k$ of a pomset $t = (E, \preccurlyeq, \xi)$, two concurrent events of $E$ belong to different chains. Using the total ordering on the set $D_1, ..., D_k$, we can define a minimal topological sorting of $(E, \preccurlyeq)$ w.r.t. this ordered chain partition. Interestingly the resulting linear extension of $t$ is MSO-definable. This means that for each $t \in \mathbb{P}_{w<k}(\Sigma)$ and each chain partition $D_1, ..., D_k$ of $t$, there exists a particular linear extension of $t$ which is definable in $t, D_1, ..., D_k$, by some MSO formula. We skip the detailed specification of this formula and rely on the following theorem to assert its existence.

**Theorem 44.** *[4, Theorem 2.4] Let $k$ be an integer. There is some MSO formula $\theta_k(x, y, X_1, ..., X_k)$ such that for each pomset $t = (E, \preccurlyeq, \xi)$ from $\mathbb{P}_{w<k}(\Sigma)$, and each chain partition $C = \{D_1, ..., D_k\}$ of $t$, the relation $\preccurlyeq_C$ defined on pairs $e, e' \in E$ by*

$$e \preccurlyeq_C e' \Leftrightarrow (t, e, e', D_1, ..., D_k) \models \theta_k(x, y, X_1, ..., X_k)$$

*is a linear extension of $(E, \preccurlyeq)$.*

It is clear that the linear pomset $(E, \preccurlyeq_C, \xi)$ is uniquely determined by the partition $(D_1, ..., D_k)$ and the formula $\theta_k$. This word is denoted by $LinE_k(t, D_1, ..., D_k)$.

Let $k$ be a fixed integer. For any sentence $\psi$, we denote by $\overline{\psi}(X_1, ..., X_k)$ the formula obtained by replacing the occurrences of $x \preccurlyeq y$ in $\psi$ by $\theta_k(x, y, X_1, ..., X_k)$. For each pomset $t \in \mathbb{P}_{w<k}(\Sigma)$ and each chain partition $D_1, ..., D_k$ of $t$, $t$ is a model of $\overline{\psi}(D_1, ..., D_k)$ iff the linear extension $LinE_k(t, D_1, ..., D_k)$ is a model of $\psi$. In other words: $(t, D_1, ..., D_k) \models \overline{\psi}$ iff $LinE_k(t, D_1, ..., D_k) \models \psi$. Given a sentence $\psi$, we define the sentence $\widehat{\psi} := \forall X_1, ..., X_k . ChainPart(X_1, ..., X_k) \rightarrow \overline{\psi}(X_1, ..., X_k)$. Then $t \models \widehat{\psi}$ if and only if for any chain partition $D_1, ..., D_k$ of $t$, the linear extension $LinE_k(t, D_1, ..., D_k)$ satisfies the formula $\psi$.

**Lemma 45.** *Let $t = (E, \preccurlyeq, \xi)$ be a pomset from $\mathbb{P}_{w<k}$. Then $t \models \widehat{\psi}$ if and only if for all subsets $D_1, ..., D_k \subseteq E$*

$$(t, D_1, ..., D_k) \models ChainPart(X_1, ..., X_k) \textit{ implies } LinE_k(t, D_1, ..., D_k) \models \psi.$$

## 4.5 Regular Quasi-Consistent and Step-Closed Languages Are MSO Definable

In this subsection we fix some finite deterministic STS $\mathcal{A}$ and consider its pomset language $\mathcal{L}$. This means that $\mathcal{L}$ is width-bounded by some natural $k$, step-closed, quasi-consistent and regular (Theorem 22). Since $\mathcal{L}$ is weak we have $\mathrm{LE}(\mathcal{L}) = \mathrm{SL}(\mathcal{A}) \cap \Sigma^\star$. It is clear now that the word language $\mathrm{LE}(\mathcal{L})$ is regular because we can derive from $\mathcal{A}$ finite automaton that accepts $\mathrm{LE}(\mathcal{L})$. Similarly for each multiset $m \in M(\Sigma)$, the word language $L_m = \{u \in \Sigma^\star \mid u.m \in \mathrm{SE}(\mathrm{Pref}(\mathcal{L}))\}$ is regular, too. By Buchi's theorem, these word languages are MSO-definable. We let $\psi_L$ be an MSO-formula for $\mathrm{LE}(\mathcal{L})$ and for each multiset $m \in M(\Sigma)$, $\psi_m$ be an MSO-sentence for $L_m$.

Recall now that $\mathcal{L} = \mathfrak{p}_f(\mathcal{A})$ by Theorem 14. Thus a pomset $t \in \mathbb{P}(\Sigma)$ belongs to $\mathcal{L}$ if and only if $\mathrm{TE}(\mathrm{Pref}(t)) \subseteq T(\mathcal{A})$ and $\mathrm{LE}(t) \subseteq \mathrm{LE}(\mathcal{L})$. By Proposition 13 the first condition amounts to check that $t \in \mathrm{Pref}(\mathcal{L})$. Now by Lemma 17 we can conclude that $t \in \mathcal{L}$ if and only if $\mathrm{LE}(t) \subseteq \mathrm{LE}(\mathcal{L})$ and for all prefixes $t' = (E', \preccurlyeq', \xi')$ of $t$, there exists at least one linear extension $u \in \mathrm{LE}(t')$ such that $u \in L_m$, where $m = \xi(\min_{\preccurlyeq}(E \setminus E'))$. Moreover, in that case, any linear extension of $t'$ belongs to $L_m$. In that way we get the next result.

**Lemma 46.** *A pomset $t = (E, \preccurlyeq, \xi)$ from $\mathbb{P}_{w<k}(\Sigma)$ belongs to $\mathcal{L}$ if and only if it fulfills the two next requirements:*

1. *for all chain partitions $D_1, ..., D_k$ of $t$, we have $LinE_k(t, D_1, ..., D_k) \models \psi_L$*
2. *for all prefixes $t' = (E', \preccurlyeq', \xi')$ of $t$, for all subsets $M \subseteq \min_{\preccurlyeq}(E \setminus E')$, and for all chain partitions $D_1, ..., D_k$ of $t'$, we have $LinE_k(t', D_1, ..., D_k) \models \psi_m$ where $m = \xi(M)$ is the multiset of labels of $M$.*

As a consequence of the two previous lemmas, a pomset $t = (E, \preccurlyeq, \xi)$ belongs to $\mathcal{L}$ if and only if $t \models \widehat{\psi_L}$ and for any prefix $t' = (E', \preccurlyeq', \xi')$ of $t$ and for any subset $M \subseteq \min_{\preccurlyeq}(E \setminus E')$, if $m = \xi(M)$ then $t' \models \widehat{\psi_m}$. This property can obviously be encoded in MSO, which yields the next statement.

**Proposition 47.** *$\mathcal{L}$ is MSO-definable.*

## 4.6 Main Result

Theorem 22 shows that the languages of finite deterministic step transition systems are precisely the pomset languages that are width-bounded, step-closed, quasi-consistent and regular. We complete now this characterization with some logical criteria.

**Theorem 48.** *Let $\mathcal{L}$ be a width-bounded, step-closed and quasi-consistent pomset language. Then $\mathcal{L}$ is the language of a* finite *deterministic step transition system if and only if it satisfies one of the six following equivalent conditions:*

| | |
|---|---|
| (i) $\mathrm{B}(\mathcal{L})$ *is regular* | (iv) $\mathrm{B}(\mathcal{L})$ *is MSO-definable and prime-bounded* |
| (ii) $\mathcal{L}$ *is regular* | (v) $\mathcal{L}$ *is MSO-definable* |
| (iii) $\mathrm{SE}(\mathcal{L})$ *is regular* | (vi) $\mathrm{SE}(\mathcal{L})$ *is MSO-definable* |

**Proof.** By Proposition 20 we have (iii) $\Rightarrow$ (ii) because $\mathcal{L}$ is step-closed. By Proposition 47 we get that (ii) $\Rightarrow$ (v). Since $\mathcal{L}$ is step-closed, $\mathrm{SE}(\mathcal{L}) \subseteq \mathcal{L}$ hence $\mathrm{SE}(\mathcal{L}) = \mathcal{L} \cap \mathbb{S}(\Sigma)$. Since $\mathbb{S}(\Sigma)$ is MSO-definable, we have (v) $\Rightarrow$ (vi). By Cor. 41, (vi) $\Rightarrow$ (iii). Thus (ii), (iii), (v), and (vi) are equivalent.

By Theorem 36, (v) implies that $\mathrm{B}(\mathcal{L})$ is prime-bounded. It follows now from Lemma 40 that (v) $\Rightarrow$ (iv). By Cor. 39 we have (iv) $\Rightarrow$ (i). Finally Proposition 25 ensures that (i) $\Rightarrow$ (ii). ■

## 5 Conclusion

These results can be applied to some well-established concurrent semantics of Petri nets. Obviously some deterministic step transition system can be derived from any (unlabelled) Place/Transition Petri net. The resulting step language corresponds to the generalized trace language investigated in [15]. The corresponding pomset language coincides with the set of *enabled partial-words* [26], also called *firing pomsets*. Besides this semantics is related to another major net semantics based on unfoldings, more precisely on pomsets generated by prefix-closed and conflict-free unfoldings [16]. It has been proved in [16] and [26] that the set of *order extensions* of the unfolding pomsets of a Petri net coincides with the set of its firing pomsets. As a direct consequence, unfolding pomsets and firing pomsets have the same basis, and Theorem 48 establishes a Büchi-like connection for this basis.

Recall now that the pomset languages of finite (non-deterministic) step transition systems are precisely the pomset languages that are width-bounded, step-closed, and regular (Prop. 5 and Theorem 21). Until now we do not know whether the six conditions from Theorem 48 are still equivalent in the more general setting of width-bounded and step-closed pomset languages. It would be nice anyway to establish a logical characterization of the expressive power of *non-deterministic* step transition systems, too.

**Acknowledgments.** Thanks to anynomous referees for many useful remarks.

## References

1. Arnold, A.: An extension of the notion of traces and asynchronous automata. RAIRO. Theoretical Informatics and Applications 25, 355–393 (1991) (Gauthiers-Villars)
2. Bollig, B., Leucker, M.: Message-passing automata are expressively equivalent to EMSO logic. Theoretical Computer Science 358, 150–172 (2006)
3. Büchi, J.R.: Weak second-order arithmetic and finite automata. Z. Math. Logik Grundlagen Math. 6, 66–92 (1960)

4. Courcelle, B.: The monadic second-order logic of graphs X: Linear orderings. Theoretical Computer Science 160, 87–143 (1996)
5. Diekert, V., Métivier, Y.: Partial Commutation and Traces. In: Rozenberg, G., Salomaa, A. (eds.) Handbook of Formal Languages, vol. 3, pp. 457–534 (1997)
6. Dilworth, R.P.: A decomposition theorem for partially ordered sets. Ann. of Math. 51(2), 161–166 (1950)
7. Doner, J.E.: Tree acceptors and some of their applications. J. Computer and Syst. Sciences 4, 406–451 (1970)
8. Droste, M., Kuske, D.: Recognizable and logically definable languages of infinite computations in concurrent automata. International Journal of Foundations of Computer Science 9, 295–313 (1998)
9. Droste, M., Gastin, P., Kuske, D.: Asynchronous cellular automata for pomsets. Theoretical Computer Science 247, 1–38 (2000)
10. Ehrenfeucht, A., Rozenberg, G.: Partial (Set) 2-structures Part II: State spaces of concurrent systems. Acta Informatica 27, 343–368 (1990)
11. Fanchon, J., Morin, R.: Regular sets of pomsets with autoconcurrency. In: Brim, L., Jančar, P., Křetínský, M., Kucera, A. (eds.) CONCUR 2002. LNCS, vol. 2421, pp. 402–417. Springer, Heidelberg (2002)
12. Gischer, J.L.: The equational theory of pomsets. Theoretical Computer Science 61, 199–224 (1988)
13. Grabowski, J.: On partial languages. Fundamenta Informatica IV(2), 427–498 (1981)
14. Henriksen, J.G., Mukund, M., Narayan Kumar, K., Sohoni, M., Thiagarajan, P.S.: A Theory of Regular MSC Languages. Information and Computation 202, 1–38 (2005)
15. Hoogers, P.W., Kleijn, H.C.M., Thiagarajan, P.S.: A Trace Semantics for Petri Nets. Information and Computation 117, 98–114 (1995)
16. Kiehn, A.: On the interrelationship between synchronised and non synchronised behaviour of Petri Nets. Journal of Information Processing and Cybernetics. EIK 24, 3–18 (1988)
17. Kuske, D.: Asynchronous cellular automata and asynchronous automata for pomsets. In: Sangiorgi, D., de Simone, R. (eds.) CONCUR 1998. LNCS, vol. 1466, pp. 517–532. Springer, Heidelberg (1998)
18. Kuske, D.: Towards a language theory for infinite N-free pomsets. Theoretical Computer Science 299, 347–386 (2003)
19. Kuske, D., Morin, R.: Pomsets for local trace languages. Journal of Automata, Languages and Combinatorics 7, 187–224 (2002)
20. Lodaya, K., Weil, P.: Series-parallel languages and the bounded-width property. Theoretical Computer Science 237, 347–380 (2000)
21. Morin, R.: Recognizable Sets of Message Sequence Charts. In: Alt, H., Ferreira, A. (eds.) STACS 2002. LNCS, vol. 2285, pp. 523–534. Springer, Heidelberg (2002)
22. Mukund, M.: Petri Nets and Step Transition Systems. International Journal of Foundations of Computer Science 3, 443–478 (1992)
23. Pratt, V.: Modelling concurrency with partial orders. International Journal of Parallel Programming 15, 33–71 (1986)
24. Thatcher, W., Wright, J.B.: Generalized finite automata theory with an application to a decision problem of second-order logic. Math. Systems Theory 2, 57–81 (1968)
25. Thomas, W.: Languages, automata, and logic. In: Rozenberg, G., Salomaa, A. (eds.) Handbook of Formal Languages, vol. 3, pp. 389–455 (1997)
26. Vogler, W.: Modular Construction and Partial Order Semantics of Petri Nets. LNCS, vol. 625. Springer, Heidelberg (1992)

# Deficiency Zero Petri Nets and Product Form

Jean Mairesse and Hoang-Thach Nguyen

LIAFA, Université Paris Diderot - Paris 7, 75205 Paris Cedex 13
mairesse@liafa.jussieu.fr, hoang-thach.nguyen@liafa.jussieu.fr

**Abstract.** Consider a Markovian Petri net with race policy. The marking process has a "product form" stationary distribution if the probability of viewing a given marking can be decomposed as the product over places of terms depending only on the local marking. First we observe that the Deficiency Zero Theorem of Feinberg, developped for chemical reaction networks, provides a structural and simple sufficient condition for the existence of a product form. In view of this, we study the classical subclass of bounded free-choice nets. Roughly, we show that the only such Petri nets having a product form are the state machines which can alternatively be viewed as Jackson networks.

## 1 Introduction

Queueing networks, Petri nets, and chemical reaction networks, are three mathematical models of "networks", each of them with an identified community of researchers.

In queueing, the existence of "product form" Markovian networks is one of the cornerstones and jewels of the theory. Monographies are dedicated to the subsect, e.g. Kelly [16] or Van Dijk [8]. Roughly, the interest lies in the equilibrium behavior of a Markovian queueing network. The existence of such an equilibrium is equivalent to the existence of a stationary distribution $\pi$ for the queue-length process. In some remarkable cases, $\pi$ not only exists but has an explicit decomposable shape called "product form". The interest is two-fold. First, from a quantitative point of view, it makes the explicit computation of $\pi$ possible, even for large systems. Second, the product form has important qualitative implications, like the "Poisson-Input Poisson-Output" Theorems. Consequently important and lasting efforts have been devoted to the quest for product form queueing networks.

It is attractive and natural to try to develop an analog theory for Markovian Petri nets, with the marking process replacing the queue-length process. There is indeed a continueing string of research on this topic since the 90ies, e.g. [5, 6, 9, 11, 13, 14, 18]. Tools have been developped in the process which build on classical objects of Petri net theory (e.g. closed support T-invariants). The most accomplished results are the ones in [13].

In chemistry and biology, has emerged the model of chemical reaction networks. Such a network is specified by a finite set of reactions between species of the type "$2A + B \rightarrow C$", meaning that two molecules of $A$ can interact with one molecule of $B$ to create one molecule of $C$. The dynamics of such models is either deterministic or stochastic, see [17].

Deterministic models are the most studied ones, they correspond to coupled sets of ordinary differential equations. The most significant result is arguably the Deficiency Zero Theorem of Feinberg [10]. Deficiency Zero is a structural property which can be very easily checked knowing the shape of the reactions in a chemical network. It does not refer to any assumption on the associated dynamics. Feinberg Theorem states that if a network satisfies the Deficiency Zero condition, then the associated deterministic dynamic model has remarkable stability properties. An intermediate result is to prove that a set of non-linear equations (NLE) have a strictly positive solution.

Stochastic models of chemical reaction networks correspond to continuous-time Markov processes of a specific shape. Such models were considered in Chapter 8 of the seminal book by Kelly [16]. There it is proved that if a set of non-linear "traffic equations" (NLTE) have a strictly positive solution then the Markov process has a product form.

How does Feinberg result connect with product form Markovian Petri nets ?

A first observation is that chemical reaction networks and Petri nets are two different descriptions of the same object. This has been identified by different authors in the biochemical community, see for instance [3] and the references therein. Conversely, Petri nets were originally introduced by Carl Adam Petri to model chemical processes, see [20].

A second observation was made recently by Anderson, Craciun, and Kurtz [2]. They observe that the NLE of Feinberg and the NLTE of Kelly are the same. It implies that if a chemical network has deficiency zero, then the stochastic dynamic model has a product form.

In the present paper, we couple the two observations together. The Deficiency Zero condition provides a sufficient condition for a Markovian Petri net to have a product form. This condition is structural and very easy to check. Moreover it is at least as strong as the criteria which were known in the Petri net literature. More precisely, we prove that if a Markovian Petri net satisfies the structural condition in [13] then it has deficiency zero.

The class of Petri nets whose Markovian version have a product form is an interesting one. It is therefore natural to study how this class intersects with the classical families of Petri nets: state machines and free-choice Petri nets.

The central result that we prove is, in a sense, a negative result. We show that within the class of free-choice Petri nets, the only ones which have a product form are closely related to state machines. We also show that the Markovian state machines are "equivalent" to Jackson networks. The latter form the most basic and classical example of product form queueing networks.

## 2 Model

We use the notation $\mathbb{R}^* = \mathbb{R} - \{0\}$. The coordinate-wise ordering of $\mathbb{R}^k$ is denoted by the symbol $\leqslant$. We say that $x \in \mathbb{R}^k$ is *strictly positive* if $x_i > 0$ for all $i$. We denote by $\mathbf{1}_S$ the indicator function of $S$, that is the mapping taking value 1 inside $S$ and 0 outside.

## 2.1 Petri Nets

Our definition of Petri net is standard, with weights on the arcs.

**Definition 1 (Petri net).** *A* Petri net *is a 6-tuple* $(\mathcal{P}, \mathcal{T}, \mathcal{F}, I, O, M_0)$ *where:*

- $(\mathcal{P}, \mathcal{T}, \mathcal{F})$ *is a directed bipartite graph, that is,* $\mathcal{P}$ *and* $\mathcal{T}$ *are non-empty and finite disjoint sets, and* $\mathcal{F}$ *is a subset of* $(\mathcal{P} \times \mathcal{T}) \cup (\mathcal{T} \times \mathcal{P})$ ;
- $I : \mathcal{T} \to \mathbb{N}^{\mathcal{P}}$ *and* $O : \mathcal{T} \to \mathbb{N}^{\mathcal{P}}$ *are such that* $[I(t)_p > 0 \Leftrightarrow (p,t) \in \mathcal{F}]$ *and* $[O(t)_p > 0 \Leftrightarrow (t,p) \in \mathcal{F}]$ ;
- $M_0$ *belongs to* $\mathbb{N}^{\mathcal{P}}$.

The elements of $\mathcal{P}$ are called *places*, those of $\mathcal{T}$ are called *transitions*. The 5-tuple $(\mathcal{P}, \mathcal{T}, \mathcal{F}, I, O)$ is called the *Petri graph*. The vectors $I(t)$ and $O(t)$, $t \in \mathcal{T}$, are called the *input bag* and the *output bag* of the transition $t$. An element of $\mathbb{N}^{\mathcal{P}}$ is called a *marking*, and $M_0$ is called the *initial marking*.

Petri nets inherit the usual terminology of graph theory. Graphically, a Petri net is represented by a directed graph in which places are represented by circles and transitions by rectangles. The initial marking is also materialized: if $M_0(p) = k$, then $k$ tokens are drawn inside the circle $p$. See Figure 1 for an example.

A Petri net is a dynamic object. The Petri graph always remains unchanged, but the marking evolves according to the *firing rule*. A transition $t$ is *enabled* in the marking $M$ if $M \geq I(t)$, then $t$ may *fire* which transforms the marking from $M$ into

$$M' = M - I(t) + O(t) \, .$$

We write $M \xrightarrow{t} M'$. A marking $M'$ is *reachable* from a marking $M$ if there exists a sequence of transitions $t_1, ..., t_k$, and a sequence of markings $M_1, ..., M_{k-1}$, such that $M \xrightarrow{t_1} M_1 \xrightarrow{t_2} \cdots \xrightarrow{t_{k-1}} M_{k-1} \xrightarrow{t_k} M'$. We denote by $\mathcal{R}(M)$ the set of markings which are reachable from $M$.

**Definition 2 (Marking graph).** *The* marking graph *of a Petri net with initial marking* $M_0$ *is the directed graph with*

- *nodes:* $\mathcal{R}(M_0)$ , *arcs:* $M \to M'$ *if* $\exists t \in \mathcal{T},\ M \xrightarrow{t} M'$.

The marking graph defines the state space on which the marking may evolve. Observe that the marking graph may be finite or infinite. In Section 2.3, we will define a *Markovian Petri net* as a continuous-time Markovian process evolving on the marking graph.

The analysis of Petri nets relies heavily on linear algebra techniques, the central object being the incidence matrix.

The *incidence matrix* $N$ of the Petri net $(\mathcal{P}, \mathcal{T}, \mathcal{F}, I, O, M_0)$ is the $(\mathcal{P} \times \mathcal{T})$-matrix $N$ defined by:

$$N_{s,t} = O(t)_s - I(t)_s \, . \tag{1}$$

**Example.** Figure 1 represents a Petri net with places $\{p_1, p_2, p_3, p_4\}$ and transitions $\{t_1, t_2, t_3, t_4\}$. The initial marking is $M_0 = (2, 1, 0, 1)$. The input and output bags are:

$$\begin{aligned} &I(t_1) = (2,0,0,0),\ O(t_1) = (0,2,0,0), \quad I(t_2) = (0,2,0,0),\ O(t_2) = (2,0,0,0)\,, \\ &I(t_3) = (1,0,1,0),\ O(t_3) = (0,1,0,1), \quad I(t_4) = (0,1,0,1),\ O(t_4) = (1,0,1,0)\,. \end{aligned}$$

The weights different from 1 are represented on the arcs.

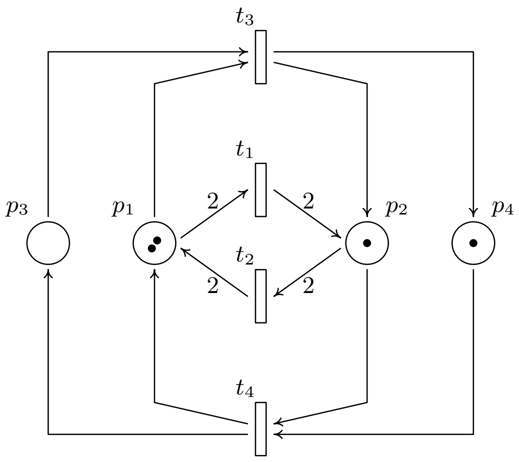


**Fig. 1.** Petri net

The reachable markings are $M_0$, $M_1 = (0, 3, 0, 1)$, $M_2 = (3, 0, 1, 0)$, and $M_3 = (1, 2, 1, 0)$ . The marking graph is represented on Figure 2.

$$\begin{tikzcd} M_0 \arrow[r, bend left, "t_1"] \arrow[d, bend left, "t_4"] & M_1 \arrow[l, bend left, "t_2"] \arrow[d, bend left, "t_4"] \\ M_2 \arrow[u, bend left, "t_3"] \arrow[r, bend left, "t_1"] & M_3 \arrow[l, bend left, "t_2"] \arrow[u, bend left, "t_3"] \end{tikzcd}$$

**Fig. 2.** Marking graph

The incidence matrix of the Petri net is:

$$N = \begin{pmatrix} -2 & 2 & -1 & 1 \\ 2 & -2 & 1 & -1 \\ 0 & 0 & -1 & 1 \\ 0 & 0 & 1 & -1 \end{pmatrix} . \qquad (2)$$

## 2.2 Chemical Reaction Networks

Coinsider a Petri net $(\mathcal{P}, \mathcal{T}, \mathcal{F}, I, O, M_0)$. If no two transitions have the same input/output bags, we can identify each $t$ with the ordered pair $(I(t), O(t))$. The Petri net can then be viewed as a triple $(\mathcal{P}, \mathcal{T} \subset \mathbb{N}^{\mathcal{P}} \times \mathbb{N}^{\mathcal{P}}, M_0 \in \mathbb{N}^{\mathcal{P}})$. (In particular, the flow relation $\mathcal{F}$ is encoded in $\mathcal{T}$.)

Petri nets have appeared with this presentation in different contexts and under different names: *vector addition systems* (see for instance [21]), or *chemical reaction networks* (see for instance [10, 2]).

In the chemical context, the elements of $\mathcal{P}$ are species. The marking is the number of molecules of the different species. The elements of $\mathcal{T}$ are *reactions.* A reaction $(c, d) \in \mathbb{N}^{\mathcal{P}} \times \mathbb{N}^{\mathcal{P}}$ is represented as follows:

$$\sum_{p \in \mathcal{P}} c_p p \longrightarrow \sum_{p \in \mathcal{P}} d_p p.$$

**Example.** The "chemical" form of the Petri net in Figure 1 is:

$$2p_1 \rightleftarrows 2p_2 \,, \qquad p_1 + p_3 \rightleftarrows p_2 + p_4 \,.$$

Let us now introduce some notions and definitions, which are borrowed from the chemical literature.

**Definition 3 (Reaction graph).** *Let $(\mathcal{P}, \mathcal{T} \subset \mathbb{N}^{\mathcal{P}} \times \mathbb{N}^{\mathcal{P}}, M)$ be a Petri net. A* complex *is a vector $u$ in $\mathbb{N}^{\mathcal{P}}$ such that: $\exists v \in \mathbb{N}^{\mathcal{P}},\ (u, v) \in \mathcal{T}$ or $(v, u) \in \mathcal{T}$. The set of all complexes is denoted by $\mathcal{C}$. The* reaction graph *associated to the Petri net is the directed graph with*

– *nodes: $\mathcal{C}$, arcs: $u \rightarrow v$ if $(u, v) \in \mathcal{T}$.*

**Definition 4 (Deficiency).** *Consider a Petri net with set of complexes $\mathcal{C}$ and incidence matrix $N$. The* deficiency *of the Petri net is*

$$\delta = |\mathcal{C}| - \ell - \mathit{rank}(N) \,,$$

*where $\ell$ is the number of connected components of the reaction graph.*

**Proposition 1 (Feinberg [10]).** *The deficiency of a Petri net is always non-negative.*

Of particular importance are the Petri nets with **deficiency 0**. This class will be central in the study of Markovian Petri nets having a product form. This will be discussed in detail in Section 3.

**Definition 5 (Weak reversibility).** *A Petri net is* weakly reversible *if every connected component of the reaction graph is strongly connected.*

Weak reversibility is a restrictive property. In Section 4.3, we show that the class of weakly reversible free-choice nets is in a sense limited to generalized state machines (each transition has at most one input and one output place).

Elementary circuits of the reaction graph can be identified with the so-called "minimal closed support T-invariants" of the Petri net literature (see [5]). In particular, a Petri net is weakly reversible if and only if it is covered by minimal closed support T-invariants. Such Petri nets are called *$\Pi$-nets* in [13].

Weak reversibility and deficiency 0 are two independent properties. The Petri net of Figure 1 is weakly reversible and has deficiency 0. If we remove one transition, $t_1$ for instance, weak reversibility disappears, yet the deficiency remains 0. The Petri net of Figure 3 is weakly reversible and has deficiency 1. If we remove one transition, the new Petri net is neither weakly reversible nor of deficiency 0.

*Algorithmic complexity.* Weak reversibility and deficiency 0 are algorithmically simple to check. Let us determine the time complexity of the algorithms with respect to the size of the Petri net (number of places and transitions, and the unary representation of weights).

Observe first that the number of complexes is bounded by $2|\mathcal{T}|$. Building the reaction graph from the Petri graph can be done in time $\mathcal{O}(|\mathcal{P}||\mathcal{T}|^2)$. A depth-first-search algorithm on the reaction graph enables to check the weak reversibility and to compute the number of connected components ($\ell$). The DFS algorithm runs in time $\mathcal{O}(|\mathcal{P}||\mathcal{T}|)$. Computing the rank of the incidence matrix can be done in time $\mathcal{O}(|\mathcal{P}||\mathcal{T}|^2)$ using a Gaussian elimination.

Globally the complexity is $\mathcal{O}(|\mathcal{P}||\mathcal{T}|^2)$ for computing the deficiency as well as for checking weak-reversibility.

## 2.3 Markovian Petri Nets with Race Policy

A Petri net is a logical object with no physical time involved. There exist several alternative ways to define timed models of Petri nets, see for instance [1, 4]. We consider the model of Markovian Petri nets with race policy. The rough description is as follows.

With each enabled transition is associated a "countdown clock" whose positive initial value is set at random. When a clock reaches 0, the corresponding transition fires. This changes the set of enabled transitions and all the clocks get reinitialized. The initial values of the clocks are chosen independently, according to an exponential distribution whose rate depends on the transition and on the current marking. With probability 1, no two clocks reach zero at the same time so the model is unambiguously defined. Enabled transitions are involved in a "race": the transition to fire is the one whose clock will reach zero first.

We now proceed to a formal definition of the model.

When $I(t) = O(t)$, the firing of transition $t$ does not modify the marking. For simplicity, we assume from now on that $I(t) \neq O(t)$ for all $t$.

**Definition 6 (Markovian Petri net with race policy).** *A* Markovian Petri net (with race policy) *is formed by a Petri net* $(\mathcal{P}, \mathcal{T}, \mathcal{F}, I, O, M_0)$ *and a set of* rate functions $(\mu_t)_{t\in\mathcal{T}}$*,* $\mu_t : \mathcal{R}(M_0) \to \mathbb{R}_+^*$*, satisfying*

$$\mu_t(M) = \begin{cases} \kappa_t \Psi(M - I(t))\Phi(M) & \textit{if } M \geqslant I(t) \\ 0 & \textit{otherwise} \end{cases}, \tag{3}$$

*for some constants $\kappa_t \in \mathbb{R}_+^*$, $t \in \mathcal{T}$, and some functions $\Psi$ and $\Phi$ valued in $\mathbb{R}_+^*$. The marking evolves as a continuous-time jump Markov process with state space $\mathcal{R}(M_0)$ and infinitesimal generator $Q = (q_{M,M'})_{M,M'}$, given by*

$$q_{M,M'} = \sum_{t:M \xrightarrow{t} M'} \mu_t(M)\,. \tag{4}$$

The shape (4) for the infinitesimal generator is the transcription of the informal description given at the beginning of the section.

The condition (3) for the rate functions $(\mu_t)_{t\in\mathcal{T}}$ is the same as the one in [14] and [13, Section 2]. (In [13, Section 3], an even more general shape for the rate function is considered.) Condition (3) is specifically cooked up in order for the product form result of Theorem 1 to hold, which explains its artificial shape. This general condition englobes two classical types of rate functions: the constant rates and the mass-action rates.

*Constant rates.* In the Petri net literature, the standard assumption is that the firing rates are constant:

$$\exists \kappa_t \in \mathbb{R}_+^*,\ \forall M \in \mathcal{R}(M_0), I(t) \geqslant M, \qquad \mu_t(M) = \kappa_t\,. \tag{5}$$

*Mass-action rates.* In the chemical literature, the rate is often proportional to the number of different subsets of tokens (i.e. molecules) that can be involved in the firing (i.e. reaction). More precisely:

$$\forall M \in \mathcal{R}(M_0), I(t) \geqslant M, \qquad \mu_t(M) = \kappa_t \prod_{p:I(t)_p \neq 0} \frac{M_p!}{(M_p - I(t)_p)!}\,. \tag{6}$$

Such rates are said to be of *mass-action* form and the corresponding stochastic process has *mass-action kinetics.* To obtain (6) from (3), set $\Phi, \Psi^{-1} : \mathbb{N}^{\mathcal{P}} \to \mathbb{R}_+^*, x \mapsto \prod_p x_p!$.

## 3 Product Form Results

We are interested in the equilibrium behavior of Markovian Petri nets. This section presents the product form results which exist in the literature. We gather results which were spread out, obtained independently either in the Petri net community, or in the chemical one.

Let $Q$ be the infinitesimal generator of the marking process. An invariant measure $\pi$ of the process is characterized by the balance equations $\pi Q = 0$, that is: $\forall x \in \mathcal{R}(M_0)$,

$$\pi(x) \sum_{t:x \geqslant I(t)} \mu_t(x) = \sum_{t:x\geqslant O(t)} \pi(x + I(t) - O(t))\mu_t(x + I(t) - O(t))\,. \tag{7}$$

A stationary distribution is an invariant probability measure. It is characterized by $\pi Q = 0$, $\sum_x \pi(x) = 1$. If $\pi$ is an invariant measure and $K = \sum_{x\in\mathcal{R}(M_0)} \pi(x) < +\infty$, then $\pi/K = (\pi(x)/K)_x$ is a stationary distribution.

When the marking graph is strongly connected, the marking process is irreducible. It follows from basic Markovian theory that the stationary distribution is unique when it exists (the ergodic case). When the state space is finite, irreducibility implies ergodicity.

**Definition 7 (Non-linear traffic equations).** *Consider a Markovian Petri net with general rates. Let $\mathcal{C}$ be the set of complexes. We call* non-linear traffic equations (NLTE) *the equations over the unknowns $(x_p)_{p\in\mathcal{P}}$ defined by: $\forall C \in \mathcal{C}$,*

$$\prod_{p:C_p\neq 0} x_p^{C_p} \sum_{t:I(t)=C} \kappa_t = \sum_{t:O(t)=C} \kappa_t \prod_{p:I(t)_p\neq 0} x_p^{I(t)_p} \,. \tag{8}$$

*(With the convention that the product over an empty set of indices equals 1.)*

The NLTE can be viewed as a kind of balance equations (what goes in equals what goes out) at the level of complexes. Their central role appears in next theorem which is essentially due to Kelly [16, Theorem 8.1] (see also [2, Theorem 4.1]). In Kelly's book, the setting is more restrictive, but the proof carries over basically unchanged. For the sake of completeness, we recall the proof.

**Theorem 1 (Kelly).** *Consider a Markovian Petri net. Assume that the NLTE (8) admit a strictly positive solution $(u_p)_{p\in\mathcal{P}}$. Then the marking process of the Petri net has an invariant measure $\pi$ defined by: $\forall x \in \mathcal{R}(M_0)$,*

$$\pi(x) = \Phi(x)^{-1} \prod_{p\in\mathcal{P}} u_p^{x_p} \,. \tag{9}$$

We say that $\pi$ has a *product form*: $\pi(x)$ decomposes as a product over the places $p$ of terms depending only on the local marking $x_p$.

Observe that $\pi(x) > 0$ for all $x$ in (9). In particular it implies that the marking process is irreducible. On the other hand, the measure defined in (9) may have a finite or infinite mass. When it has a finite mass, the marking process is ergodic, and the normalization of $\pi$ is the unique stationary distribution.

In the case of mass-action rates (6), we get

$$\sum_{x\in\mathcal{R}(M_0)} \pi(x) = \sum_{x\in\mathcal{R}(M_0)} \prod_{p\in\mathcal{P}} \frac{u_p^{x_p}}{x_p!} \leqslant \sum_{x\in\mathbb{N}^{\mathcal{P}}} \prod_{p\in\mathcal{P}} \frac{u_p^{x_p}}{x_p!} = \exp(\sum_p u_p) < +\infty \,.$$

So we are always in the ergodic case. For constant rates (5), if the state space $\mathcal{R}(M_0)$ is infinite, the ergodicity depends on the value of the constants $\kappa_t$.

*Proof.* It suffices to verify that $\pi$ of the form (9) satisfies (7) when $(u_p)_{p\in\mathcal{P}}$ is a solution to (8).

In (7), by replacing $\pi$ and $\mu_t$ with the right-hand sides of (9) and (3), we obtain, after simplification:

$$\sum_{t:x\geqslant I(t)} \kappa_t \Psi(x - I(t))\Phi(x)\Phi(x)^{-1} \prod_p u_p^{x_p} =$$
$$\sum_{t:x\geqslant O(t)} \kappa_t \Psi(x - O(t))\Phi(x + I(t) - O(t))\Phi(x + I(t) - O(t))^{-1} \prod_p u_p^{x_p + I(t)_p - O(t)_p}$$

which is equivalent to

$$\sum_{x \geqslant C} \sum_{t:I(t)=C} \kappa_t \Psi(x-C) \prod_p u_p^{x_p} = \sum_{x \geqslant C} \sum_{t:O(t)=C} \kappa_t \Psi(x-C) \prod_p u_p^{x_p+I(t)_p-C_p} .$$

A sufficient condition for the above equality to hold is to have, for each $C \in \mathcal{C}$,

$$\sum_{t:I(t)=C} \kappa_t \Psi(x-C) \prod_p u_p^{x_p} = \sum_{t:O(t)=C} \kappa_t \Psi(x-C) \prod_p u_p^{x_p+I(t)_p-C_p} ,$$

after simplification we get,

$$\prod_{p:C_p \neq 0} u_p^{C_p} \sum_{t:I(t)=C} \kappa_t = \sum_{t:O(t)=C} \kappa_t \prod_{p:I(t)_p \neq 0} u_p^{I(t)_p} .$$

This last set of equations means precisely that $(u_p)_{p \in \mathcal{P}}$ is a solution to the NLTE. □

Theorem 1 is the core result. Below, all the developments consist in determining conditions under which Theorem 1 applies. More precisely, we want conditions on the model ensuring the existence of a strictly positive solution to the NLTE and the finiteness of the measure $\pi$. The ideal situation is as follows:

– *structural* properties of the Petri net (i.e. independent of the firing rates) ensure the existence of a strictly positive solution to the NLTE;
– conditions on the firing rates ensure the finiteness of the measure $\pi$.

Solving the non-linear traffic equations is still a challenging task. We may avoid a direct attack to these equations by considering a simpler system of equations called the linear traffic equations.

**Definition 8 (Linear traffic equations).** *We call* linear traffic equations (LTE) *the equations over the unknowns $(y_C)_{C \in \mathcal{C}}$ defined by: $\forall C \in \mathcal{C}$,*

$$y_C \sum_{t:I(t)=C} \kappa_t = \sum_{t:O(t)=C} \kappa_t y_{I(t)} . \tag{10}$$

*Furthermore, if $\emptyset \in \mathcal{C}$, then $y_\emptyset = 1$.*

The following proposition provides a simple and structural criterium for the existence of a strictly positive solution to the LTE.

**Proposition 2.** *The following statements are equivalent:*

– *$\exists (\kappa_t)_{t \in \mathcal{T}}$ such that the equations (10) have a strictly positive solution.*
– *$\forall (\kappa_t)_{t \in \mathcal{T}}$, the equations (10) have a strictly positive solution.*
– *The Petri net is weakly reversible.*

Proofs can be found in [5, Theorem 3.5] or [10, Corollary 4.2]. We recall the argument from [5] which is simple and illuminating.

*Proof.* The *reaction process* is a continuous-time Markov process, analog to the marking process, except that it is built on the reaction graph instead of the marking graph. More precisely, the state space is the set of complexes $\mathcal{C}$ and the infinitesimal generator $\widetilde{Q} = (\widetilde{q}_{u,v})_{u,v}$ is defined by

$$\widetilde{q}_{u,v} = \sum_{t:I(t)=u,O(t)=v} \kappa_t \ .$$

(The discrete-time version of this process was introduced in [14] under the name "routing process".) The key observation is that the LTE (10) are precisely the balance equations $y\widetilde{Q} = 0$ of the reaction process. The result now follows using standard Perron-Frobenius theory. □

The NLTE and the LTE are clearly linked.

**Lemma 1.** *If the NLTE (8) have a strictly positive solution $u = (u_p)_{p\in\mathcal{P}}$, then $v = (v_C)_{C\in\mathcal{C}}$,*

$$v_C = \prod_{p:C_p\neq 0} u_p^{C_p} \ ,$$

*is a strictly positive solution to the LTE (10).*

So weak reversibility is a necessary condition to have a strictly positive solution to the NLTE, and to be able to apply Theorem 1. Unfortunately, it is not a sufficient condition as shown by the following example.

**Example.** Let us consider a Markovian Petri net whose underlying Petri graph is shown in Figure 3, and is equivalently defined by the chemical reactions:

$$p_1 \rightleftarrows p_2 \qquad p_3 \rightleftarrows p_4 \qquad p_1 + p_3 \rightleftarrows p_2 + p_4 \ .$$

This is a weakly reversible Petri net, thus its LTE always have a strictly positive solution regardless of the choice of the constants $\kappa_t$. The NLTE are:

$$\kappa_1 x_1 = \kappa_2 x_2 \qquad \kappa_3 x_3 = \kappa_4 x_4 \qquad \kappa_5 x_1 x_3 = \kappa_6 x_2 x_4 \ . \tag{11}$$

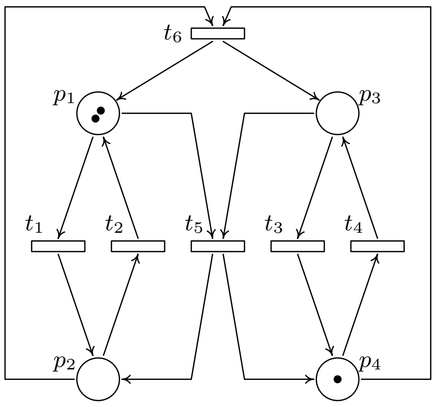


**Fig. 3.** A weakly reversible Petri net

The system (11) does not always have a strictly positive solution. For example, set $\kappa_1 = \kappa_2 = \kappa_3 = \kappa_4 = \kappa_5 = 1$, and $\kappa_6 = 2$. Any solution to (11) must satisfy either $x_1 = x_2 = 0$ or $x_3 = x_4 = 0$.

Depending on the values of the constants $(\kappa_t)_t$, the Markovian Petri net may or may not have a product form invariant measure. Anticipating on Theorem 3, the deficiency of the Petri net has to be different from 0, and indeed we have a deficiency which is equal to 1.

So now the goal is to find additional conditions on top of weak reversibility to ensure the existence of a product form.

An early result in this direction appears in Coleman, Henderson and Taylor [6, Theorem 3.1]. The condition is not structural (i.e. rate dependent) and not very tractable. Next result, due to Haddad, Moreaux, Sereno, and Silva [13, Theorem 9], provides a structural sufficient condition.

**Proposition 3.** *Consider a Markovian Petri net (set of complexes $\mathcal{C}$). Assume that the Petri net is weakly reversible. Let $N$ be the incidence matrix of the Petri net. Let $A$ be the node-arc incidence matrix of the reaction graph, that is the $(\mathcal{C} \times \mathcal{T})$-matrix defined by $A_{u,t} = -\mathbf{1}_{\{I(t)=u\}} + \mathbf{1}_{\{O(t)=u\}}$. Assume that there exists a $\mathbb{Q}$-valued $(\mathcal{C} \times \mathcal{P})$-matrix $B$ such that $BN = A$. Then the marking process has an invariant measure $\pi$ given by: $\forall x \in \mathcal{R}(M_0)$,*

$$\pi(x) = \Phi(x)^{-1} \prod_{p \in \mathcal{P}} \Big( \prod_{C \in \mathcal{C}} v_C^{B_{C,p}} \Big)^{x_p} ,$$

*where $v$ is a strictly positive solution to the LTE.*

Independently of the efforts in the Petri net community ([6, 13]), the following result was proved on the chemical side by Feinberg [10, Theorem 5.1].

**Theorem 2 (Feinberg).** *Consider a Markovian Petri net. Assume that the Petri net has deficiency 0. Then the NLTE have a strictly positive solution if and only if the network is weakly reversible.*

By combining Theorems 1 and 2, we obtain the following result.

**Theorem 3.** *Consider a Petri net which is weakly reversible and has deficiency 0. Consider any associated Markovian Petri net. The NLTE have a strictly positive solution $(u_p)_p$ and the marking process has a product form invariant measure:*

$$\pi(x) = \Phi(x)^{-1} \prod_{p \in \mathcal{P}} u_p^{x_p} .$$

*If we assume furthermore that the rates are of mass-action type (6), then the marking process is ergodic and its stationary distribution is:*

$$\pi(x) = C \prod_{p \in \mathcal{P}} \frac{u_p^{x_p}}{x_p!} ,$$

*where $C = \left(\sum_x u_p^{x_p}/x_p!\right)^{-1}$.*

The above result is interesting. Indeed, the "deficiency 0" condition is structural and very simple to handle. We now prove that the result in Theorem 3 is at least as strong as the one in Proposition 3.

**Proposition 4.** *Consider a weakly reversible Petri net. Assume that there exists a $(\mathcal{C} \times \mathcal{P})$-matrix $B$ such that $BN = A$ (with the notations of Prop. 3). Then the Petri net has deficiency 0.*

*Proof.* Let $\ell$ be the number of connected components of the reaction graph. Proposition 1 states that $\text{rank}(N) \leqslant |\mathcal{C}| - \ell$, and the deficiency is 0 iff $\text{rank}(N) = |\mathcal{C}| - \ell$. Since $BN = A$, we also have $\text{rank}(A) \leqslant \text{rank}(N)$. So if we can prove that $\text{rank}(A) = |\mathcal{C}| - \ell$, it will imply that $\text{rank}(N) = |\mathcal{C}| - \ell$, and that the Petri net has deficiency 0.

Assume that $\ell = 1$. Consider $x \in \mathbb{R}^{\mathcal{C}} - \{(0, \ldots, 0)\}$ such that $xA = (0, \ldots, 0)$. Let $C$ be such that $x_C \neq 0$. Consider $D \in \mathcal{C}$. Since $\ell = 1$, there exists an undirected path $(C = C_0)$ — $C_1$ — $\cdots$ — $(C_k = D)$ in the reaction graph. Assume wlog that $C_i \to C_{i+1}$ and let $t_i \in \mathcal{T}$ be such that $I(t_i) = C_i, O(t_i) = C_{i+1}$. By definition of $A$, we have $(xA)_{t_i} = x_{C_{i+1}} - x_{C_i}$. So we have $x_{C_i} = x_{C_{i+1}}$ for all $i$, and $x_C = x_D$. We have proved that

$$xA = (0, \ldots, 0) \implies x \in \mathbb{R}(1, \ldots, 1) ,$$

and in particular $\text{rank}(A) = |\mathcal{C}| - 1$. For a general value of $\ell$, we get similarly that $\text{rank}(A) = |\mathcal{C}| - \ell$. □

We believe that the converse of Proposition 4 is also true, but we have not been able to prove it.

## 4 Markovian Free-Choice Nets and Product Form

The class of Petri nets whose Markovian version have a product form is an interesting one. It is therefore natural to study how this class intersects with the classical families of Petri nets: state machines and free-choice Petri nets.

The central result of this section is, in a sense, a negative result. We show that within the class of free-choice Petri nets, the only ones which are weakly reversible are closely related to state machines. We also show that the Markovian state machines are "equivalent to" Jackson networks. The latter form the most basic and classical example of product form queueing networks.

From now on, we consider only *non-weighted* Petri nets, that is Petri nets with $I, O : \mathcal{T} \to \{0,1\}^{\mathcal{P}}$. In this case, the input/output bags can be retrieved from the flow relation $\mathcal{F}$ and we can define the Petri net as a quadruple $(\mathcal{P}, \mathcal{T}, \mathcal{F}, M_0)$. We also identify complexes and subsets of $\mathcal{P}$.

For a node $x \in \mathcal{T} \cup \mathcal{P}$, set ${}^\bullet x = \{y : \ (y, x) \in \mathcal{F}\}$ and $x^\bullet = \{y : \ (x, y) \in \mathcal{F}\}$. For a set of nodes $S \subset \mathcal{T} \cup \mathcal{P}$, set ${}^\bullet S = \bigcup_{x \in S} {}^\bullet x$ and $S^\bullet = \bigcup_{x \in S} x^\bullet$.

## 4.1 State Machines

**Definition 9 (State machine and generalized state machine).** *A non-weighted Petri net $\mathcal{N} = (\mathcal{P}, \mathcal{T}, \mathcal{F}, M_0)$ is a:*

- State machine (SM) *if for all transition $t$, $|^\bullet t| = |t^\bullet| = 1$;*
- Generalized state machine (GSM) *if for all transition $t$, $|^\bullet t| \leq 1$, $|t^\bullet| \leq 1$.*

**Definition 10 (Associated state machine).** *Given a generalized state machine $\mathcal{N} = (\mathcal{P}, \mathcal{T}, \mathcal{F}, M_0)$, the* associated state machine *is $\mathcal{N}' = (\mathcal{P}', \mathcal{T}, \mathcal{F}', M'_0)$ in which:*

- $\mathcal{P}' = \mathcal{P} \cup \{p\}, p \notin \mathcal{P}$ ,
- $\mathcal{F}' = \mathcal{F} \cup \{(p,t), t \in \mathcal{T}, |^\bullet t| = 0\} \cup \{(t,p), t \in \mathcal{T}, |t^\bullet| = 0\}$ ,
- $\forall x \in \mathcal{P},\ M'_0(x) = M_0(x), \qquad M'_0(p) = 0$ .

Figure 4 shows a SM, a GSM and its associated SM.

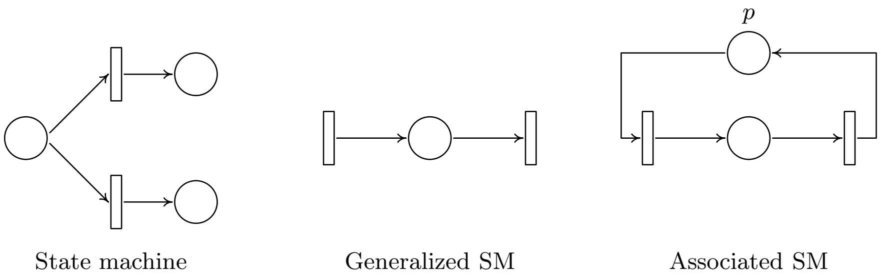


**Fig. 4.** State machine, generalized state machine and associated state machine

**Lemma 2.** *The reaction graph and the Petri graph of a state machine are isomorphic. The reaction graph of a GSM and the Petri graph of its associated SM are isomorphic.*

*Proof.* In a SM, each complex is just one place. Starting from the Petri graph and replacing $[p \to t \to q]$, $p, q \in \mathcal{P}$, $t \in \mathcal{T}$, by $[p \to q]$, we get the reaction graph. For GSM, the mapping is the same with the empty complex corresponding to the "new" place in the associated SM. □

**Corollary 1.** *A SM is weakly reversible iff each connected component is strongly connected. A GSM is weakly reversible iff in the associated SM, each connected component is strongly connected.*

In a SM, the complexes are the places. So the NLTE and the LTE coincide exactly. For a GSM, the complexes are the places and the empty set. With the convention $y_\emptyset = 1$, we still have that the NLTE and the LTE coincide. Next proposition follows.

**Proposition 5.** *Consider a weakly reversible GSM. For every rates $(\kappa_t)_t$, the NLTE have a strictly positive solution.*

*Proof.* In the weakly reversible case, the LTE have a strictly positive solution for every choice of the rates, Proposition 2. Therefore the NLTE have a strictly positive solution for every choice of the rates. $\square$

The above proof does not require Feinberg's Theorem 2. However, it turns out that the deficiency is 0, which provides a second proof of Prop. 5 using Theorem 2.

**Proposition 6.** *Generalized state machines have deficiency 0.*

We first prove a weaker statement:

**Lemma 3.** *State machines have deficiency 0.*

*Proof.* Consider a SM with incidence matrix $N$. Let $A$ be the node-arc incidence matrix of the reaction graph defined as in Proposition 3. Using Lemma 2, we get immediately that $A = N$.

Assume that the SM is connected: $\ell = 1$. It suffices to prove that $\text{rank}(A) = |\mathcal{C}| - 1$. Consider $x \in \mathbb{R}^{\mathcal{C}} - \{(0,\ldots,0)\}$ such that $xA = (0,\ldots,0)$. Let $C$ be such that $x_C \neq 0$. Let $D$ be directly linked to $C$. Without loss of generality, suppose that $C = I(t), D = O(t)$ for some $t$. By definition of $A$, we have $(xA)_t = x_D - x_C$, so $x_D = x_C$. Since $\ell = 1$, by recursively applying this argument, we have $x_C = x_D$ for every $D$. Hence:

$$xA = (0,\ldots,0) \implies x \in \mathbb{R}(1,\ldots,1)\,,$$

and in particular $\text{rank}(A) = |\mathcal{C}| - 1$. For a general value of $\ell$, we get similarly that:

$$\text{rank}(A) = |\mathcal{C}| - \ell\,. \tag{12}$$

In the connected case, the above formula can also be viewed as a special instance of the Rank Theorem for well-formed free-choice nets, see [7, Chapter 6]. Using (12), we obtain for the deficiency:

$$\delta = |\mathcal{C}| - \text{rank}(N) - \ell = |\mathcal{C}| - (|\mathcal{C}| - \ell) - \ell = 0\,. \qquad \square$$

*Proof (Proposition 6).* Consider a GSM $\mathcal{N}$ and its associated SM $\mathcal{N}'$.

Call $\mathcal{C}$ (resp. $\mathcal{C}'$), $N$ (resp. $N'$) and $\ell$ (resp. $\ell'$) the set of complexes, the incidence matrix and the number of connected components of the reaction graph of $\mathcal{N}$ (resp. $\mathcal{N}'$).

Since $\mathcal{N}$ and $\mathcal{N}'$ have the same reaction graph (Lemma 2), we have:

$$|\mathcal{C}| = |\mathcal{C}'|\,,\ \ell = \ell'\,. \tag{13}$$

By construction of $\mathcal{N}'$, $N'$ is $N$ augmented with a row $(x_t)_{t\in\mathcal{T}}$ defined by

$$x_t = \mathbf{1}_{\{t^\bullet = \emptyset\}} - \mathbf{1}_{\{^\bullet t = \emptyset\}}\,,$$

(where $t^\bullet$ and $^\bullet t$ are defined in $\mathcal{N}$). We have $\mathrm{rank}(N') \geq \mathrm{rank}(N)$. On the other hand, observe that $\forall t \in \mathcal{T}, x_t = -\sum_{s\in\mathcal{P}} N_{s,t}$, so $N' = BN$, where $B$ is the $\mathcal{P}\times\mathcal{P}$ identity matrix augmented with the row $(-1,\ldots,-1)$. Hence $\mathrm{rank}(N') = \mathrm{rank}(BN) \leq \mathrm{rank}(N)$. So:

$$\mathrm{rank}(N') = \mathrm{rank}(N)\,. \tag{14}$$

Together (13) and (14) imply that $\mathcal{N}$ and $\mathcal{N}'$ have the same deficiency. Since $\mathcal{N}'$ has deficiency zero (Lemma 3), $\mathcal{N}$ also has deficiency zero. □

By coupling Proposition 5 and Theorem 1, or alternatively Proposition 6 and Theorem 3, we get the result below.

**Corollary 2.** *Consider a Markovian weakly reversible GSM. The NLTE have a strictly positive solution* $(u_p)_p$*. The marking process admits a product form invariant measure given by:* $\forall x \in \mathcal{R}(M_0)$,

$$\pi(x) = \Phi(x)^{-1}\prod_{p\in\mathcal{P}} u_p^{x_p}\,.$$

*In the case of a SM,* $\mathcal{R}(M_0)$ *is finite, the marking process is ergodic, and* $\pi$ *can be normalized to give a product form stationary distribution:* $\forall x \in \mathcal{R}(M_0)$,

$$\widetilde{\pi}(x) = B\Phi(x)^{-1}\prod_{p\in\mathcal{P}} u_p^{x_p}\,,$$

*where* $B = \left(\sum_{x\in\mathcal{R}(M_0)}\Phi(x)^{-1}\prod_{p\in\mathcal{P}} u_p^{x_p}\right)^{-1}$.

Corollary 2 is far from a surprising or new result, as we now show.

## 4.2 Jackson Networks

The product form result for Jackson networks is one of the cornerstones of Markovian queueing theory. It was originally proved by Jackson [15] for open networks and by Gordon & Newell [12] for closed networks.

Consider a Markovian weakly reversible SM with constant rates $(\kappa_t)_{t\in\mathcal{T}}$. It can be transformed into a Jackson network as follows:

- A place $s$ becomes a simple queue, that is a single server Markovian queue with an infinite buffer. The service rate at queue $s$ is $\mu_s = \sum_{t\in s^\bullet}\kappa_t$.
- The routing matrix $P$ of the Jackson network is the stochastic matrix defined as follows: $\forall u, v \in \mathcal{P}$,

$$P_{u,v} = \begin{cases} \mu_u^{-1}\sum_{t:{}^\bullet t=u,t^\bullet=v}\kappa_t & \text{if } \exists t\in\mathcal{T}, u\rightarrow t\rightarrow v \\ 0 & \text{otherwise}\end{cases}\,.$$

- A token in place $s$ becomes a customer in queue $s$.

Consider now a Markovian weakly reversible GSM with constant rates $(\kappa_t)_{t\in\mathcal{T}}$. On top of the above transformations, we do the following:

– A transition $t$ with ${}^\bullet t = \emptyset$ becomes an external Poisson arrival flow of rate $\kappa_t$ in queue $t^\bullet$.

The routing matrix $P$ is now substochastic. Indeed, if the transition $t$ is such that $t^\bullet = \emptyset$, then $\sum_v P_{{}^\bullet t,v} < 1$.

In the SM case, the Jackson network is *closed*, that is without arrivals from the outside and without departures to the outside. In the GSM case with input and output transitions, the Jackson network is *open*.

The transformation from (G)SM to Jackson network is illustrated on Fig. 5.

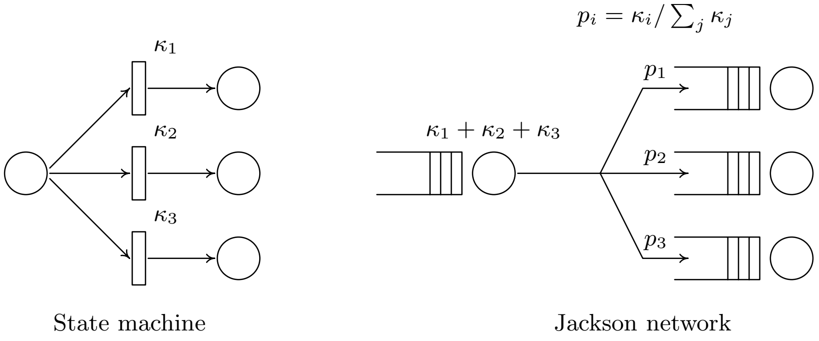


**Fig. 5.** From (generalized) state machine to Jackson network

A Jackson network can be translated into a Markovian (G)SM using the same construction in the reverse direction.

The two models are identical in a strong sense. Precisely, the marking process of the state machine and the queue-length process of the Jackson network have the same infinitesimal generator.

The classical product form results for Jackson networks (Jackson [15] and Gordon & Newell [12]) are exactly the translation via the above transformation of Corollary 2. In the open case, the weak-reversibility implies the classical "without capture" condition of Jackson networks.

The above transformation from GSM to queueing network can also be performed in the case of general rate functions of type (3). Queueing networks with those rate functions are called *Whittle networks* in the literature. The existence of product form invariant measures for these networks is a classical result, see for instance [22] and the references therein.

## 4.3 Free-Choice Petri Nets

We study the family of live and bounded free-choice nets. This is an important class of Petri nets realizing a nice compromise between modelling power and tractability, see the dedicated monography of Desel & Esparza [7]. We show that the only such Petri nets having a product form are, in a sense, the GSM.

**Definition 11 (Free-choice Petri net).** *A* free-choice *Petri net is a non-weighted Petri net* $(\mathcal{P}, \mathcal{T}, \mathcal{F}, M_0)$ *such that: for every two transitions* $t_1$ *and* $t_2$*, either* $^\bullet t_1 = {}^\bullet t_2$ *or* $^\bullet t_1 \cap {}^\bullet t_2 = \emptyset$*.*

Some authors call the above an *extended free-choice* Petri net and have a more restrictive definition for free-choice Petri nets.

In Figure 6, the Petri net on the left is free-choice, while the one on the right is not free-choice.

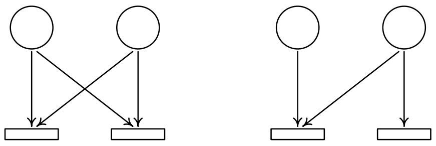


**Fig. 6.** Free-choice (left) and non free-choice (right) Petri nets

**Definition 12 (Cluster).** *The* cluster *of a node* $x \in \mathcal{P} \cup \mathcal{T}$*, denoted by* $[x]$*, is the minimal set of nodes such that:* $(i)$ $x \in [x]$ ; $(ii)$ $\forall t \in \mathcal{T} : \ t \in [x] \implies {}^\bullet t \subset [x]$ ; $(iii)$ $\forall p \in \mathcal{P} : \ p \in [x] \implies p^\bullet \subset [x]$ .

The clusters form a partition of the set of nodes, see [7, Proposition 4.5], and therefore of the places. Moreover, we have the following.

**Lemma 4.** *Consider a weakly reversible free-choice Petri net. The non-empty complexes are disjoint subsets of* $\mathcal{P}$*. The partition of* $\mathcal{P}$ *induced by the non-empty complexes is the same as the partition of* $\mathcal{P}$ *induced by the clusters.*

*Proof.* In a weakly reversible free-choice Petri net, the non-empty complexes are also non-empty input bags, which are disjoint according to the definition of free-choiceness.

It follows from the definition of clusters that every non-empty input bag is entirely contained in a cluster. This cluster is unique because the clusters partition the set of places. Let $I$ be a non-empty complex (which is also a non-empty input bag). Denote by $[I]$ the cluster containing $I$. We have $I^\bullet \subset [I]$, so $I \cup I^\bullet \subset [I]$. Since the Petri net is free-choice, $^\bullet t = I$ for all $t \in I^\bullet$. The set $I \cup I^\bullet$ satisfies the three conditions of the definition of clusters, so we have $[I] \subset I \cup I^\bullet$. We conclude that $[I] = I \cup I^\bullet$ and $I$ is the set of places of the cluster $[I]$.

Conversely, let $[x]$ be a cluster such that $[x] \cap \mathcal{P} \neq \emptyset$. Let $I$ be a non-empty input bag contained in $[x]$. We have, using the above, $[I] = I \cup I^\bullet \subset [x]$. By minimality, $[I] = [x]$ and $[x] \cap \mathcal{P} = I$. □

Under the assumptions of Lemma 4, the non-empty complexes are disjoint. Thus each non-empty complex behaves as if it was a “big place”. Consider the operation which reduces each non-empty complex to a single place. The resulting Petri net is a generalized state machine. And this generalized state machine is weakly reversible because the original free-choice Petri net was weakly reversible. Let us define all this more formally.

**Definition 13 (Reduced generalized state machine).** *Let $\mathcal{N} = (\mathcal{P}, \mathcal{T}, \mathcal{F}, M_0)$ be a weakly reversible free-choice Petri net with set of complexes $\mathcal{C}$. We call the* reduced generalized state machine (RGSM) *of $\mathcal{N}$ the GSM $\mathcal{RN} = (\mathcal{C} \setminus \{\emptyset\}, \mathcal{T}, \widetilde{\mathcal{F}}, \widetilde{M_0})$ where:*

- $\widetilde{\mathcal{F}} = \{({}^\bullet t, t), (t, t^\bullet), t \in \mathcal{T}\}$;
- $\widetilde{M_0}$ *is defined by:* $\forall C \in \mathcal{C} \setminus \{\emptyset\}$, $\widetilde{M_0}(C) = \min_{p \in C} M_0(p)$.

**Lemma 5.** *Let $\mathcal{N}$ be a weakly reversible free-choice Petri net and $\mathcal{RN}$ its RGSM. The marking graph of $\mathcal{RN}$ is isomorphic to the one of $\mathcal{N}$. If $\mathcal{N}$ and $\mathcal{RN}$ are Markovian with the same rates then the two marking processes are "identical", meaning that they have the same infinitesimal generator.*

*Proof.* Consider $f : \mathcal{R}(M_0) \to \mathbb{N}^{\mathcal{C} \setminus \emptyset}$ defined by $f(M)_C = \min_{p \in \mathcal{C}} M(p)$.

It follows from the definition of $\mathcal{RN}$ that if $M \xrightarrow{t} M'$ in $\mathcal{N}$ then $f(M) \xrightarrow{t} f(M')$ in $\mathcal{RN}$, so $f(\mathcal{R}(M_0)) = \mathcal{R}(\widetilde{M_0})$ and the marking graph of $\mathcal{RN}$ is the marking graph of $\mathcal{N}$ up to a renaming of the nodes.

Since the marking graphs are the same, the infinitesimal generators are also identical if the two Petri nets have the same rates. □

Now let us compare the structural characteristics of the original free-choice Petri net $\mathcal{N}$ and of the reduced generalized state machine $\mathcal{RN}$.

**Lemma 6.** *Let $\mathcal{N}$ be a weakly reversible free-choice Petri net. The RGSM $\mathcal{RN}$ is weakly reversible and has the same deficiency as $\mathcal{N}$.*

*Proof.* The weak reversibility of $\mathcal{RN}$ follows directly from the definition of reduced generalized state machines.

The Petri graph of $\mathcal{RN}$ is isomorphic to its reaction graph, Lemma 2. Now by construction, $\mathcal{N}$ and $\mathcal{RN}$ have the same reaction graph. So the number of complexes and the number of connected components of the reaction graph do not change. Call $N$ and $N'$ the incidence matrices of $\mathcal{N}$ and $\mathcal{RN}$ respectively. Let $C$ be an arbitrary complex, let $p, p'$ be two places of $C$. For every transition $t$, we have $N_{p,t} = N_{p',t} = N'_{C,t}$, which implies that $\text{rank}(N) = \text{rank}(N')$. So the two Petri nets have the same deficiency. □

**Corollary 3.** *Weakly reversible free-choice Petri nets have deficiency 0.*

*Proof.* This follows from Prop. 6 and Lemma 6. □

Now all the results for weakly reversible GSM can be applied to weakly reversible free-choice Petri nets. We get the following.

**Theorem 4.** *Let $\mathcal{N}$ be a free-choice Petri net. Then $\mathcal{N}$ is weakly reversible if and only if its NLTE have a strictly positive solution.*

*In this case, the Petri net has deficiency zero. Let $(u_p)_p$ be a strictly positive solution to the NLTE. The marking process has a product form invariant measure $\pi$ given by: $\forall x \in \mathcal{R}(M_0)$,*

$$\pi(x) = \Phi(x)^{-1} \prod_{p \in \mathcal{P}} u_p^{x_p} .$$

*If $\emptyset \notin \mathcal{C}$ then the state space $\mathcal{R}(M_0)$ is finite, the marking process is ergodic and $\pi$ can be normalized to give a product form stationary distribution: $\forall x \in \mathcal{R}(M_0)$,*

$$\widetilde{\pi}(x) = B\Phi(x)^{-1} \prod_{p \in \mathcal{P}} u_p^{x_p} ,$$

*where $B = \left(\sum_{x \in \mathcal{R}(M_0)} \Phi(x)^{-1} \prod_{p \in \mathcal{P}} u_p^{x_p}\right)^{-1}$.*

## References

[1] Ajmone-Marsan, M., Balbo, G., Bobbio, A., Chiola, G., Conte, G., Cumani, A.: The effect of execution policies on the semantics and analysis of stochastic Petri nets. IEEE Trans. on Software Engin. 15(7), 832–846 (1989)

[2] Anderson, D., Craciun, G., Kurtz, T.: Product-form stationary distributions for deficiency zero chemical reaction networks (2008)

[3] Angeli, D., De Leenheer, P., Sontag, E.D.: A Petri net approach to the study of persistence in chemical reaction networks. Mathematical Biosciences 210(2), 598–618 (2007)

[4] Baccelli, F., Cohen, G., Olsder, G.J., Quadrat, J.P.: Synchronization and Linearity. John Wiley & Sons, New York (1992)

[5] Boucherie, R., Sereno, M.: On closed support $T$-invariants and the traffic equations. J. Appl. Probab. 35(2), 473–481 (1998)

[6] Coleman, J.L., Henderson, W., Taylor, P.G.: Product form equilibrium distributions and a convolution algorithm for stochastic Petri nets. Performance Evaluation 26(3), 159–180 (1996)

[7] Desel, J., Esparza, J.: Free Choice Petri Nets. Cambridge Tracts Theoret. Comput. Sci., vol. 40. Cambridge Univ. Press, Cambridge (1995)

[8] Van Dijk, N.M.: Queueing Networks and Product Forms: A Systems Approach. John Wiley & Sons, Chichester (1993)

[9] Donatelli, S., Sereno, M.: On the product form solution for stochastic Petri nets. In: Jensen, K. (ed.) ICATPN 1992. LNCS, vol. 616, pp. 154–172. Springer, Heidelberg (1992)

[10] Feinberg, M.: Lectures on chemical reaction networks. Given at the Math. Research Center, Univ. Wisconsin (1979), http://www.che.eng.ohio-state.edu/~feinberg/LecturesOnReactionNetworks

[11] Florin, G., Natkin, S.: Generalization of queueing network product form solutions to stochastic Petri nets. IEEE Trans. Software Engrg. 17(2), 99–107 (1991)

[12] Gordon, W., Newell, G.: Closed queuing systems with exponential servers. Oper. Res. 15, 254–265 (1967)

[13] Haddad, S., Moreaux, P., Sereno, M., Silva, M.: Product-form and stochastic Petri nets: a structural approach. Performance Evaluation 59(4), 313–336 (2005)

[14] Henderson, W., Lucic, D., Taylor, P.: A net level performance analysis of stochastic Petri nets. J. Austral. Math. Soc. Ser. B 31(2), 176–187 (1989)

[15] Jackson, J.R.: Networks of waiting lines. Oper. Res. 5, 518–521 (1957)

[16] Kelly, F.: Reversibility and Stochastic Networks. Wiley, New-York (1979)

[17] Kurtz, T.G.: The relationship between stochastic and deterministic models for chemical reactions. The Journal of Chemical Physics 57(7), 2976–2978 (1972)

[18] Lazar, A.A., Robertazzi, T.G.: Markovian Petri net protocols with product form solution. Performance Evaluation 12(1), 67–77 (1991)

[19] Murata, T.: Petri nets: Properties, analysis and applications. Proceedings of the IEEE, 541–580 (1989)

[20] Petri, C.A., Reisig, W.: Petri net. Scholarpedia 3(4), 6477 (2008)

[21] Reutenauer, C.: The mathematics of Petri nets. Prentice-Hall, Englewood Cliffs (1990)

[22] Serfozo, R.: Introduction to stochastic networks. Applications of Mathematics, vol. 44. Springer, New York (1999)

# Bisimilarity Minimization in $O(m \log n)$ Time

Antti Valmari

Tampere University of Technology, Department of Software Systems
PO Box 553, FI-33101 Tampere, Finland
Antti.Valmari@tut.fi

**Abstract.** A new algorithm for bisimilarity minimization of labelled directed graphs is presented. Its time consumption is $O(m \log n)$, where $n$ is the number of states and $m$ is the number of transitions. Unlike earlier algorithms, it meets this bound even if the number of different labels of transitions is not fixed. It is based on refining a partition on states with respect to the labelled transitions. A splitter is a pair consisting of a set in the partition and a label. Earlier algorithms consume lots of time in scanning splitters that have no corresponding relevant transitions. The new algorithm avoids this by maintaining the sets of the corresponding transitions. To facilitate this, a refinable partition data structure with amortized constant time operations is introduced. Detailed pseudocode and correctness proof are presented, as well as some measurements.

**Keywords:** Analysis of reachability graphs, verification of systems.

## 1 Introduction

*Bisimilarity* (also known as *strong bisimilarity*) has an important role in the analysis and verification of the behaviours of concurrent systems. For instance, two finite systems satisfy the same CTL or CTL$^*$ formulae if and only if they are bisimilar [2], and two process-algebraic systems are observationally equivalent in the sense of [10] if and only if their so-called saturated versions are bisimilar [8]. Bisimilarity abstracts away precisely the part of information stored by the state of the system that does not have any effect on subsequent behaviour of the system (this only holds in the absence of the notion of "invisible action"). Unlike isomorphism, bisimilarity can unite different states. These properties make it also a useful mathematical tool for dealing with other concepts, like symmetries.

Bisimilarity is an equivalence relation for comparing the vertices of a directed graph whose vertices or edges or both have labels. We will use the words *state* and *transition* instead of "vertex" and "edge" from now on, because the vertices represent states of the concurrent system and edges represent (semantic) transitions between states. Absence of state labels is equivalent to every state having the same label, and similarly with transitions. So we may simplify the discussion by assuming that both states and transitions have labels.

If two states $s_1$ and $s_2$ are bisimilar, then they have the same label, and they can simulate each other's transitions in the following sense. If $s_1$ can make a transition with label $a$ to some state $s'_1$, then there is a state $s'_2$ that is bisimilar with $s'_1$ such that $s_2$ can make an $a$-transition to $s'_2$. In the same fashion, whatever transition $s_2$ can make, $s_1$ can simulate it.

The description of bisimilarity given above cannot be used as a definition, because it is circular: to explain what it means that $s_1$ and $s_2$ are bisimilar, it appeals to the bisimilarity of $s'_1$ and $s'_2$. Many different relations satisfy the description. Therefore, the precise definition uses the auxiliary concept of *bisimulation.* A binary relation between states is a bisimulation if and only if it meets the description of bisimilarity given above. The empty relation and the identity relation are trivially bisimulations. It is not difficult to check that the union of any set of bisimulations over the same graph is a bisimulation. It is the largest bisimulation, and it is an equivalence. It is the bisimilarity relation.

Bisimilarity can be applied to reachability graphs of Petri nets. The label of a semantic transition could be the name of the Petri net transition whose occurrence created the semantic transition, or it could be something more abstract, like a common name of several Petri net transitions. The label of a state could be a collection of Boolean variables that describe some properties of the state, such as there is a token in a certain subset of places. Using the marking as the label of the state as such would make bisimilarity useless, because then each state would be bisimilar only with itself.

Two systems can be compared by taking their disjoint union and checking that their initial states are bisimilar in it. If the systems have several initial states, then each initial state of each system must have a bisimilar initial state in the other system. For each system, there is a unique smallest system that is bisimilar to it. It can be found by removing the states that are not reachable from any initial state, dividing the set of remaining states to equivalence classes according to bisimilarity, and fusing each equivalence class into a single state. The label and output transitions of the fused state are copied from one of its states, where the end state of the copy transition is the fused state that contains the end state of the copied transition. Thanks to the properties of bisimilarity, it does not matter which state in the fused state is used in the copying. The fused state is made an initial state if and only if it contains an original initial state.

In this paper we concentrate on finding the bisimilarity equivalence classes. We present a new algorithm whose asymptotic time consumption is better than that of earlier algorithms. The consumption does not depend on the number of different labels of transitions. This is nice when there are many different labels, like "send$\langle x\rangle$" and "receive$\langle x\rangle$" where $x$ may assume many different values.

In Section 2 we describe the problem in more detail and discuss earlier work. The new algorithm uses three copies of a special refinable partition data structure that is described in Section 3. The new algorithm is presented in Section 4. Detailed proofs of its correctness and performance are deferred to Section 5, because they cannot be followed before having seen the algorithm as a whole. Some measurements that were made with a prototype implementation are shown in Section 6. Section 7 contains the conclusions.

## 2 Background

The bisimilarity minimization problem is the following. A *partition* $\mathcal{S}$ of a set $S$ is a collection of non-empty, mutually disjoint sets $S_1, S_2, \ldots, S_k$ such that $S_1 \cup S_2 \cup \cdots \cup S_k = S$. A *refinement* of $\mathcal{S}$ is any partition $\{Z_1, Z_2, \ldots, Z_h\}$ such that $Z_1 \cup \cdots \cup Z_h = S$ and each $Z_i$ is a subset of some $S_j$. Given is a labelled directed graph $(S, L, \Delta)$, where $\Delta \subseteq S \times L \times S$, together with an initial partition $\mathcal{I}$ of $S$. By $s -a\rightarrow s'$ we mean that $(s, a, s') \in \Delta$. A partition $\mathcal{S}$ is *compatible* with $\Delta$, if and only if for every $S \in \mathcal{S}$, $S' \in \mathcal{S}$, $s_1 \in S$, $s_2 \in S$, $s'_1 \in S'$, and $a \in L$ such that $s_1 -a\rightarrow s'_1$, there is an $s'_2 \in S'$ such that $s_2 -a\rightarrow s'_2$. It has been proven that there is a unique partition that is a refinement of $\mathcal{I}$, compatible with $\Delta$, and contains as few sets as possible. The task is to find that partition.

In this formulation of the problem, the state labels of Section 1 have been replaced by the initial partition. This is not an important modification, because the only thing done with state labels in the definition of bisimilarity is checking whether the labels of two states are the same or not. The initial partition can be constructed quickly enough by sorting the states according to their labels.

We denote the numbers of states, transitions, and labels by $n = |S|$, $m = |\Delta|$, and $\alpha = |L|$. The number of sets in the initial partition is denoted with $k$. To avoid getting into troublesome technicalities with complexity formulae, we assume that $n \leq 2m$. This is not a significant restriction, because to violate it the directed graph must be rather pathological. If every state of a graph has at least one input or output transition, then it meets the assumption.

In some applications of the bisimilarity minimization problem, only those states are relevant that are reachable from an initial state. Furthermore, it may be that a state is irrelevant also if no final state is reachable from it and it is not initial. It is well known that irrelevant states can be removed in $O(m + n)$ time by basic graph traversal algorithms.

The bisimilarity minimization problem can be solved by starting with the initial partition, and splitting sets of the partition as long as necessary. In this context, the sets of the partition are traditionally called *blocks*. If $s_1$ and $s_2$ are in the same block $B$ and $s_1 -a\rightarrow s'_1$, where $s'_1$ is in block $B'$, but there is no $s'_2 \in B'$ such that $s_2 -a\rightarrow s'_2$, then $B$ must be split so that $s_1$ and $s_2$ go into different halves. This splitting may make further splitting necessary. There may be $s''_1$, $s''_2$, $s''$ in some block $B''$, and $b$ such that $s''_1 -b\rightarrow s_1$, $s''_2 -b\rightarrow s_2$, $s'' -b\rightarrow s_1$, $s'' -b\rightarrow s_2$, and they do not have other $b$-labelled output transitions. Then the separation of $s_1$ and $s_2$ into different blocks makes it necessary to separate $s''$, $s''_1$, and $s''_2$ into three different blocks. We will call this *three-way splitting*.

Hopcroft's famous deterministic finite automaton (DFA) minimization algorithm [7] contains an early sub-quadratic algorithm for an important sub-problem of the bisimilarity minimization problem. DFA minimization consists of removing irrelevant states and solving a restricted version of the bisimilarity minimization problem. In this version, $k \leq 2$ (the final states and the other states), and the graph is *deterministic*, that is, for each $s$ and $a$, there is at most one $s'$ such that $s -a\rightarrow s'$. Thus $m \leq \alpha n$, while in general $m \leq \alpha n^2$.

Hopcroft's algorithm runs in $O(\alpha n \log n)$ time (see [5] or [9]). It uses *splitters*. Precise meaning varies in the literature, but let us define a splitter as a block–label pair $(B, a)$. It is used for splitting each block according to whether its states do or do not have an outgoing $a$-labelled transition whose end state is in $B$. In Hopcroft's algorithm, these transitions are traversed backwards, and their start states are moved to tentative new blocks. This is much better than scanning a block and checking which of its states have an $a$-transition to some state in $B$, because the latter approach may involve costly scanning of numerous states that lack such a transition.

Because DFAs are deterministic, three-way splitting is never necessary. As a consequence, if $(a, B)$ has been used for splitting and then $B$ splits to $B_1$ and $B_2$, it is not necessary to use both $(a, B_1)$ and $(a, B_2)$ for further splitting. Hopcroft's algorithm chooses the (in some sense) "smaller" of them. This guarantees that each time when a transition is used for splitting, the size of some set is at most half of its size in the previous time. Therefore, the same transition is used at most a logarithmic number of times. This made it possible to reach $O(\alpha n \log n)$ time complexity instead of $O(\alpha n^2)$.

An $O(m \log n)$ time algorithm for the sub-problem of bisimilarity minimization where $\alpha = 1$ (or, equivalently, transitions have no labels) was presented by Paige and Tarjan [11]. Now the graph needs not be deterministic and three-way splitting is necessary. To facilitate the use of the "half the size" trick, the algorithm uses *compound blocks*. A compound block is a collection of blocks that once constituted together a single block that has been used for splitting. The use of the largest block in the compound block may be avoided in further splitting. A counter-based technique was used to find out whether the start state of the current transition has an output transition also to elsewhere in the current compound block, in addition to the current block.

Generalizing the Paige–Tarjan algorithm to $\alpha \geq 1$ while maintaining its good complexity is not trivial. The algorithm in [4, p. 229] does not meet the challenge. The paper does not give its time complexity, but there is certainly an $\alpha n$ term, because the algorithm scans the set of labels for each block that it uses in a splitter. Furthermore, it relies on the restrictive assumption that there is a global upper bound to the number of output transitions of any state and label (p. 228).

In [3, p. 242], the label of each transition is represented by adding a new state in the middle of the transition and initially partitioning these states according to the labels they represent. Then the Paige–Tarjan algorithm can be used as such. The time complexity is $O(m \log m)$, which is slightly worse than $O(m \log n)$. The approach also consumes more memory by a constant factor than the algorithm presented in Section 4.

When $\alpha$ is not fixed, an $O(m \log n)$ algorithm even for the deterministic case had not been published until 2008 [12]. The problem has been the time spent in scanning empty splitters, that is, splitters whose block does not have incoming transitions with the label. Nothing needs to be done for them, but they are so numerous that simply looking at each of them separately takes too much time. In [12], this is avoided by maintaining the non-empty sets of transitions that correspond to splitters, and using these sets instead of the splitters to organize the work. The sets constitute a partition of the set of transitions that can be maintained similarly to the blocks. Therefore, [12] presents a refinable partition data structure, one instance of which is used for the blocks and another for the transitions.

In this paper we apply the above idea to the Paige–Tarjan algorithm, to design an $O(m \log n)$ algorithm for the bisimilarity minimization problem. To implement three-way splitting of blocks and to mimic the compound blocks, new features are added to the refinable partition data structure. The counter-based technique in [11] is replaced by a third instance of the structure.

## 3 A Refinable Partition Data Structure

In this section, a refinable partition data structure is presented. It is an extension of the structure presented in [12]. It maintains a partition $\{A_1, A_2, \ldots, A_{sets}\}$ of the set $\{1, 2, \ldots, \mathit{items}\}$ for some integer constant *items*. Later in this paper three instances of it will be used, one where the elements are states and $\mathit{items} = n$, and two where the elements are transitions and $\mathit{items} = m$.

The partition is refinable, meaning that it is possible to replace any $A_i$ by two or three sets, provided that they are non-empty and disjoint and their union is $A_i$. This operation is called *splitting*. One part inherits the index $i$ from $A_i$, while the other parts each get a brand new index.

To indicate which elements go into which subset of $A_i$, elements are *1-marked* and perhaps also *2-marked* before splitting $A_i$. There is one splitting operation that divides $A_i$ to its 1-marked states and the remaining states, so that the set of 1-marked states gets a new index, and the remaining states retain the old index $i$. If either subset would be empty, then the operation does not divide $A_i$. There is also another splitting operation that does the same for 2-marked states. Each operation returns the index of the new subset, or zero to indicate that $A_i$ did not split. The reason for having these two splitting operations instead of one three-way splitting operation is that returning the index or zero would be clumsier with the three-way operation. To discuss marking and splitting, let $A_i^1$ and $A_i^2$ denote the sets of 1-marked and 2-marked elements of $A_i$, respectively. Initially all elements of $A_i$ are unmarked, that is, both $A_i^1$ and $A_i^2$ are empty.

One of the three instances of the data structure uses *bunches*. The bunches are a partition of the set $\{A_1, \ldots, A_{sets}\}$. Therefore, a bunch $U_u$ is a set of sets $\{A_{u_1}, A_{u_2}, \ldots, A_{u_g}\}$. A bunch cannot contain just any subset of $\{A_1, \ldots, A_{sets}\}$. Instead, a bunch starts its life containing precisely one set $A_i$. When any set in a bunch is split, the bunch inherits all of its parts. There is also an operation that extracts a set from a non-singleton bunch and makes a new bunch of it.

Furthermore, there are services for scanning a set or a bunch, and for other duties. All services provided by the data structure are listed below.

*Size*$(s)$ Returns the number of elements in the set with index $s$, that is, $|A_s|$.

*Set*$(e)$ Returns the index of the set that element $e$ belongs to, that is, the $s$ such that $e \in A_s$.

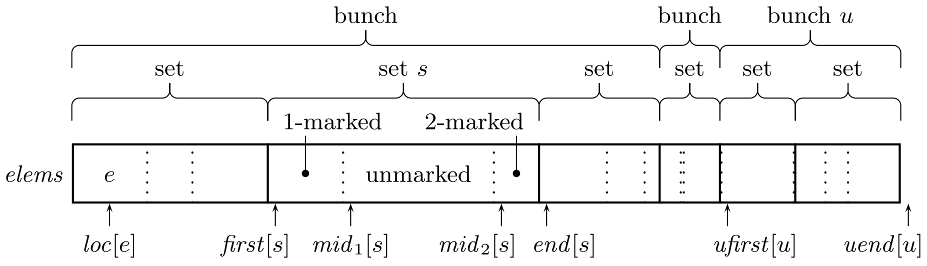


**Fig. 1.** Illustrating the refinable partition data structure

*Mark*1$(e)$ **and** *Mark*2$(e)$ Mark the element $e$ for splitting the set $A_s$ that contains $e$. *Mark*1 adds $e$ to $A_s^1$ and *Mark*2 to $A_s^2$, unless it is already in $A_s^1 \cup A_s^2$.

*Split*1$(s)$ **and** *Split*2$(s)$ If $A_s^1 = \emptyset$ or $A_s = A_s^1 \cup A_s^2$, then *Split*1$(s)$ unmarks all 1-marked elements in $A_s$ and returns zero. Otherwise, it updates $A_s := A_s - A_s^1$, adds a new set $A_z := A_s^1$ to the partition, puts it into the same bunch with $A_s$, and returns $z$. In the end, $A_z^1 = A_z^2 = A_s^1 = \emptyset$, but $A_s^2$ has not changed. *Split*2 works similarly on 2-marked elements.

*No_marks*$(s)$ Returns True if and only if $A_s^1 = A_s^2 = \emptyset$.

*First*$(s)$ **and** *Next*$(e)$ The elements of $A_s$ can be scanned by first executing $e := \mathit{First}(s)$ and then **while** $e \neq 0$ **do** $e := \mathit{Next}(e)$. Each element will be returned exactly once, but the order in which they are returned is unspecified. While scanning a set, *Mark*1, *Mark*2, *Split*1, and *Split*2 must not be executed. These operations are provided to promote data abstraction. Instead of using them, it would be slightly more efficient to scan $A_s$ directly from the arrays that implement the data structure.

*Bunch*$(s)$ Returns the index of the bunch that set $s$ belongs to.

*Bunch_first*$(u)$ **and** *Bunch_next*$(e)$ Let $U_u = \{A_{u_1}, A_{u_2}, \ldots, A_{u_g}\}$ be a bunch. With these operations, the elements of $A_{u_1} \cup A_{u_2} \cup \cdots \cup A_{u_g}$ can be scanned, similarly to how *First*$(s)$ and *Next*$(e)$ scan a set in the partition.

*Has_many*$(u)$ Returns False if and only if bunch $U_u$ consists of precisely one set.

*Extract_set*$(u)$ Let $U_u = \{A_{u_1}, A_{u_2}, \ldots, A_{u_g}\}$ be a bunch. If $g = 1$, then this operation returns zero without changing anything. Otherwise, it selects some $i$, introduces a new bunch $\{A_{u_i}\}$, removes $A_{u_i}$ from $U_u$, and returns $u_i$. The chosen $i$ is such that if $U_u$ has a unique biggest set, then it is not $A_{u_i}$.

*Left_neighbour*$(e)$ **and** *Right_neighbour*$(e)$ If the partition consists of one set, then both of these return zero. Otherwise, at least one of them returns an element that is not currently in the same set as $e$, but was in the same set until the most recent splitting of the set. The other one may return zero or an element. The motivation for these operations is explained in Section 4.

The implementation of the services is illustrated in Figure 1, and shown in Figures 2 and 3. We will soon discuss the implementation of the most complicated operations. The variables *sets* and *bunches* tell the numbers of sets and bunches. The implementation uses them and the following arrays:

```
$\underline{Size(s)}$
return $end[s] - first[s]$

$\underline{Set(e)}$
return $sidx[e]$

$\underline{First(s)}$
return $elems[first[s]]$    /* Certainly exists, because the $A_i$ are non-empty */

$\underline{Next(e)}$
if $loc[e] + 1 \geq end[sidx[e]]$ then return $0$ else return $elems[loc[e] + 1]$

$\underline{Mark1(e)}$
$s := sidx[e];\ \ell := loc[e];\ m := mid_1[s]$
if $m \leq \ell < mid_2[s]$ then
    $mid_1[s] := m + 1$
    $elems[\ell] := elems[m];\ loc[elems[\ell]] := \ell;\ elems[m] := e;\ loc[e] := m$

$\underline{Mark2(e)}$
$s := sidx[e];\ \ell := loc[e];\ m := mid_2[s] - 1$
if $mid_1[s] \leq \ell \leq m$ then
    $mid_2[s] := m$
    $elems[\ell] := elems[m];\ loc[elems[\ell]] := \ell;\ elems[m] := e;\ loc[e] := m$

$\underline{Split1(s)}$
if $mid_1[s] = mid_2[s]$ then $mid_1[s] := first[s]$
if $mid_1[s] = first[s]$ then return $0$
else
    $sets := sets + 1;\ uidx[sets] := uidx[s]$
    $first[sets] := first[s];\ end[sets] := mid_1[s];\ first[s] := mid_1[s]$
    $mid_1[sets] := first[sets];\ mid_2[sets] := end[sets]$
    for $\ell := first[sets]$ to $end[sets] - 1$ do $sidx[elems[\ell]] := sets$
    return $sets$

$\underline{Split2(s)}$
if $mid_1[s] = mid_2[s]$ then $mid_2[s] := end[s]$
if $mid_2[s] = end[s]$ then return $0$
else
    $sets := sets + 1;\ uidx[sets] := uidx[s]$
    $first[sets] := mid_2[s];\ end[sets] := end[s];\ end[s] := mid_2[s]$
    $mid_1[sets] := first[sets];\ mid_2[sets] := end[sets]$
    for $\ell := first[sets]$ to $end[sets] - 1$ do $sidx[elems[\ell]] := sets$
    return $sets$

$\underline{No\_marks(s)}$
if $mid_1[s] = first[s] \wedge mid_2[s] = end[s]$ then return True else return False
```

**Fig. 2.** Main features of the refinable partition data structure

*elems* Contains $1, 2, \ldots, items$ in such an order that elements that belong to the same set are one after another. It is also the case that the sets that belong to the same bunch are one after another in *elems*.

```
$\underline{Bunch(s)}$
return $uidx[s]$

$\underline{Bunch\_first(u)}$
return $elems[ufirst[u]]$

$\underline{Bunch\_next(e)}$
if $loc[e] + 1 \geq uend[uidx[sidx[e]]]$ then return $0$ else return $elems[loc[e] + 1]$

$\underline{Has\_many(u)}$
if $end[sidx[elems[ufirst[u]]]] \neq uend[u]$ then return True else return False

$\underline{Extract\_set(u)}$
$s_1 := sidx[elems[ufirst[u]]]$;  $s_2 := sidx[elems[uend[u] - 1]]$
if $s_1 = s_2$ then return $0$
else
    $bunches := bunches + 1$
    if $Size(s_1) \leq Size(s_2)$ then $ufirst[u] := end[s_1]$ else $uend[u] := first[s_2]$;  $s_1 := s_2$
    $ufirst[bunches] := first[s_1]$;  $uend[bunches] := end[s_1]$;  $uidx[s_1] := bunches$
    return $s_1$

$\underline{Left\_neighbour(e)}$
$\ell := first[sidx[e]]$;  if $\ell > 1$ then return $elems[\ell - 1]$ else return $0$

$\underline{Right\_neighbour(e)}$
$\ell := end[sidx[e]]$;  if $\ell \leq items$ then return $elems[\ell]$ else return $0$
```

**Fig. 3.** Bunch- and neighbour-features of the refinable partition data structure

*first* **and** *end* Indicate the segment in *elems* where the elements of a set are stored. That is, $A_s = \{ elems[f], elems[f+1], \ldots, elems[\ell - 1] \}$, where $f = first[s]$ and $\ell = end[s]$.

$mid_1$ **and** $mid_2$ Let $f$ and $\ell$ be as above, and let $m_1 = mid_1[s]$ and $m_2 = mid_2[s]$. Then $A_s^1 = \{ elems[f], \ldots, elems[m_1 - 1] \}$, the unmarked elements are $elems[m_1], \ldots, elems[m_2 - 1]$, and $A_s^2 = \{ elems[m_2], \ldots, elems[\ell - 1] \}$.

*loc* Tells the location of each element in *elems*, that is, $elems[loc[e]] = e$.

*sidx* The index of the set that $e$ belongs to is $sidx[e]$. That is, $e \in A_{sidx[e]}$.

*uidx* The index of the bunch that $A_s$ belongs to is $uidx[s]$. That is, $A_s \in U_{uidx[s]}$.

*ufirst* **and** *uend* The union of the sets in bunch $U_u$ is $\{ elems[f], elems[f+1], \ldots, elems[\ell - 1] \}$, where $f = ufirst[u]$ and $\ell = uend[u]$.

To avoid marking the same element more than once, $Mark1(e)$ first tests that the element $e$ is in the segment for unmarked elements of the set that contains $e$. If it is, then $Mark1$ swaps the element with the first unmarked element, and moves the borderline between 1-marked and unmarked elements one location forward. The *loc* array is updated according to the new locations of the swapped elements. $Mark2$ works similarly, but with the last unmarked element.

If set $s$ does not contain 1-marked elements, $Split1(s)$ returns zero and terminates on its second line. If it does not contain unmarked elements, the first line unmarks all 1-marked elements, leading to termination on the second line. Otherwise $Split1$ adjusts the number of sets and the set boundaries so that the

1-marked elements become a new set whose all elements are unmarked. The second statement on line 4 makes the new set a member of the same bunch as the original set. The **for**-loop updates the set index of the 1-marked elements to refer to the new set. *Split*2 works similarly with 2-marked elements.

*Has_many*$(u)$ finds the end location of the set that the first element of bunch $u$ belongs to, and tests if it is different from the end location of the bunch.

*Extract_set*$(u)$ first finds the indices of the first and last set in bunch $u$. If they are the same, the bunch consists of only one set, and *Extract_set* returns zero. Otherwise *Extract_set* chooses the smaller of the two sets, removes it from the bunch, and makes a new bunch of it.

*Left_neighbour*$(e)$ returns the element that is immediately before the set that contains $e$ in *elems*, or zero, if the set is the first set in *elems*. *Right_neighbour* works similarly at the opposite end of the set.

The time consumption of initializing the data structure with the partition that consists of one set is linear in *items*. The operations obviously run in constant time, except *Split*1 and *Split*2. Fortunately amortized analysis reveals that they, too, can be treated as constant time in the analysis of the algorithm as a whole. Their running times are proportional to the number of *Mark*1- and *Mark*2-operations executed after the previous splitting. Thus the total cost does not change, even if the split operations are only charged constant cost.

## 4 The Algorithm

In this section the new bisimilarity minimization algorithm is described. Its operation and asymptotic time consumption are discussed at an informal level. Detailed correctness and performance proofs are deferred to Section 5.

It is assumed that states and labels are represented with numbers. That is, $S = \{1, 2, \ldots, n\}$ and $L = \{1, 2, \ldots, \alpha\}$. We have $\Delta \subseteq S \times L \times S$ and $m = |\Delta|$, that is, there are $m$ transitions. Let $\mathcal{I} = \{S_1, S_2, \ldots, S_k\}$ denote the initial partition of $S$. The input to the algorithm consists of $n$, $\alpha$, $\Delta$, and $S_2, \ldots, S_k$. The set $S_1$ need not be given, because it is $S - (S_2 \cup \cdots \cup S_k)$.

Let $\Delta_{a,B} = \Delta \cap (S \times \{a\} \times B)$ and $\Delta_{s,a,B} = \Delta \cap (\{s\} \times \{a\} \times B)$. That is, $\Delta_{a,B}$ is the set of those transitions whose label is $a$ and whose end state is in $B$. Adding the requirement that the start state must be $s$ converts $\Delta_{a,B}$ to $\Delta_{s,a,B}$.

It is assumed that transitions are represented via three arrays *tail*, *label*, and *head*. Each transition $(s, a, s')$ has an index $t$ in the range $1, \ldots, m$ such that *tail*$[t] = s$, *label*$[t] = a$, and *head*$[t] = s'$. It is also assumed that the indices of the transitions that share the same head state $s$ are available via *In_transitions*$[s]$ in some unspecified order. It may be implemented similarly to *elems*, *first*, and *end*, and initialized in $\Theta(m + n)$ time with counting sort. For convenience, confusing the transitions with their indices will be allowed in formulae, like in *In_transitions*$[s] = \{\, (s_1, a, s_2) \in \Delta \mid s_2 = s \,\}$.

For keeping track of work to be done, the algorithm uses four *worksets*. In the prototype implementation, stacks were used as the worksets. However, they need not be stacks. It suffices that they provide three constant time operations:

*Add*$(e)$ that adds the element $e$ to the workset without checking whether it already is there, *Remove* that removes any element of the workset and returns the removed element, and *Empty* that returns True if and only if the workset is empty. The *capacity* of a workset is the maximum number of elements it can store.

The algorithm uses the following data structures:

*Blocks*. This is a refinable partition data structure on $\{1, \ldots, n\}$. Its sets are the blocks. The index of the set in *Blocks* is used as the index of the block also elsewhere in the algorithm.

*Splitters*. This is a refinable partition data structure on $\{1, \ldots, m\}$. Each set in it consists of the indices of the $a$-labelled input transitions of some block $B$, for some label $a$. That is, $\mathit{Splitters} = \{ \Delta_{a,B} \mid a \in L \wedge B \in \mathit{Blocks} \wedge \Delta_{a,B} \neq \emptyset \}$. That this property remains valid is not obvious from the code, so it will be proven later as Lemma 1 (1). The bunch feature of *Splitters* will be used. When saying that a transition is in a bunch it is meant that the bunch contains a splitter that contains the transition.

*Outsets*. This, too, is a refinable partition data structure on $\{1, \ldots, m\}$, but it stores a finer partition than *Splitters*. Transitions that are in the same set of *Outsets* also share their start state. That is, $\mathit{Outsets} = \{ \Delta_{s,a,B} \mid s \in S \wedge a \in L \wedge B \in \mathit{Blocks} \wedge \Delta_{s,a,B} \neq \emptyset \}$. This will be proven as Lemma 1 (2).

*Unready_Bunches*. This is an initially empty workset of capacity $\lfloor m/2 \rfloor$. It contains the indices of the bunches of *Splitters* that consist of two or more splitters. This will be proven as Lemma 1 (3).

*Touched_Blocks*. This is an initially empty workset of capacity $n$. It contains the indices of the blocks that have been affected when using a splitter, but have not yet been split. In other words, precisely those blocks contain marked states.

*Touched_Splitters* **and** *Touched_Outsets*. These are initially empty worksets of capacity $m$. They contain the indices of the sets in *Splitters* and *Outsets*, respectively, that must be updated, because the block where their transitions end has been split. In other words, precisely those splitters or outsets contain marked transitions.

Before discussing the main algorithm, it is useful to introduce the *Update* subroutine that is shown in Figure 4. Each time a block has been split, it is necessary to split sets in *Splitters* and *Outsets* accordingly, to keep them consistent with *Blocks* in the above-mentioned sense. The updating of *Splitters* may make it necessary to update *Unready_Bunches*, to maintain its above-mentioned relation to *Splitters*. These duties are taken care of by *Update*.

When *Update* is called, $b$ and $b'$ contain the indices of the halves of the block that has just been split. Both halves must be non-empty.

To understand *Update*, let us first discuss what happens to a single set in *Outsets* and temporarily ignore the rest. *Update* chooses one of the halves of the block that has just been split, and marks those transitions of the outset that end in the chosen half. (The implementation of this will be discussed soon.) If there

```
    Update(b, b')
 1  if $Blocks.Size(b) \leq Blocks.Size(b')$ then $s := Blocks.First(b)$
 2  else $s := Blocks.First(b')$
 3  while $s \neq 0$ do
 4      for $t \in In\_transitions[s]$ do
 5          $p := Splitters.Set(t)$;  $o := Outsets.Set(t)$
 6          if $Splitters.No\_marks(p)$ then $Touched\_Splitters.Add(p)$
 7          if $Outsets.No\_marks(o)$ then $Touched\_Outsets.Add(o)$
 8          $Splitters.Mark1(t)$;  $Outsets.Mark1(t)$
 9      $s := Blocks.Next(s)$
10  while $\neg Touched\_Splitters.Empty$ do
11      $p := Touched\_Splitters.Remove$
12      $u := Splitters.Bunch(p)$;  if $Has\_many(u)$ then $u := 0$
13      $p' := Splitters.Split1(p)$;  if $u \neq 0 \wedge p' \neq 0$ then $Unready\_Bunches.Add(u)$
14  while $\neg Touched\_Outsets.Empty$ do
15      $o := Touched\_Outsets.Remove$;  $o' := Outsets.Split1(o)$
```

**Fig. 4.** The *Update* subroutine

are no such transitions, then nothing happens to the outset. Otherwise, after the marking phase, *Update* calls the split operation on the outset. If all transitions of the outset were marked, then the split operation just unmarks them. Otherwise, it divides the outset to two halves, those transitions that were marked and those that were not, and unmarks all of its transitions. As a consequence, if all transitions of the outset end in the same half-block, then *Update* does not modify the outset; otherwise, it divides it to two outsets according to in which half-block its transitions end.

To obtain good performance, *Update* updates all outsets in one batch. On lines 1 and 2, *Update* finds the first state in the smaller half-block. As will be discussed later, the performance of the algorithm depends on choosing the smaller half. On lines 3 to 9, $s$ scans through the states in the chosen half, and $t$ scans the input transitions of $s$ on lines 4 to 8. Transitions in the outsets are marked on line 8. The indices of the outsets whose transitions are marked are collected into *Touched_Outsets*. The test on line 7 ensures that the index is added to *Touched_Outsets* only once. The test works, because it is executed before the transition is marked. Lines 14 and 15 pick each outset that contains marked transitions one at a time, and splits it.

The processing of *Splitters* is otherwise similar to *Outsets*, but it contains an additional step on lines 12 and 13. Line 13 may increase the number of splitters in the bunch that contains $p$. If it is increased from one to two, then the bunch index must be added to *Unready_Bunches*, to maintain the property that it contains the indices of precisely those bunches that consist of more than one splitter. If line 13 does not actually split $p$, the test $p' \neq 0$ fails and *Unready_Bunches* is not changed. If the bunch is already in *Unready_Bunches*, then $u$ becomes zero on line 12 and *Unready_Bunches* is not changed. The code is a bit complicated, because *Has_many* must be executed before $p$ is split.

```
    $\underline{\mathit{Main\_part}}$
16  initialize $\mathit{Blocks}$ to $\{S\}$ and $\mathit{Splitters}$ to $\{\, \Delta_{a,S} \mid a \in L \wedge \Delta_{a,S} \neq \emptyset \,\}$
17  make every set of $\mathit{Splitters}$ a singleton bunch
18  initialize $\mathit{Outsets}$ to $\{\, \Delta_{s,a,S} \mid s \in S \wedge a \in L \wedge \Delta_{s,a,S} \neq \emptyset \,\}$
19  for $i := 2$ to $k$ do
20      for $s \in S_i$ do $\mathit{Blocks.Mark}1(s)$
21      $b := \mathit{Blocks.Split}1(1)$;  $\mathit{Update}(1, b)$
22  for $u := 1$ to $\mathit{Splitters.bunches}$ do
23      $t := \mathit{Splitters.Bunch\_first}(u)$
24      while $t \neq 0$ do
25          $s := \mathit{tail}[t]$;  $b := \mathit{Blocks.Set}(s)$
26          if $\mathit{Blocks.No\_marks}(b)$ then $\mathit{Touched\_Blocks.Add}(b)$
27          $\mathit{Blocks.Mark}1(s)$;  $t := \mathit{Splitters.Bunch\_next}(t)$
28      while $\neg\, \mathit{Touched\_Blocks.Empty}$ do
29          $b := \mathit{Touched\_Blocks.Remove}$
30          $b' := \mathit{Blocks.Split}1(b)$;  if $b' \neq 0$ then $\mathit{Update}(b, b')$
31  while $\neg\, \mathit{Unready\_Bunches.Empty}$ do
32      $u := \mathit{Unready\_Bunches.Remove}$;  $p := \mathit{Splitters.Extract\_set}(u)$
33      if $\mathit{Splitters.Has\_many}(u)$ then $\mathit{Unready\_Bunches.Add}(u)$
34      $t := \mathit{Splitters.First}(p)$
35      while $t \neq 0$ do
36          if $t = \mathit{Outsets.First}(\, \mathit{Outsets.Set}(t) \,)$ then
37              $s := \mathit{tail}[t]$;  $b := \mathit{Blocks.Set}(s)$
38              if $\mathit{Blocks.No\_marks}(b)$ then $\mathit{Touched\_Blocks.Add}(b)$
39              $t_1 := \mathit{Outsets.Left\_neighbour}(t)$;  $t_2 := \mathit{Outsets.Right\_neighbour}(t)$
40              if    $t_1 > 0 \wedge \mathit{tail}[t_1] = s \wedge \mathit{Splitters.Bunch}(\mathit{Splitters.Set}(t_1)) = u$
41                  $\vee\ t_2 > 0 \wedge \mathit{tail}[t_2] = s \wedge \mathit{Splitters.Bunch}(\mathit{Splitters.Set}(t_2)) = u$
42              then $\mathit{Blocks.Mark}1(s)$ else $\mathit{Blocks.Mark}2(s)$
43          $t := \mathit{Splitters.Next}(t)$
44      while $\neg\, \mathit{Touched\_Blocks.Empty}$ do
45          $b := \mathit{Touched\_Blocks.Remove}$
46          $b' := \mathit{Blocks.Split}1(b)$;  if $b' \neq 0$ then $\mathit{Update}(b, b')$
47          $b' := \mathit{Blocks.Split}2(b)$;  if $b' \neq 0$ then $\mathit{Update}(b, b')$
```

**Fig. 5.** Main part of the bisimilarity minimization algorithm

*Touched_Splitters* and *Touched_Outsets* are always empty when *Update* is started, because they are initially empty, not used elsewhere, and *Update* leaves them empty. The number of *Remove*-operations on them is thus the same as the number of *Add*-operations. The cost of each split-operation is linear in the number of the corresponding mark-operations. Other individual operations in *Update* take constant time. Therefore, its execution time is dominated by lines 3 to 9. It is thus linear in the sum of the number of states in the scanned half-block and the total number of their input transitions.

The main algorithm is shown in Figure 5. It starts by initializing *Blocks*, *Splitters*, and *Outsets* on lines 16 to 18 according to the situation where there is only one block, and each splitter constitutes alone a bunch. *Unready_Bunches* is initially empty. This is consistent with the effect of line 17.

The time consumption of the initialization is not otherwise a problem, but the initialization of *Splitters* and *Outsets* requires putting the transitions in a suitable order in *Splitters.elems* and *Outsets.elems*. Sorting the transitions with heapsort would take $\Theta(m \log m)$ time in the worst case, which is more than is allowed. Sorting them with counting sort using the label as the key would take $\Theta(m + \alpha)$ time and memory. That is too much, when $\alpha = \omega(m \log n)$. There is a trick with which the transitions can be classified according to their labels in $\Theta(m)$ time and $\Theta(m + \alpha)$ memory. It is based on [1, Exercise 2.12] and was applied to the present purpose in [12]. The resulting order is suitable for *Splitters*. Counting sorting the transitions with the start state as the key before using the trick makes the resulting order suitable also for *Outsets*, and only takes $\Theta(n + m) = \Theta(m)$ extra time and memory.

Lines 19 to 21 split the original block according to the initial partition given in the input, and update the other data structures accordingly. In the beginning of line 21, block 1 is $S_1 \cup S_i \cup S_{i+1} \cup \cdots \cup S_k$. The split operation extracts $S_i$ and makes it block $b$. Because the $S_i$ constitute a partition, none of them is empty. Therefore, the half-blocks 1 and $b$ are both non-empty when *Update* is called.

Now blocks have to be split until the partition is compatible. As has been discussed above, the sets in *Splitters* correspond precisely to the non-empty $\Delta_{a,B}$, where $a$ is a label and $B$ is a block. Therefore, the splitting obligation introduced by $a$ and $B$ can be met by marking the start states of the transitions in the corresponding splitter and then splitting the blocks that contain marked states.

To keep track of pending splitting obligations, the algorithm uses the bunches of *Splitters* together with *Unready_Bunches*. The bunches are used largely in the same way as compound blocks were used in [11].

The algorithm first establishes and then maintains the property that *if two states are in the same block, then, for each bunch in Splitters, either none or both of them have an output transition in the bunch.* This invariant is crucial for performance, as it allows to skip a largest splitter in the bunch when splitting.

The test on line 31 implies that when the algorithm terminates, each bunch consists of a single splitter. Therefore, upon termination, if two states are in the same block, then, for each splitter, either none or both of them have an output transition in the splitter. This means that all splitting obligations have been satisfied. In other words, for all states $s_1$, $s_2$, and $s'_1$, labels $a$, and blocks $B$ and $B'$, if $s_1 -a\rightarrow s'_1$ and $s_1 \in B$ and $s_2 \in B$ and $s'_1 \in B'$, then $s_1$ has an output transition in the splitter that corresponds to $\Delta_{a,B'}$, so also $s_2$ has, implying that there is some $s'_2 \in B'$ such that $s_2 -a\rightarrow s'_2$. That is, the partition is compatible.

The **for**-loop on lines 22 to 30 establishes the above-mentioned property. For each bunch, it splits the blocks so that the property starts to hold. Each cycle around the **for**-loop processes *Blocks* similarly to the processing of *Outsets* in *Update*. On lines 23 to 27, the start states of the transitions in the bunch are marked, and the indices of the blocks that contain marked states are collected into *Touched_Blocks*. *Touched_Blocks* is discharged on lines 28 to 30 by splitting the touched blocks. For each splitting, if both halves are non-empty, then *Update* is called, to keep the other data structures consistent with *Blocks*.

Lines 22 to 30 actually separate the states according to the labels of their output transitions. This is because bunches have not yet been divided after their initialization, although splitters in them may have. Each bunch thus contains precisely the $a$-labelled transitions for some label $a$. Therefore, lines 22 to 30 separate two states if and only if, for some label $a$, one of them has and the other does not have an $a$-labelled output transition.

Lines 31 to 47 discharge *Unready_Bunches* while maintaining the above-mentioned property. On line 32, one bunch is chosen for processing, and a splitter is extracted from it. If the remaining part of the bunch still contains more than one splitter, line 33 puts it back to *Unready_Bunches*, in accordance with the main property of *Unready_Bunches*.

To re-establish the above-mentioned property, it may be necessary to separate two states because only one of them has an output transition in the remaining bunch $u$, or because only one of them has an output transition in the extracted splitter $p$ (which is now a bunch on its own). This means that states of each block have to be separated into three groups: those that have output transitions only in $p$, only in $u$, or in both. Lines 34 to 47 do that according to a pattern that we have seen twice before. We now discuss what is new on those lines.

The first novelty is the test on line 36. All transitions in the same set of *Outsets* have the same start state. They also have the same left neighbour and the same right neighbour in the sense of line 39. So they all have the same effect on line 42. Therefore, it suffices to investigate only one of them on lines 37 to 42. This is what the test on line 36 achieves. The test is an optimization that affects neither correctness nor asymptotic time consumption, but improves practical time consumption.

The second novelty is the splitting of blocks into three parts, and the test on lines 40 and 41 that controls the splitting. We claim that a state $s$ that has an output transition in $p$ or $u$ is 1-marked, is 2-marked, or remains unmarked, if it has an output transition in both $p$ and $u$, only in $p$, or only in $u$, respectively. Because $t$ scans $p$, $s$ is marked in some way if and only if it has an output transition in $p$. If $s$ is 1-marked, then $t_1$ or $t_2$ is its output transition in $u$. It remains to be shown that if a marked $s$ has an output transition in $u$, then $t_1$ or $t_2$ is such a transition. For each $s$ and $a$, the order of the $\Delta_{s,a,B}$ in *Outsets.elems* is the same as the order of the $\Delta_{a,B}$ in *Splitters.elems*, because the algorithm updates *Splitters* and *Outsets* in a similar way. As a consequence, the output transitions of $s$ that are in $p$ or $u$ are contiguously in *Outsets.elems*. Thus *Left_neighbour* or *Right_neighbour* or both find an output transition in $u$, if any exists.

Thanks to three issues, the algorithm runs in $O(m \log n)$ time. The first is Hopcroft's trick [7]: because *Extract_set* tries two splitters and extracts the smaller of them, each time when a transition is used for splitting blocks, it belongs to a splitter whose size is at most half the size in the previous time. All transitions in a splitter have the same label, so there can be at most $n^2$ of them.

Thus each transition can be used at most $\log_2 n^2 = 2 \log_2 n$ times for splitting. Bunches were needed to make it legal to skip the largest splitter.

The second is Knuutila's trick [9]: because *Update* chooses the smaller half-block, each time when a state is used for updating splitters and outsets, it belongs to a block whose size is at most half the size in the previous time. Thus each state can be used at most $\log_2 n$ times for updating.

The third issue is from [12]. It is the organisation of the work in such a way that the set of labels is never scanned. Instead, subsets of transitions are scanned so that if there are no transitions for some label, then no work is done for that label. Failure to obey this principle would easily introduce an $\Omega(n\alpha)$ term to time consumption.

## 5 Detailed Proofs

The previous section explained the principle of the algorithm. In this section, detailed proofs of its correctness and performance are presented.

As is obvious from earlier discussion, it is important that *Splitters* and *Outsets* are consistent with *Blocks*, and *Unready_Bunches* is consistent with *Splitters*. The duty of the *Update* subroutine is to re-establish consistency each time *Blocks* has changed. Let us state this precisely, and check that also the main algorithm maintains consistency where necessary.

**Lemma 1.** *The following hold everywhere after line 18, except (1) and (2) on lines 21, 30, 46, and 47 and within Update, and (3) on lines 32, 33, and 13.*

*(1) For any two transitions, they are in the same set in Splitters if and only if they have the same label and they end in the same set in Blocks.*
*(2) For any two transitions, they are in the same set in Outsets if and only if they have the same start state and the same label, and they end in the same set in Blocks. This is equivalent to that they have the same start state and belong to the same set in Splitters.*
*(3) Unready_Bunches contains the indices of precisely those bunches of splitters that contain two or more splitters.*

*Proof.* The main algorithm starts by initializing *Blocks*, *Splitters*, and *Outsets* on lines 16 to 18 according to the situation where there is only one block. This makes (1) and (2) hold. It also makes (3) hold, because *Unready_Bunches* is initially empty and line 17 makes each bunch of splitters to consist of a single splitter. From then on, each time a block has been split (lines 21, 30, 46, and 47) resulting in two non-empty sub-blocks, *Update* is called with the two halves as the parameters. It splits *Splitters* and *Outsets* so that (1) and (2) are re-established.

The number of splitters in a bunch grows only on line 13. It has already been discussed. The number of splitters in a bunch decreases only on line 32. It removes the bunch from *Unready_Bunches* and removes a splitter from the bunch. Line 33 checks whether the bunch should have remained in *Unready_Bunches*, and puts it back there if necessary. The removed splitter becomes a bunch of its own. It is a singleton bunch, so it is not added to *Unready_Bunches*. □

The test on lines 40 to 42 is tricky enough to deserve a lemma of its own.

**Lemma 2.** *If line 42 1-marks state $s$, then $s$ has an outgoing transition in bunch $u$. If line 42 2-marks $s$, then $s$ does not have an outgoing transition in $u$.*

*Proof.* If line 42 1-marks $s$, then $t_1$ or $t_2$ is clearly such a transition.

Assume now that such a transition $t'$ exists. The state $s$ has been found on line 37 via some $t$. Immediately before the extract operation on line 32 both $t$ and $t'$ were in $u$. So, by lines 16 and 17, they have the same label, say $a$. Let $B$ and $B'$ be the blocks where $t$ and $t'$ end. So $B \neq B'$, $t \in \Delta_{s,a,B}$, and $t' \in \Delta_{s,a,B'}$.

Let $X \sqsubseteq Y$ denote that the elements of set $X$ occupy at most as big indices in the *elems* array in question as the elements of set $Y$.

Assume first that $\Delta_{s,a,B} \sqsubseteq \Delta_{s,a,B'}$. Thanks to line 18, if $\Delta_{s,a,B} \sqsubseteq \Delta_{s'',a'',B''} \sqsubseteq \Delta_{s,a,B'}$, then $s'' = s$ and $a'' = a$. Therefore, $t_2 \neq 0$ and $t_2$ is in some $\Delta_{s,a,B''}$, where $B'' \neq B$. The operation of *Update* implies that $\Delta_{a,B} \sqsubseteq \Delta_{a,B''} \sqsubseteq \Delta_{a,B'}$ in *Splitters.elems*. Because *Extract_set* extracted $\Delta_{a,B}$ from one end of $u$ while $\Delta_{a,B'}$ stayed in $u$, also $\Delta_{a,B''}$ was and stayed in $u$. As a consequence, $t_2$ is in $u$ and passes the test on line 41. So $s$ is 1-marked.

The case $\Delta_{s,a,B'} \sqsubseteq \Delta_{s,a,B}$ is symmetric with $t_1$ replacing $t_2$. □

The next lemma says that the algorithm does not do any splitting that it should not.

**Lemma 3.** *If the algorithm puts two states in different blocks, then those states belong to different blocks in each partition that is a refinement of $\mathcal{I}$ and compatible with $\Delta$.*

*Proof.* If two states go into different blocks on line 21, then they are in different blocks in $\mathcal{I}$.

When lines 22 to 30 are executed, the bunches of splitters still contain the same transitions as originally, although they may have been divided into many splitters. Thus each execution of lines 23 to 27 scans some $\Delta_{a,S}$. Therefore, if line 30 separates two states, then one of them has and the other does not have an outgoing $a$-transition.

The case remains where line 46 or 47 puts the states into different blocks. Assume that $s_1$ is moved to a new block on line 46 and $s_2$ on line 47, while $s_0$ stays in the original block. By Lemma 1 (1), there is some $B \in$ *Blocks* and $a \in L$ such that $s_1$ and $s_2$ have but $s_0$ does not have an $a$-transition to $B$. ($B$ is the block and $a$ is the label that correspond to the splitter that is scanned on lines 34 to 43.) It is thus necessary to separate $s_0$ from $s_1$ and $s_2$ to obtain a compatible partition.

By Lemma 2, $s_1$ has and $s_2$ does not have an output transition that belongs to $u$. Let $a$ be the label and $B'$ the end block of that transition of $s_1$. Then $s_1$ has and $s_2$ does not have an $a$-transition that ends in $B'$. So it is correct to put $s_1$ and $s_2$ into different blocks. □

The next lemma says that the algorithm does all the splitting that it should.

**Lemma 4.** *When the algorithm terminates, Blocks is a refinement of $\mathcal{I}$ and compatible with $\Delta$.*

*Proof.* Lines 19 to 21 ensure that *Blocks* will be a refinement of $\mathcal{I}$. The rest of the proof is based on the following Gries-style [6] invariant.

> On line 31, for every states $s_1$ and $s_2$ that are in the same block, transition $t_1$ that starts at $s_1$, and bunch of splitters $u$ that contains $t_1$, there is a transition $t_2$ that starts at $s_2$ and is in $u$.

Lines 22 to 30 make the invariant hold by separating $s_1$ to a different block from $s_2$, if $t_2$ does not exist.

The constituents of the invariant may change only when a block is split or the set of transitions in a bunch is modified. Splitting a block is not a threat to the invariant (merging blocks would be, but the algorithm does not do that). Only *Extract_set* modifies the set of transitions in a bunch, and the only place where it is executed is line 32. There a bunch is divided to a new singleton bunch that consists of the splitter $p$, and $u$ that contains the rest of the original bunch.

The purpose of lines 34 to 47 is to split blocks into up to three parts according to the existence of an output transition in $p$ but not in $u$, in $u$ but not in $p$, and in both. If $s$ does not have an output transition in $p$, then it stays in its block. In the remaining two cases, by Lemma 2 it is put in a different block on line 46 or 47 depending on which case holds. So the invariant is re-established.

Lemma 1 (3) implies that when the algorithm terminates, every bunch consists of precisely one splitter. Then the invariant actually says that for every states $s_1$ and $s_2$ that are in the same block, transition $t_1$ that starts at $s_1$, and splitter $p$ that contains $t_1$, there is a transition $t_2$ that starts at $s_2$ and is in $p$. By Lemma 1 (1), this is equivalent to that *Blocks* is compatible with $\Delta$. □

The efficiency of the algorithm is stated in the next lemma.

**Lemma 5.** *The algorithm runs in $O(m \log n)$ time and $O(m + \alpha)$ memory (assuming that $n \leq 2m$).*

*Proof.* All data structures consume $O(n)$, $O(m)$, or $O(\alpha)$ memory. The running time of lines 16 to 18 was discussed in Section 4. Excluding the time spent in the loops within *Update*, lines 19 to 21 are obviously $O(n)$ and lines 22 to 30 $\Theta(m)$.

Because *Extract_set* avoids choosing the largest set, each splitter that is used as the $p$ on lines 32 to 47 inherits at most half of the transitions of the bunch from which it is extracted. The splitter becomes a new bunch. As a consequence, when any transition is used anew for splitting a block, it belongs to a splitter whose size is at most half the size in the previous time. Initially a splitter contains at most $n^2$ transitions. So the same transition can be used at most $2 \log_2 n$ times. Therefore, lines 36 to 43 and 45 to 47 are executed at most $2m \log_2 n$ times. Because splitters are not empty, lines 32 to 34 are executed at most the same number of times as line 36, and lines 35 and 44 at most twice that many times. Line 31 is executed once more than line 32.

By now the act of calling *Update* has been taken into account in the analysis, but the execution of *Update* has not. Lines 1 and 2 determine whether $b$ or $b'$ is scanned. If $b'$ is scanned, then each scanned state was marked on line 20, 27, or 42. Otherwise other states are scanned, but their number is at most the same. So the $n$, $m$, or $2m \log_2 n$ bound applies to line 9. Whenever lines 5 to 8 are executed anew for some $t$, the test on lines 1 and 2 guarantees that $head[t]$ belongs to a block whose size is at most half of the size in the previous time. This implies an $m \log_2 n$ upper bound. Lines 11 to 13 and 15 are executed at most as often as line 5. So every line of the algorithm meets the $O(m \log n)$ bound. □

(It is indeed the case that the reasons why lines 8 and 9 meet the time bound are different. Each of them may execute more often than the other, as *In_transitions*$[s]$ may be empty for many $s$. A similar issue was discussed in [9].)

**Corollary 1.** *The algorithm solves the bisimilarity minimization problem in* $O(m \log n)$ *time and* $O(m + \alpha)$ *memory (assuming that* $n \leq 2m$*).*

## 6 Experience with a Prototype Implementation

For the purpose of testing the new algorithm and getting an idea of its performance, the present author implemented it in C++. No reference implementation was available for the general problem, but, thanks to [12], two comparable programs were available for the special case of DFA minimization. Therefore, a pre-processing stage was added that removes unreachable states, and non-initial states from which no state in $S_1$ is reachable. This made the new program applicable as such both to DFA minimization and to bisimilarity minimization with at most two initial blocks.

The author tested the correctness of his implementation first by giving an extensive set of randomly generated DFAs to the new and the reference program, and checking that the outputs were isomorphic. Then he tested the new program with more than 300 randomly generated nondeterministic graphs of various sizes and densities. Unfortunately, in the nondeterministic case there is no reference program, no straightforward way to check isomorphism, nor other simple way of fully checking the output. Therefore, each nondeterministic graph was given to the program in four different versions, and it was checked that the four outputs had the same number of states and the same number of transitions. Two of the versions were obtained by randomly permuting the numbering of states in the original version, and the first output was used as the fourth input.

Timing measurements were conducted by Petri Lehtinen and executed on a PC with Linux and 1 gigabyte of memory. A sample of results with randomly generated nondeterministic graphs is shown in Table 1. Each entry shows the fastest and slowest of three measurements, made with $|S_1| = n/2 + d$ and $|S_2| = n/2 - d$, where $d \in \{-1, 0, 1\}$. The times are shown in seconds. The clock was started when the input file had been read, and stopped when the program was ready to start writing the output file. No attempt was made to optimize the implementation to the extreme. In particular, all instances of the refinable

**Table 1.** Running time with nondeterministic input containing two initial blocks

| $n$ | $\alpha$ | $m = 20\,000$ | $m = 50\,000$ | $m = 100\,000$ | $m = 200\,000$ | $m = 500\,000$ | $1\,000\,000$ |
|---|---|---|---|---|---|---|---|
| 1 000 | 10 | 0.017 0.017 | 0.043 0.044 | 0.119 0.123 | 0.092 0.365 | 0.257 0.260 | 0.539 0.547 |
| 1 000 | 100 | 0.021 0.022 | 0.084 0.088 | 0.251 0.255 | 0.501 0.505 | 1.016 1.025 | 2.330 2.402 |
| 10 000 | 10 | 0.005 0.005 | 0.027 0.027 | 0.074 0.079 | 0.505 0.512 | 1.404 1.456 | 2.926 2.983 |

partition data structure contained also the arrays and functionality (like the bunches) that the particular instance does not need.

Because it is difficult to generate a precise number of transitions according to the uniform distribution, sometimes the generated number was slightly smaller than the desired number. Running time depends also on the size of the result: the smaller it is, the less splitting of blocks. It is also very difficult to get full control of all other activity that is going on in a modern computer. As a consequence, the measurements contain some noise. The results should be considered as typical, not as the absolute truth.

When $m$ is big enough compared to $n\alpha$, each state is likely to have an output transition with every label to both a state in $S_1$ and in $S_2$, causing the graph to minimize to 2 states and $4\alpha$ transitions, while with a smaller $m$ the graph does not reduce much. This explains the anomaly with $n = 1\,000$, $\alpha = 10$, and $m = 200\,000$. The row $n = 10\,000$ is subject to another, smoother phenomenon that unduly reduces execution time: when $m$ is small, many states are removed in the pre-processing stage as unreachable from the initial state or as unable to reach any final state. Altogether, the issue of precise running time is complicated.

## 7 Conclusions

The algorithm in this paper looks complicated. To some extent it is because it was presented in great detail. A significant part of its implementation could be obtained by copying the pseudocode in the figures and the definitions of arrays in the main text, and converting them to the programming language in question. This is how the author implemented the prototype. Algorithm descriptions in research papers (like [11]) are often so sketchy that they are very hard to implement. The author wanted this not to be the case with the present paper.

Only one of the instances of the refinable partition data structure used by the algorithm uses *Mark*2 and *Split*2, and only one uses the bunch feature. These features are supported by additional arrays. Leaving them out from the instances that do not use them would improve the performance of the prototype.

The neighbour trick on lines 39 to 41 and Lemma 2 is ugly, because it breaks the otherwise clean abstract interface of the data structure. It also made it necessary to introduce *Outsets* and *Touched_Outsets*. In [11], a similar problem was solved by keeping track of how many transitions to each compound block each state has. Finding the appropriate counter quickly enough is not trivial, so the technique is somewhat complicated. It is plausible that something similar could have been done in the new algorithm. We leave it for the future to find out if it would work and be better than the chosen approach.

In verification of concurrent systems, it is common to use equivalence notions that abstract away from invisible actions. Bisimilarity does not do that. However, it preserves all commonly used equivalences. Therefore, the new algorithm can be used as a preprocessing stage that makes the graph smaller before it is given to a reduction or minimization algorithm of the equivalence in question. Because the new algorithm is cheap compared to most algorithms for other equivalences, this kind of preprocessing may save a lot of time in practice.

When minimizing with respect to observation equivalence by saturating the graph and then running bisimilarity minimization [8], the growth in the number of transitions caused by saturation is a problem. A natural, but apparently difficult, topic for future research is whether saturation could be replaced by adding suitable graph traversal to the new algorithm, without losing too much of its good performance.

## References

1. Aho, A.V., Hopcroft, J.E., Ullman, J.D.: The Design and Analysis of Computer Algorithms. Addison-Wesley, Reading (1974)
2. Browne, M.C., Clarke, E.M., Grumberg, O.: Characterizing Finite Kripke Structures in Propositional Temporal Logic. Theoret. Comput. Sci. 59, 115–131 (1988)
3. Dovier, A., Piazza, C., Policriti, A.: An Efficient Algorithm for Computing Bisimulation Equivalence. Theoret. Comput. Sci. 311, 221–256 (2004)
4. Fernandez, J.-C.: An Implementation of an Efficient Algorithm for Bisimulation Equivalence. Science of Computer Programming 13, 219–236 (1989/1990)
5. Gries, D.: Describing an Algorithm by Hopcroft. Acta Inform. 2, 97–109 (1973)
6. Gries, D.: The Science of Programming. Springer, Heidelberg (1981)
7. Hopcroft, J.: An $n \log n$ Algorithm for Minimizing States in a Finite Automaton. Technical Report CS-190, Stanford University (1970)
8. Kanellakis, P., Smolka, S.: CCS Expressions, Finite State Processes, and Three Problems of Equivalence. In: 2nd ACM Symposium on Principles of Distributed Computing, pp. 228–240 (1983)
9. Knuutila, T.: Re-describing an Algorithm by Hopcroft. Theoret. Comput. Sci. 250, 333–363 (2001)
10. Milner, R.: Communication and Concurrency. Prentice-Hall, Englewood Cliffs (1989)
11. Paige, R., Tarjan, R.: Three Partition Refinement Algorithms. SIAM J. Comput. 16(6), 973–989 (1987)
12. Valmari, A., Lehtinen, P.: Efficient Minimization of DFAs with Partial Transition Functions. In: Albers, S., Weil, P. (eds.) STACS 2008, Symposium on Theoretical Aspects of Computer Science, Bordeaux, France, pp. 645–656 (2008), http://drops.dagstuhl.de/volltexte/2008/1328/

# P-Semiflow Computation with Decision Diagrams

Gianfranco Ciardo[1], Galen Mecham[1], Emmanuel Paviot-Adet[2,3], and Min Wan[4]

[1] Dept. of Computer Science and Engineering, Univ. of California, Riverside
{ciardo,mgalen}@cs.ucr.edu
[2] Université P. & M. Curie, LIP6 - CNRS UMR 7606 - Paris, France
emmanuel.Paviot-Adet@lip6.fr
[3] Université Paris Descartes, Inst. Univ. de Technologie - Paris, France
[4] Yahoo! Inc.
minwan@yahoo-inc.com

**Abstract.** We present a symbolic method for p-semiflow computation, based on zero-suppressed decision diagrams. Both the traditional explicit methods and our new symbolic method rely on Farkas' algorithm, and compute a generator set from which any p-semiflow for the Petri net can be derived through a linear combination. We demonstrate the effectiveness of four variants of our algorithm by applying them on a suite of Petri net models, showing that our symbolic approach can produce results in cases where the explicit approach is infeasible.

## 1 Introduction

This section begins by briefly summarizing our Petri net notation, then it reviews the classic problem of p-semiflow computation, and gives some background on the class of decision diagrams we use.

### 1.1 Petri Nets

We adopt the standard definition of a Petri net, as a directed bipartite graph $(\mathcal{P}, \mathcal{T}, \mathbf{F}^-, \mathbf{F}^+)$, where

- $\mathcal{P}$ and $\mathcal{T}$ are sets of *places* and *transitions*, respectively drawn as circles and rectangles, satisfying $\mathcal{P} \cap \mathcal{T} = \emptyset$ and $\mathcal{P} \cup \mathcal{T} \neq \emptyset$. We let $n = |\mathcal{P}|$ and $m = |\mathcal{T}|$.
- A marking $\boldsymbol{\mu} \in \mathbb{N}^n$ assigns a number of tokens $\boldsymbol{\mu}_p$ to each place $p \in \mathcal{P}$.
- $\mathbf{F}^- : \mathcal{P} \times \mathcal{T} \to \mathbb{N}$ and $\mathbf{F}^+ : \mathcal{P} \times \mathcal{T} \to \mathbb{N}$, are $n \times m$ *incidence* matrices describing the cardinalities of the input arcs from $p \in \mathcal{P}$ to $t \in \mathcal{T}$ and of the output arcs from $t \in \mathcal{T}$ to $p \in \mathcal{P}$, respectively. Graphically, the cardinality is written on the arc, the default being 1, except that a missing arc indicates a cardinality of 0.

If the Petri net is *marked*, each place contains a number of *tokens*, collectively described by the *marking* $\mu : \mathcal{P} \to \mathbb{N}$. Starting from the *initial marking* $\mu^{init}$, the net then evolves as follows: (1) a transition $t$ is *enabled* in marking $\mu$ if $\boldsymbol{\mu}_p \geq \mathbf{F}^-_{p,t}$ for each place $p$, and (2) any enabled transition can *fire*, changing the marking of the net from $\boldsymbol{\mu}$ to $\boldsymbol{\mu}'$, where $\boldsymbol{\mu}'_p = \boldsymbol{\mu}_p - \mathbf{F}^-_{p,t} + \mathbf{F}^+_{p,t}$ for each place $p$.

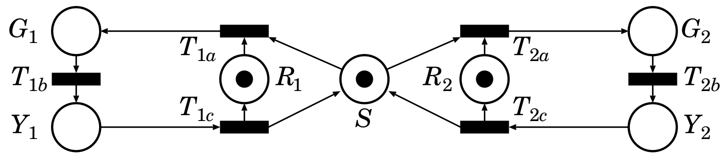


**Fig. 1.** Petri net of a simple traffic light controller

Fig. 1 shows an example of a Petri net modeling a traffic light controller at an intersection. Places $G_1$, $Y_1$, and $R_1$ represent the traffic lights for the north/south direction, while places $G_2$, $Y_2$, and $R_2$ represent the traffic lights for the east/west direction. A token in place $G_x$, $Y_x$, or $R_x$, represents that the corresponding green, yellow, or red light is on, respectively, for $x \in \{1, 2\}$. Transitions $T_{xa}$, $T_{xb}$, and $T_{xc}$ define the order through which a traffic light will cycle: first green, then yellow, then red, and back to green. For example, firing transition $T_{xa}$ consumes a token in place $R_x$ and produces a token in place $G_x$, transitioning from red to green. To ensure mutual exclusion, an additional place $S$ is used. Firing transition $T_{xc}$ to turn a light red produces a token in place $S$, while firing a transition $T_{xa}$ will consume that token. Fig. 1 shows the initial marking of the net with tokens in both $R_1$ and $R_2$ (representing the state of both lights being red) and a token in $S$ (meaning that either one of the two lights is allowed to transition to green, but not both).

## 1.2 Explicit P-Semiflow Generation

Invariant analysis is concerned with relationships satisfied by any reachable marking, thus based on the net structure rather than on the initial marking. Much work has focused on computing *p-semiflows* [4,3], i.e., non-zero solutions $\mathbf{w} \in \mathbb{N}^n$ to the set of linear "flow" equations $\mathbf{w} \cdot \mathbf{F} = \mathbf{0}$, where $\mathbf{F} = \mathbf{F}^+ - \mathbf{F}^-$ is the *flow matrix*. A p-semiflow $\mathbf{w}$ specifies the constraint $\sum_{p \in \mathcal{P}} \mathbf{w}_p \cdot \boldsymbol{\mu}_p = C$ on any reachable marking $\boldsymbol{\mu}$, where the initial marking $\boldsymbol{\mu}^{init}$ determines the constant $C = \sum_{p \in \mathcal{P}} \mathbf{w}_p \cdot \boldsymbol{\mu}^{init}_p$. However, invariant analysis provides necessary, not sufficient, conditions on reachability; a marking $\boldsymbol{\mu}$ might satisfy all known invariants and still be unreachable from $\boldsymbol{\mu}^{init}$ through a legal firing sequence.

Letting the *support* of a p-semiflow $\mathbf{w}$ be the set of places with a positive weight in $\mathbf{w}$, $Supp(\mathbf{w}) = \{p \in \mathcal{P} : \mathbf{w}_p > 0\}$, we say that a p-semiflow $\mathbf{w}$ is *minimal* if (a) it is *scaled back*, i.e., the greatest common divisor of its entries is 1, and (b) it has *minimal support*, i.e., no other p-semiflow $\mathbf{v}$ exists with a strictly smaller support $Supp(\mathbf{v}) \subset Supp(\mathbf{w})$.

Since any linear combination of p-semiflows is a p-semiflow, the set of scaled back p-semiflows is infinite unless it contains a single element. It is desirable

```
set of array[$m$+$n$] of int $ExpPSemiflows$(int $n$, int $m$, set of array[$m$+$n$] of int $\mathcal{A}$) is
 local int $j$;
 local array[$m$+$n$] of int $\mathbf{a}, \mathbf{a}_N, \mathbf{a}_P$;
 local set of array[$m$+$n$] of int $\mathcal{A}_N, \mathcal{A}_P$;
 1 for $j = 1$ to $m$ do
 2   $\mathcal{A}_N \leftarrow \emptyset$;                          • set of rows with negative entry $j$
 3   $\mathcal{A}_P \leftarrow \emptyset$;                          • set of rows with positive entry $j$
 4   foreach $\mathbf{a} \in \mathcal{A}$ do
 5     if $\mathbf{a}[j] < 0$ then $\mathcal{A}_N \leftarrow \mathcal{A}_N \cup \{\mathbf{a}\}$;
 6     if $\mathbf{a}[j] > 0$ then $\mathcal{A}_P \leftarrow \mathcal{A}_P \cup \{\mathbf{a}\}$;
 7   $\mathcal{A} \leftarrow \mathcal{A} \setminus (\mathcal{A}_N \cup \mathcal{A}_P)$;         • remove from $\mathcal{A}$ rows with nonzero entry $j$
 8   foreach $(\mathbf{a}_N, \mathbf{a}_P) \in \mathcal{A}_N \times \mathcal{A}_P$ do
 9     $v \leftarrow MinimumCommonMultiple(-\mathbf{a}_N[j], \mathbf{a}_P[j])$;
10     $\mathcal{A} \leftarrow \mathcal{A} \cup \{(-v/\mathbf{a}_N[j]) \cdot \mathbf{a}_N + (v/\mathbf{a}_P[j]) \cdot \mathbf{a}_P\}$;   • entry $j$ of new row is 0
11 return $\mathcal{A}$;
```

**Fig. 2.** An explicit algorithm to compute P-semiflows

to compute a *generator set* $\mathcal{W} = \{\mathbf{w}^{(1)}, ..., \mathbf{w}^{(r)}\}$ of minimal p-semiflows, from which any p-semiflow $\mathbf{w}$ can be derived as the non-negative linear combination $\mathbf{w} = \sum_{i=1}^{r} \alpha_i \mathbf{w}^{(i)}$, where $\alpha_i \in \mathbb{N}$, for $1 \leq i \leq r$, is uniquely determined by $\mathbf{w}$. It is well-known that the generator set is unique, since it contains all and only the minimal p-semiflows, but its size can be exponential in the number of places $n$.

An algorithm to compute all minimal p-semiflows, possibly plus some with non-minimal support, is based on Farkas' algorithm [5], which operates on a matrix $[\mathbf{T}|\mathbf{P}]$, initially set to $[\mathbf{F} \,|\, \mathbf{I}] \in \mathbb{Z}^{n \times (m+n)}$, the juxtaposition of the flow matrix $\mathbf{F} \in \mathbb{Z}^{n \times m}$ with the $n \times n$ identity matrix $\mathbf{I}$. The algorithm iteratively creates new rows with an increasing number of zero entries in the first $m$ columns. At the end, any remaining row $[\mathbf{t} \,|\, \mathbf{p}]$ of this matrix is such that the $m$ entries of $\mathbf{t}$ are all zero and the $n$ entries of $\mathbf{p}$ describe a p-semiflow. Algorithm *ExpPSemiflows* in Fig. 2 shows the pseudocode for this explicit approach, assuming that the matrix is stored as a set $\mathcal{A}$ of integer row vectors, each of length $m+n$ (treating the matrix as a set of rows is convenient, since we need to add and remove rows of this matrix). *ExpPSemiflows* works by iteratively *annulling* all the columns of $\mathbf{T}$ beginning with the leftmost column, 1, and ending with the rightmost column, $m$. To annul column $j$, it first removes from $\mathcal{A}$ all rows with a negative or positive entry in the $j^{th}$ column and puts them into two new sets $\mathcal{A}_N$ and $\mathcal{A}_P$, respectively. It then computes the pairwise linear combination of each row in $\mathcal{A}_N$ with each row in $\mathcal{A}_P$, choosing positive integer scalars such that the result has a zero entry in column $j$, and adds the resulting row back to set $\mathcal{A}$. Once all $m$ columns of $\mathbf{T}$ have been thus annulled, the resulting matrix $\mathbf{P}$ represents a set of p-semiflows for the net. The next step is to minimize this set of p-semiflows by both scaling back $\mathbf{P}$ and removing all p-semiflows with non-minimal support.

To scale back $\mathcal{A}$, we divide each row $\mathbf{a} \in \mathcal{A}$ by the greatest common divisor of all entries in $\mathbf{a}$, i.e., $\mathcal{A} \leftarrow (\mathcal{A} \setminus \{\mathbf{a}\}) \cup \{\mathbf{a}/gcd(\mathbf{a}[m+1], ..., \mathbf{a}[m+n])\}$. Then, we need to eliminate from $\mathcal{A}$ the non-minimal support p-semiflows. This can be done by considering each pair of distinct rows $\mathbf{x}$ and $\mathbf{y}$ in $\mathcal{A}$. If $Supp(\mathbf{x})$ is a proper subset of $Supp(\mathbf{y})$, then $\mathbf{y}$ is non-minimal and is eliminated from $\mathcal{A}$. Such an approach, however, requires $|\mathcal{A}| \cdot (|\mathcal{A}| - 1)/2$ comparisons of supports, each of size $n$, thus has time complexity $O(|\mathcal{A}|^2 \cdot n)$. For greater efficiency, we can eliminate rows of $\mathcal{A}$ during, rather than after, the execution of Algorithm *ExpPSemiflows*. This way, before the $j^{\text{th}}$ step, $\mathcal{A}$ contains only and all the minimal support p-semiflows of the subnet obtained from the original net by deleting the transitions corresponding to columns $j, ..., m$ of $\mathbf{T}$ [3]. Then, each newly-generated row is added to $\mathcal{A}$ in step 10 of Algorithm *ExpPSemiflows* only if it does not contain the support of a row already in $\mathcal{A}$. While the worst-case complexity remains $O(|\mathcal{A}|^2 \cdot n)$, the early elimination of as many rows as possible from $\mathcal{A}$ tends to be quite beneficial in practice.

To justify eliminating $\mathbf{y}$ at the $j^{\text{th}}$ step, we can show that there must exist a row $\mathbf{z}$ such that $Supp(\mathbf{z}) \subset Supp(\mathbf{y})$ and $\mathbf{y}$ can be obtained by scaling back a linear combination of $\mathbf{x}$ and $\mathbf{z}$. Let $k_z\mathbf{z} = k_y\mathbf{y} - k_x\mathbf{x}$, where $k_x$, $k_y$, and $k_z$ are positive integers, for any $p \in Supp(\mathbf{y})$ we have $k_y\mathbf{y}[p] \geq k_x\mathbf{x}[p]$, and there exists $q \in Supp(\mathbf{y})$ s.t. $k_y\mathbf{y}[q] = k_x\mathbf{x}[q]$. Such integers $k_x$, $k_y$, and $k_z$ can always be found. Then, $\mathbf{z}$ is a p-semiflow: $\mathbf{z} \cdot \mathbf{F} = 1/k_z(k_y\mathbf{y} - k_x\mathbf{x}) \cdot \mathbf{F}$, thus $\mathbf{z} \cdot \mathbf{F} = \mathbf{0}$. The support of $\mathbf{z}$ is strictly included in that of $\mathbf{y}$ and $k_y\mathbf{y} = k_x\mathbf{x} + k_z\mathbf{z}$. Then, $\mathbf{y}$ can be safely removed from the generator set. If $\mathbf{x}$ or $\mathbf{z}$ are not minimal, then other minimal p-semiflows exist to reconstruct them, and in turn to reconstruct $\mathbf{y}$.

Fig. 3 on the left shows the initial matrix $[\mathbf{T} \,|\, \mathbf{P}]$ for the traffic light example of Fig. 1, with null entries omitted for readability. Since $\mathbf{T}$ is initially the flow matrix $\mathbf{F}$, the first $m = 6$ columns encode the number of tokens that will be added to or subtracted from each place when the corresponding transition fires. For example, the second column tells us that firing transition $T_{1b}$ removes a token from $G_1$ and adds a token to $Y_1$, representing traffic light 1 transitioning from green to yellow. The same figure on the right shows the output of Algorithm *ExpPSemiflows*. The first $m$ columns are omitted, as they are all zero, and the last $n$ columns, initially encoding an identity matrix, now encode all minimal p-semiflows of the net. As no p-semiflow in the final matrix is a linear combination of any other p-semiflows, all p-semiflows have minimal support. In addition, all p-semiflows are obviously scaled back, as their entries are either 0 or 1, thus we have the generator set $\mathcal{W} = \{(1,1,1,0,0,0,0), (0,0,0,1,1,1,0), (1,1,0,1,1,0,1)\}$. Each $\mathbf{w} \in \mathcal{W}$ can be tested to ensure that it is indeed a p-semiflow by verifying that $\mathbf{w} \cdot \mathbf{F} = \mathbf{0}$.

| | **T** | | | | | | **P** | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $T_{1a}$ | $T_{1b}$ | $T_{1c}$ | $T_{2a}$ | $T_{2b}$ | $T_{2c}$ | $G_1$ | $Y_1$ | $R_1$ | $G_2$ | $Y_2$ | $R_2$ | $S$ |
| $G_1$ | 1 | −1 | | | | | 1 | | | | | | |
| $Y_1$ | | 1 | −1 | | | | | 1 | | | | | |
| $R_1$ | −1 | | 1 | | | | | | 1 | | | | |
| $G_2$ | | | | 1 | −1 | | | | | 1 | | | |
| $Y_2$ | | | | | 1 | −1 | | | | | 1 | | |
| $R_2$ | | | | −1 | | 1 | | | | | | 1 | |
| $S$ | −1 | | 1 | −1 | | 1 | | | | | | | 1 |

| **$\mathbf{P}_{\mathbf{final}}$** | | | | | | |
|---|---|---|---|---|---|---|
| $G_1$ | $Y_1$ | $R_1$ | $G_2$ | $Y_2$ | $R_2$ | $S$ |
| 1 | 1 | 1 | | | | |
| | | | 1 | 1 | 1 | |
| 1 | 1 | | 1 | 1 | | 1 |

**Fig. 3.** Initial and final matrices for the traffic light Petri net

Since $\mathcal{W}$ contains p-semiflows for the traffic light controller, we can use it to gain valuable insights into how the controller functions. For example, the p-semiflow $(1,1,1,0,0,0,0)$, together with the initial marking, implies that exactly one token will be circulating in $G_1$, $Y_1$, and $R_1$, in any reachable marking, i.e., at any given moment of time, exactly one of the three lights (red, yellow, green) for the north/south direction will be on. The p-semiflow $(0,0,0,1,1,1,0)$ specifies the analogous constraint for the east/west direction. Finally, the p-semiflow $(1,1,0,1,1,0,1)$, together with the initial marking, specifies the constraint that exactly one place in $\{G_1,Y_1,G_2,Y_2,S\}$ contains a token, i.e., if the green or yellow light in the north/south direction is on, then the green and yellow lights in the east/west direction are off, and vice versa. One final aspect of these results is that each place is included in the support of at least one p-semiflow. This indicates that the state space for the net is finite in size for any initial marking.

## 1.3 Decision Diagrams

To increase the time and memory efficiency of p-semiflow generation, we use *decision diagrams*, in particular, a variant of extensible multi-way decision diagrams (MDDs) [12] where variable domains are a-priori unknown sets of integers, i.e., each variable can assume a negative, zero, or positive value. Formally, given a set of $L$ variables ordered as $x_L \succ x_{L-1} \succ \cdots \succ x_1$, we define such a decision diagram as a directed acyclic edge-labeled multi-graph such that:

– A *nonterminal* node $p$ is associated with a variable $p.var = x_k$, $L \geq k \geq 1$, and has an infinite set of outgoing edges, each indexed by a different integer.
– The only *terminal* nodes, with no outgoing edges, are $\mathbf{0}$ and $\mathbf{1}$. For ease of notation, we let $\mathbf{0}.var = \mathbf{1}.var = x_0$, where $x_k \succ x_0$ for $L \geq k \geq 1$.
– The edge with index $i$ originating from nonterminal node $p$ points to a node $q$ satisfying $p.var \succ q.var$. We write this as $p[i] = q$.

We use a *zero-suppressed* [10] semantic for an edge skipping variables, i.e., $p[i] = q$, with $p.var = x_k$, $q.var = x_h$, and $k > h+1$, is equivalent to a path $p[i] = p_{k-1}$, $p_{k-1}[0] = p_{k-2}$, ..., $p_{h+1}[0] = q$, where each intermediate node $p_l$ is associated with variable $x_l$ and is such that, except for the outgoing edge indexed by 0, all its other edges point to $\mathbf{0}$. We then define the set of $k$-tuples encoded by a node $p$, with $p.var = x_k$, as

$$\mathcal{X}(p) = \begin{cases} \emptyset & \text{if } p = \mathbf{0} \\ \{\epsilon\} & \text{if } p = \mathbf{1} \\ \bigcup_{i:p[i]\neq\mathbf{0}\wedge p[i].var=x_h} \{i\}\cdot\{0^{k-h-1}\}\cdot\mathcal{X}(p[i]) & \text{otherwise,} \end{cases}$$

where $\{\epsilon\}$ indicates the set containing only the empty tuple, $0^t$ indicates a tuple of length $t \geq 0$ of zeros, and "$\cdot$" is the ordinary concatenation operator.

Conceptually, a node $p$ has an infinite number of outgoing edges. We enforce a finite representation by requiring that only a finite number of outgoing edges point to nodes other than $\mathbf{0}$, and storing only these edges. We can then define

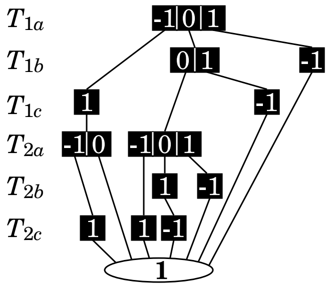


**Fig. 4.** ZMDD encoding the flow matrix **F** for our Petri net

two canonical forms, a quasi-reduced and a fully-reduced one[1]. Either one forbids nodes where all edges point to **0** and duplicates, i.e., distinct nodes associated to the same variable must have different edge sets: if $p.var = q.var$, then there must be an $i$ such that $p[i] \neq q[i]$. Then, the quasi-reduced form, QMDD, forbids variable-skipping altogether, except for edges pointing to **0**: if $p.var = x_k$, any edge $p[i] = q \neq \mathbf{0}$ satisfies $q.var = x_{k-1}$. The zero-suppressed, form, ZMDD, forces instead variable-skipping whenever possible: there can exist no node $p$ where only $p[0]$ does not point to **0**. In the following we use the term MDD to indicate that our algorithms can be implemented either with QMDDs or with ZMDDs, however, we assume ZMDDs, as they simplify the pseudocode for the algorithms we introduce, and are more efficient in practice, since most p-semiflows are usually very sparse, especially for large nets.

Fig. 4 shows the ZMDD encoding the flow matrix **F**, as a set of tuples (rows), for our example. The terminal **0** and any edge pointing to it are omitted.

## 2 Our Contribution

As described in Sect. 1.2, the time and memory required to build a generator set is at least proportional to its size, which can be exponential in $m$. We then present a symbolic algorithm that, by using MDDs, has the potential to be much more efficient in many practical nets.

### 2.1 A Symbolic Algorithm to Compute P-Semiflows

Algorithm *SymPSemiflows* in Fig. 5 is a symbolic implementation of p-semiflows computation. Unlike the explicit version of Fig. 2, *SymPSemiflows* operates on an MDD $a$ rather than on a matrix **A**. The MDD $a$ encodes the matrix $[\mathbf{T}|\mathbf{P}]$ as a set of rows over $L = m + n$ variables, $v_1 \succ \cdots \succ v_m \succ w_1 \succ \cdots \succ w_n$ (we still associate variable $x_0$ to the terminal nodes).

[1] The proofs of canonicity for the quasi-reduced and the fully-reduced forms are analogous to that for quasi-reduced BDDs [9] and ZBDDs [10], respectively.

*Union* and *Intersection* in Fig. 5 are the usual MDD disjunction and conjunction operations. $SymLinComb(\rho_x, x, \rho_y, y)$ is instead used to symbolically compute *all* pairwise linear combinations of the rows encoded by MDDs $x$ and $y$, multiplied respectively by the positive integers $\rho_x$ and $\rho_y$. As with all decision diagram manipulations, the recursive algorithm starts at the root and ends at the terminals. However, the recursion might end early if the desired result is

```
mdd $SymPSemiflows$(int $m$, mdd $a$) is
 local  int  $j, i_N, i_P, \rho, \rho_N, \rho_P$;
 local  mdd $a_N, a_P$;
  1 for $j = 1$ to $m$ do
  2   if $a.var \neq v_j$ skip;          • nothing to do if $a$ encodes only rows with $v_j = 0$
  3   $a_N \leftarrow Intersection(a, Potential(v_j < 0))$;        • set of rows with negative $v_j$
  4   $a_P \leftarrow Intersection(a, Potential(v_j > 0))$;        • set of rows with positive $v_j$
  5   $a \leftarrow Intersection(a, Potential(v_j = 0))$;       • redefine $a$ for the next iteration
  6   foreach $i_N$ s.t $a_N[i_N] \neq \mathbf{0}$ do
  7     foreach $i_P$ s.t. $a_P[i_P] \neq \mathbf{0}$ do
  8       $\rho \leftarrow MinimumCommonMultiple(-i_N, i_P)$;
  9       $\rho_N \leftarrow \rho/(-i_N)$;
 10       $\rho_P \leftarrow \rho/i_P$;
 11       $a \leftarrow Union(a, SymLinComb(\rho_N, a_N[i_N], \rho_P, a_P[i_P]))$;
 12 return $a$;
```

```
mdd $SymLinComb$(int $\rho_x$, mdd $x$, int $\rho_y$, mdd $y$) is
 local  int  $i_x, i_y, t$;
 local  mdd $r$;
  1 if $x = \mathbf{1}$ and $y = \mathbf{1}$ then return $\mathbf{1}$;
  2 if $x = \mathbf{0}$ or $y = \mathbf{0}$ then return $\mathbf{0}$;
  3 if $InCache(C_{SymLinComb}, \rho_x, x, \rho_y, y, r)$ then return $r$;
  4 if $x.var \succ y.var$ then                                   • $y.var$ is skipped
  5   $r \leftarrow NewNode(x.var)$;
  6   foreach $i_x$ s.t. $x[i_x] \neq \mathbf{0}$ do
  7     $r[\rho_x i_x] \leftarrow SymLinComb(\rho_x, x[i_x], \rho_y, y)$;
  8 else if $y.var \succ x.var$ then                              • $x.var$ is skipped
  9   $r \leftarrow NewNode(y.var)$;
 10   foreach $i_y$ s.t. $y[i_y] \neq \mathbf{0}$ do
 11     $r[\rho_y i_y] \leftarrow SymLinComb(\rho_x, x, \rho_y, y[i_y])$;
 12 else                                                          • $y.var = x.var$
 13   $r \leftarrow NewNode(x.var)$;
 14   foreach $i_x$ s.t. $x[i_x] \neq \mathbf{0}$ do
 15     foreach $i_y$ s.t. $y[i_y] \neq \mathbf{0}$ do
 16       $t \leftarrow \rho_x i_x + \rho_y i_y$;
 17       $r[t] \leftarrow Union(r[t], SymLinComb(\rho_x, x[i_x], \rho_y, y[i_y]))$
 18 $r \leftarrow UniqueTableInsert(r)$;
 19 $CacheAdd(C_{SymLinComb}, \rho_x, x, \rho_y, y, r)$;
 20 return $r$;
```

**Fig. 5.** A symbolic algorithm to compute p-semiflows

found in the *operation cache*, implemented as a hash table searched with a key $(C_x, ops)$, where $C_x$ is the code for operation $x$ and *ops* are the operands. The pseudocode accesses this cache through two functions: $InCache(C_x, ops, res)$ fills *res* with the result and returns *true* if $(C_x, ops)$ is a hit, otherwise it leaves *res* unchanged and returns *false*, while $CacheAdd(C_x, ops, res)$ inserts *res* as the lookup value for $(C_x, ops)$.

$Potential(cond)$ returns the set of rows satisfying boolean condition *cond*. As we assume integer variable domains, this set could be infinite but, in our specific case, this is not a problem because we use *Potential* only in an intersection with a finite set of rows $\mathcal{X}(a)$. Thus, in practice, we can modify the *Intersection* operator to "fake" the second argument, as condition *cond* in our case can be enforced on the first argument, node $a$, which is associated to variable $v_j$.

Algorithm *SymPSemiflows* works by eliminating variable $v_1$ through $v_m$, in that order. This accomplishes the task of annulling each column in matrix $\mathbf{T}$, after which we obtain the MDD encoding the p-semiflows. Given MDD $a$, the process of annulling a column at $a.var = v_j$ involves the following steps:

1. Create two new MDDs $a_P$ and $a_N$, encoding the set of rows with positive $v_j$ and negative $v_j$, respectively.
2. Remove these rows from $a$, which results in $a$ encoding only rows with null $v_j$. Due to the ZMDD encoding, $a$ is set to $a[0]$, thus the height of $a$ decreases by at least one.
3. Perform the pairwise linear combination between each $a_P[i_P]$ and $a_N[i_N]$.
4. Finally, combine the resulting MDD with $a$ using a *Union* operation.

Due to our use of ZMDDs and to the for-loop order, line 2 of *SymPSemiflows* effectively checks whether all rows encoded by $a$ have $v_1 = ... = v_j = 0$. After executing line 5, we are guaranteed that this is the case, thus $a.var = v_{j+1}$, or possibly even lower. Also, if, at iteration $j$ some rows have $v_j > 0$ but no row has $v_j < 0$, or vice versa, i.e., if $a_N = \mathbf{0}$ and $a_P \neq \mathbf{0}$ or $a_N \neq \mathbf{0}$ and $a_P = \mathbf{0}$, no work is required, and, at iteration $j+1$, $a$ simply encodes the rows with $v_j = 0$.

Algorithm *SymLinComb* recursively performs the pairwise symbolic linear combination of all rows encoded by MDDs $x$ and $y$, using $\rho_x$ and $\rho_y$ as weights. Given two MDDs with $x.var = y.var$, we create a new node $r$ such that $r.var = x.var = y.var$. Then, for each pair of edges $x[i_x]$ and $y[i_y]$, we add a new edge $r[t]$ to $r$, where $t = \rho_x i_x + \rho_y i_y$. This edge points to the symbolic linear combination of $x[i_x]$ and $y[i_y]$, still with the same weights $\rho_x$ and $\rho_y$. If $r$ already contains an edge $r[t]$, a union is performed. A special case occurs when $x.var \neq y.var$. For example, if $x.var > y.var$, then $y$ encodes only rows with null $x.var$ and, we only need to scale each edge of $x[i_x] \neq \mathbf{0}$ by the scalar $\rho_x$, with each edge pointing to the linear combination of $x[i_x]$ and $y$.

### 2.2 Removing Non-minimal P-Semiflows

Recall that, at the start of algorithm *SymPSemiflows*, MDD $a$ has $m+n$ variables $v_1 \succ \cdots \succ v_m \succ w_1 \succ \cdots \succ w_n$, with $v_1, ..., v_m$ corresponding to the transitions.

At the end of any given iteration, the p-semiflows found thus far are represented by all nodes $p$ s.t. $p.var \in \{w_1, ..., w_n\}$. Thus, the nodes above variable $w_1$ must be ignored to obtain the current set of p-semiflows. For this, we define an MDD operation called *Prune* which, given an MDD $a$, returns the *Union* of all nodes $p$ such that (1) $p$ is either $a$ or a descendant of $a$, and (2) either $p.var = w_1$ or $w_1 \succ p.var$ and $p$ is pointed to by an edge from a node $q$ s.t. $q.var \succ w_1$. Then, the set of semiflows represented by an MDD $a$ with $a.var \succeq w_1$ is $\mathcal{X}(Prune(a))$.

*SymPSemiflows*, just like *ExpPSemiflows*, can compute some non-minimal p-semiflows in addition to the minimal ones. Recall that there are two causes that can make a p-semiflow non-minimal: its support is not minimal, or its support is minimal, but it is not scaled back. We now describe how to eliminate these non-minimal p-semiflows from the result returned by *SymPSemiflows*.

## 2.3 Removing Non-minimal Support P-Semiflows

The elimination of non-minimal support p-semiflows is not a trivial task. Various explicit approaches have been proposed [3], differing in how and when non-minimal testing and elimination take place. Recall the following proposition [3]:

**Proposition 1.** Let $[\mathbf{T}|\mathbf{P}]$ be the matrix obtained after annulling $k-1$ columns, where $\mathbf{P}$ only contains the minimal p-semiflows of $\mathcal{A}^{(k-1)}$. Let $[\mathbf{t}_u|\mathbf{p}_u]$ be a row obtained (not necessarily the first row) while annulling column $k$ of $[\mathbf{T}|\mathbf{P}]$ as a combination of $[\mathbf{t}_i|\mathbf{p}_i]$ and $[\mathbf{t}_j|\mathbf{p}_j]$. Then, $\mathbf{p}_u$ is a non-minimal p-semiflow of $\mathcal{A}^k$ iff there exists a row $\mathbf{p}_e$ in $\mathbf{P}$ s.t. $\mathbf{p}_i \neq \mathbf{p}_e \neq \mathbf{p}_j$ and $Supp(\mathbf{p}_e) \subseteq Supp(\mathbf{p}_u)$.

Proposition 1 implies that the support of a newly generated row cannot be a subset of the support of a row already in $[\mathbf{T}|\mathbf{P}]$, thus new rows never eliminate old rows. It is possible for the support of a newly generated row to equal that of a row already in $[\mathbf{T}|\mathbf{P}]$, but, if this occurs, we must retain only the old row, because the old row has $v_k = 0$, therefore cannot be one of the rows involved in the computation of a p-semiflow at step $k$.

The elimination of non-minimal support p-semiflows can be performed periodically during the computation of *SymPSemiflows*, or at the end of the algorithm. As the number of p-semiflows can be exponential, it is often beneficial to perform non-minimal support p-semiflow elimination as the algorithm progresses. Algorithm *MinSuppInt* in Fig. 6 performs an internal comparison to eliminate a p-semiflow that can be obtained via the linear combination of at least two other p-semiflows also in $a$. We can therefore use a single MDD $a$ to store the result of the p-semiflow computation, possibly refining $a$ periodically using *MinSuppInt*. After each call to *SymLinComb*, the result is immediately unioned with $a$. Since newly generated rows cannot eliminate old rows, Algorithm *MinSuppInt* can be applied at any time to the intermediate result of the p-semiflow computation, to reduce the number of rows.

Another strategy is instead to use a second MDD $b$ as an accumulator, storing either the result of a single linear combination *SymLinComb*, or the union of multiple linear combinations. MDD $a$ encodes a set of minimal support p-semiflows. Algorithm *MinSuppExt* merges $a$ with the intermediate results in $b$, by first applying *ElimNMSupp* to $b$ to remove from $b$ all p-semiflows which have non-minimal supports with respect to the other p-semiflows in $b$, and then performing an external elimination step during which it removes any p-semiflow found in $b$ whose support is either found in $a$ or can be generated as a union of supports from both $a$ and $b$. As Proposition 1 guarantees that a new row cannot eliminate an old one, we do not have to perform an external elimination step on $a$ with respect to $b$. The result is a refined MDD $b$ containing p-semiflows, each of them having a support which is minimal with respect to all other p-semiflows in both $a$ and $b$. MDDs $a$ and $b$ can then be safely unioned.

Before describing Algorithms *MinSuppInt* and *MinSuppExt*, we define some helper functions they utilize. Algorithm *MkBool* in Fig. 6 performs a fixpoint iteration to compute the MDD encoding the supports of all p-semiflows (minimal or otherwise). In other words, $MkBool(a)$ takes an MDD $a$ encoding the set of p-semiflows $\mathcal{X}(Prune(a))$ and returns the boolean MDD, i.e., a (zero-suppressed) BDD, encoding the set of boolean vectors

$$\{\mathbf{b}\in\mathbb{B}^n : \exists\mathbf{x}\in\mathcal{X}(Prune(a)), \forall i, 1\le i\le n, \mathbf{b}[i]=1 \Leftrightarrow \mathbf{x}[i]>0\}.$$

Lines 4–7 of *MkBool* implement the *Prune* operation, thus eliminate all nodes associated with variables above $w_1$. Lines 9–12 build a node representing the boolean equivalent of node $a$. In all our pseudocode, type bdd indicates a BDD on variables $w_1 \succ \cdots \succ w_n$.

Algorithm *Filter* eliminates from $\mathcal{X}(Prune(a))$ any path representing a p-semiflow whose support is obtainable as the union of two or more p-semiflows in $\mathcal{X}(Prune(a))$. It does so by taking an MDD $a$ and a minimal support BDD $b$ and computing the MDD $t$ such that $t$ contains all p-semiflows found in $\mathcal{X}(Prune(a))$ whose support is found in $b$. In other words, *Filter* returns the MDD encoding

$$\{\mathbf{x}\in\mathcal{X}(Prune(a)) : \exists\mathbf{b}\in\mathcal{X}(b), \forall i, 1\le i\le n, \mathbf{b}[i]=1 \Leftrightarrow \mathbf{x}[i]>0\}.$$

Lines 10–20 filter all nodes at or below $w_1$. Since the transition information above variable $w_1$ is still pertinent, lines 6–8 copy these nodes into the result.

Algorithm *EWOr* in Fig. 6 is needed when eliminating all non-minimal supports from a BDD. *EWOr* takes two BDDs $p$ and $q$ in input and returns a BDD $r$ encoding the "element-wise-or" of all pairs of tuples, one from $\mathcal{X}(p)$ and one from $\mathcal{X}(q)$, i.e., $\mathcal{X}(r) = \{\mathbf{i}\vee\mathbf{j} : \mathbf{i}\in\mathcal{X}(p), \mathbf{j}\in\mathcal{X}(q)\}$. This is an unusual operation for BDDs, quite unlike the much more familiar (non-element-wise) union of sets encoded by two BDDs, but it nevertheless has an efficient and elegant symbolic implementation. To generating $r[0]$, we simply have to recursively call *EWOr* on $p[0]$ and $q[0]$ (line 11). This is because the element-wise-or $\mathbf{t}$ of a pair of tuples from $\mathbf{i}\in\mathcal{X}(p)$ and $\mathbf{j}\in\mathcal{X}(q)$ cannot produce a 1 in position $\mathbf{t}_{(0)}$ if $\mathbf{i}_{(0)} = \mathbf{j}_{(0)} = 0$. Generating $r[1]$ is instead more involved, as the element-wise-or $\mathbf{t}$ of any pair of tuples from $\mathbf{i}\in\mathcal{X}(p)$ and $\mathbf{j}\in\mathcal{X}(q)$ can have a 1 in position $\mathbf{t}_{(0)}$ if either $\mathbf{i}_{(0)} = 1$, or $\mathbf{j}_{(0)} = 1$, or both. We must therefore recursively call *EWOr* on three pairs of nodes: $p[0]$ and $q[1]$, $p[1]$ and $q[0]$, and, finally, $p[1]$ and $q[1]$. An ordinary *Union* operation is used to combine the three results together and obtain $r[1]$.

Algorithm *ElimNMSupp* performs the actual work of eliminating from a BDD all non-minimal supports, taking a BDD $p$ and returning a BDD $r$ that contains

```
bdd $MkBool$(mdd $a$) is
local int $i$;
local bdd $b$;
if $a.var = x_0$ then return $a$;
if $InCache(C_{MkBool}, a, b)$ then
  return $b$;
if $a.var \succ w_1$ then
  $b \leftarrow \mathbf{0}$;
  foreach $i$ s.t. $a[i] \neq \mathbf{0}$ do
    $b \leftarrow Union(b,\ MkBool(a[i]))$;
else
  $b \leftarrow NewNode(a.var)$;
  if $a[0] \neq \mathbf{0}$ then $b[0] \leftarrow MkBool(a[0])$;
  foreach $i > 0$ s.t. $a[i] \neq \mathbf{0}$ do
    $b[1] \leftarrow Union(b[1], MkBool(a[i]))$;
$b \leftarrow UniqueTableInsert(b)$;
$CacheAdd(C_{MkBool}, a, b)$;
return $b$;
```

```
bdd $EWOr$(bdd $p$, bdd $q$) is
local bdd $r$, $r_{01}$, $r_{10}$, $r_{11}$;
if $p = \mathbf{0}$ or $q = \mathbf{0}$ then return $\mathbf{0}$;
if $p = q$ then return $p$;
if $InCache(C_{EWOr}, p, q, r)$ then
  return $r$;
if $q.var \succ p.var$ then $Swap(p, q)$;
$r \leftarrow NewNode(p.var)$;
if $p.var \succ q.var$ then
  $r[0] \leftarrow EWOr(p[0], q)$;
  $r[1] \leftarrow EWOr(p[1], q)$;
else
  $r[0] \leftarrow EWOr(p[0], q[0])$;
  $r_{01} \leftarrow EWOr(p[0], q[1])$;
  $r_{10} \leftarrow EWOr(p[1], q[0])$;
  $r_{11} \leftarrow EWOr(p[1], q[1])$;
  $r[1] \leftarrow Union(r_{01}, Union(r_{10}, r_{11}))$;
$r \leftarrow UniqueTableInsert(r)$;
$CacheAdd(C_{EWOr}, p, q, r)$;
return $r$;
```

```
mdd $MinSuppExt$(mdd $dirty$, mdd $clean$) is
local bdd $b_{dirty}$, $b_{clean}$, $b_{union}$, $b_{min}$;
$b_{dirty} \leftarrow MkBool(dirty)$;
$b_{dirty} \leftarrow ElimNMSupp(b_{dirty})$;        • int.
$b_{clean} \leftarrow MkBool(clean)$;
$b_{union} \leftarrow EWOr(b_{dirty}, b_{clean})$;
$b_{min} \leftarrow Diff(b_{dirty}, b_{union})$         • ext.
return $Filter(dirty, b_{min})$;
```

```
mdd $Filter$(mdd $a$, bdd $b$) is
local int $i$;
local mdd $f$;
if $a = \mathbf{0}$ or $b = \mathbf{0}$ then return $\mathbf{0}$;
if $a = \mathbf{1}$ and $b = \mathbf{1}$ then return $\mathbf{1}$;
if $InCache(C_{Filter}, a, b, f)$ then
  return $f$;
$f \leftarrow NewNode(Highest(a.var, b.var))$;
if $a.var \succ w_1$ then
  foreach $i$ s.t. $a[i] \neq \mathbf{0}$ do
    $f[i] \leftarrow Filter(a[i], b)$;
else if $a.var \succ b.var$ then
  if $a[0] \neq \mathbf{0}$ then
    $f \leftarrow Filter(a[0], b)$;
else if $b.var \succ a.var$ then
  if $b[0] \neq \mathbf{0}$ then
    $f \leftarrow Filter(a, b[0])$;
else        • $a.var = b.var$, filter node $a$
  if $a[0] \neq \mathbf{0}$ and $b[0] \neq \mathbf{0}$ then
    $f[0] \leftarrow Filter(a[0], b[0])$;
  if $b[1] \neq \mathbf{0}$ then
    foreach $i > 0$ s.t. $a[i] \neq \mathbf{0}$ do
      $f[i] \leftarrow Filter(a[i], b[1])$;
$f \leftarrow UniqueTableInsert(f)$;
$CacheAdd(C_{Filter}, a, b, f)$;
return $f$;
```

```
bdd $ElimNMSupp$(bdd $p$) is
local bdd $r$, $r_0$, $r_1$, $t$;
if $p.var = x_0$ then return $p$;
if $InCache(C_{ElimNMSupp}, p, r)$ then
  return $r$;
$r_0 \leftarrow ElimNMSupp(p[0])$;
$r_1 \leftarrow ElimNMSupp(p[1])$;
if $r_0 = r_1$ then return $r_0$;
$r \leftarrow NewNode(p.var)$;
$r[0] \leftarrow r_0$;
$r[1] \leftarrow r_1$;
$t \leftarrow EWOr(r[0], r[1])$;
$r[1] \leftarrow Diff(r[1], t)$;
$r \leftarrow UniqeTableInsert(r)$;
$CacheAdd(C_{ElimNMSupp}, p, r)$;
return $r$;
```

```
mdd $MinSuppInt$(mdd $a$) is
local bdd $b$;
$b \leftarrow MkBool(a)$;
$b \leftarrow ElimNMSupp(b)$;
return $Filter(a, b)$;
```

**Fig. 6.** Symbolic minimal support P-semiflow generation

only the minimal supports in $p$. As with *EWOr*, the case for $p[0]$ is simpler, requiring only a recursive call to *ElimNMSupp* on $p[0]$ (line 4). To generate $r[1]$, we begin by recursively calling *ElimNMSupp* on $p[1]$. However, $r[1]$ could now contain supports obtainable by element-wise-or from $\mathcal{X}(r[0])$ and $\mathcal{X}(r[1])$. Such supports must be eliminated from $r[1]$ to enforce minimality (lines 9–11). First, we use *EWOr* to generate $t$, the BDD encoding supports obtainable via linear combinations between $\mathcal{X}(r[0])$ and $\mathcal{X}(r[1])$, then we use a set difference, *Diff*, to remove these supports from $r[1]$. Algorithms *EWOr* and *ElimNMSupp* are essential to the efficiency of our approach, as they allow us to reduce the cost of recognizing and eliminating rows with nonminimal support symbolically, and they are fundamentally different from the way the explicit approach operates.

We can now discuss algorithms to generate the minimal support p-semiflows. Algorithm *MinSuppInt*, shown in Fig. 6, is relatively straightforward. First, it generates a BDD $b$ encoding all minimal supports. This is done by combining *MkBool* and *ElimNMSupp* to generate all the minimal supports found in a MDD $a$. Then, it uses *Filter* to remove all p-semiflows from $a$ whose support is not found in $b$. Algorithm *MinSuppExt* instead takes as input two MDDs, *dirty*, containing minimal support p-semiflows in addition to possibly non-minimal support ones, and *clean*, containing only minimal support p-semiflows. Then, it eliminates from *dirty* all non-minimal support p-semiflows, whose support can be generated as an element-wise-or of supports from *dirty* and *clean*. This algorithm can be used to compute the union of the p-semiflows in *clean* and *dirty*, by first eliminating from the latter all p-semiflows that would make the resulting MDD non-minimal.

## 2.4 When to Perform Non-minimal Support P-Semiflow Elimination

Algorithm *SymPSemiflows* of Fig. 5 computes the p-semiflows of a net, but does not minimize them. We propose four variants for doing this, see Fig. 7. The first three variants perform non-minimal support elimination while executing *SymPSemiflows*: V1 minimizes immediately after performing each linear combination, while V2 and V3 minimize after annulling a column. We compare these with V4, which simply performs minimization only once, after *SymPSemiflows* has completed its work; as we will see empirically, this last option can be the best, but it can also perform poorly, if the (more numerous) p-semiflows it manages during the execution do not lend themselves to a compact MDD encoding.

Given an MDD $a$, V1 calls *MinSuppExt* immediately after performing each symbolic linear combination, to minimize the result with respect to $a$, then unions it to $a$; this has the advantage of eliminating non-minimal support p-semiflows as soon as possible, thus is potentially very space efficient. V3 simply calls *MinSuppInt* on $a$ after annulling a column and is as simple to implement as V4. V2 instead uses an MDD *linComb* to accumulate the newly computed p-semiflows then, once the entire column has been annulled, it minimizes *linComb* with respect to $a$ using *MinSuppExt*, and unions it with $a$.

```
mdd $SymPSemiflows$(int $m$, mdd $a$) is                    • common portion to all variants
local  int  $j, i_N, i_P, \rho, \rho_N, \rho_P,$;
local  mdd $a_N, a_P, linComb, newRows$;                      • $newRows$ is used only in V2
 1  for $j = 1$ to $m$ do
 2    if $a.var \neq v_j$ skip;
 3    $a_N \leftarrow Intersection(a, Potential(v_j < 0))$;
 4    $a_P \leftarrow Intersection(a, Potential(v_j > 0))$;
 5    $a \leftarrow Intersection(a, Potential(v_j = 0))$;
 6    $newRows \leftarrow \mathbf{0}$;                                              • only for V2
 7    foreach $i_N$ s.t $a_N[i_N] \neq \mathbf{0}$ do
 8      foreach $i_P$ s.t. $a_P[i_P] \neq \mathbf{0}$ do
 9        $\rho \leftarrow MinimumCommonMultiple(-i_N, i_P)$;
10        $\rho_N \leftarrow \rho/(-i_N)$;
11        $\rho_P \leftarrow \rho/i_P$;
              • V1: Minimize after each linear combination (using external comparisons)
$12_1$        $linComb = SymLinComb(\rho_N, a_N[i_N], \rho_P, a_P[i_P])$;
$13_1$        $a \leftarrow Union(a, MinSuppExt(linComb, a))$;
$14_1$ return $a$;
                    • V2: Minimize after annulling column (using external comparisons)
$15_2$        $linComb \leftarrow SymLinComb(\rho_N, a_N[i_N], \rho_P, a_P[i_P])$;
$16_2$        $newRows \leftarrow Union(newRows, linComb)$;
$17_2$    $a \leftarrow Union(a, MinSuppExt(newRows, a))$;
$18_2$ return $a$;
                    • V3: Minimize after annulling column (using internal comparisons)
$19_3$        $a \leftarrow Union(a, SymLinComb(\rho_N, a_N[i_N], \rho_P, a_P[i_P]))$;
$20_3$    $a \leftarrow MinSuppInt(a)$;
$21_3$ return $a$;
                          • V4: Minimize only at the end (using internal comparisons)
$22_4$        $a \leftarrow Union(a, SymLinComb(\rho_N, a_N[i_N], \rho_P, a_P[i_P]))$;
$23_4$ return $MinSuppInt(a)$;
```

**Fig. 7.** Minimal support computation on-the-fly

## 2.5 Scaling Back P-Semiflows

After eliminating all p-semiflows with non-minimal support, we scale back the remaining ones using the brute force Algorithm *SymScalePsemiflows* in Fig. 8. This in turn uses Algorithm *ScaleByNumber*, which takes an MDD $a$ and an integer $\mu$ and returns two MDDs: $s$, encoding all scaled-back p-semiflows in $a$ which could be scaled by $\mu$, $\mathcal{X}(s) = \{\mathbf{x} \in \mathbb{N}^n : \exists \mathbf{a} \in \mathcal{X}(a), \mathbf{x} = \mathbf{a}/\mu\}$, and $u$, encoding those that could not, $\mathcal{X}(u) = \{\mathbf{a} \in \mathcal{X}(a) : \mathbf{a}/\mu \notin \mathbb{N}^n\}$. *SymScalePsemiflows* takes an MDD $a$ and scales its p-semiflows until none is divisible by an integer greater than one. First, it finds the largest value $\gamma$ contained in $a$, then it repeatedly attempts to scale $a$ by the prime numbers between 2 and $\sqrt{\gamma}$, using *ScaleByNumber*. Each time, the two MDDs returned by *ScaleByNumber* are unioned back to produce the new scaled-back MDD.

```
mdd $SymScalePsemiflows$(mdd $a$) is
 local int $\gamma, \mu$;
 local mdd $s, u$;
 1 $\gamma \leftarrow \max_{i \in \mathbb{N}}\{p[i] \neq \mathbf{0}$ : $p$ is a node in the MDD $a\}$;
 2 foreach $\mu \in \{2, ..., \lfloor\sqrt{\gamma}\rfloor$ : $\mu$ is prime$\}$ do
 3   repeat
 4     $\langle s, u\rangle \leftarrow ScaleByNumber(a, \mu)$;                • $s$ encodes scaled paths
 5     $a \leftarrow Union(s, u)$;     • update $a$ by combining scaled and unscaled paths
 6   until $s = \mathbf{0}$;
 7 return $a$;
```

```
$\langle$mdd, mdd$\rangle$ $ScaleByNumber$(mdd $a$, int $\mu$) is
 local int $i$;
 local mdd $s, u$;
 1 if $a.var = x_0$ return $\langle \mathbf{1}, \mathbf{0}\rangle$;
 2 if $InCache(C_{ScaleByNumber}, a, \mu, \langle s, u\rangle)$ then return $\langle s, n\rangle$;
 3 $s \leftarrow NewNode(a.var)$;                  • $s$ will encode paths that were scaled
 4 $u \leftarrow NewNode(a.var)$;            • $u$ will encode paths that could not be scaled
 5 foreach $i$ s.t $a[i] \neq \mathbf{0}$ do
 6   if $\mu$ divides $i$ then
 7     $\langle s[i/\mu], u[i]\rangle \leftarrow ScaleByNumber(a[i], \mu)$;      • $a[i]$ is divisible by $\mu$, scale it
 8   else
 9     $u[i] \leftarrow a[i]$;                                   • $a[i]$ cannot be scaled
 10 $s \leftarrow UniqueTableInsert(s)$;
 11 $u \leftarrow UniqueTableInsert(u)$;
 12 $CacheAdd(C_{ScaleByNumber}, a, \mu, \langle s, u\rangle)$;
 13 return $\langle s, u\rangle$;
```

**Fig. 8.** A symbolic algorithm to scale back p-semiflows

## 3 Experimental Results

Each variant of the *SymPSemiflows* algorithm is implemented in SMART [2] and all experiments are performed on a Pentium4 3.0GHz PC with 1.0GB of available memory, running CentoOS Linux 2.6.9. We use the following parametric models (for models with multiple parameters, the same value is used for all parameters; in our discussion, the parameter value is appended to the model name):

**trains:** a circular railway system with $u$ trains and $s$ rail trunks [6].
**slot:** a local area network protocol with $u$ nodes in the network [11].
**robin:** a round robin solution to the mutual exclusion among $u$ processes [8].
**aloha:** the ALOHA networking protocol, a precursor to Ethernet.
**classic:** a classic Petri net with $m$ transitions and $m$ stages of $u$ places each, used to demonstrate that the number of p-semiflows can be exponential, $u^m$ [3]. Fig. 9 shows this net for $m = 4$ and $u = 3$.
**classicX:** an extended version of the previous model, where the first input arc of each transition has cardinality 1, the next one has cardinality 2, and so on. The output arcs for each transition likewise have increasing cardinalities. The number of p-semiflows remains $u^m$.

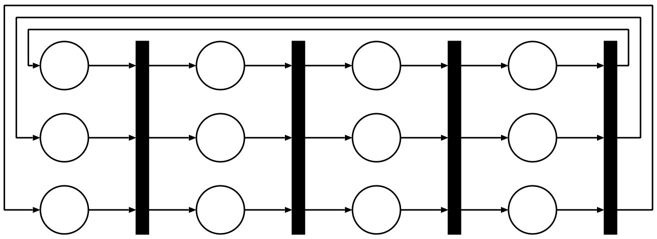


**Fig. 9.** Petri net *classic*

**mmarch:** a multi-threaded architecture with $u \times u$ processing nodes [7].
**phil:** the classic dining philosophers problem, with $u$ philosophers.
**power:** an electric power distribution system with $u$ electric generators. It uses arcs with cardinality greater than one and has a single p-semiflow.

Figs. 10 and 11 show the runtime results. For each model we report the number of transitions (trans), the number of p-semiflows, and the number of nodes and edges required to encode the final set of p-semiflows symbolically using an MDD. Then, for each of the four variants, "V1", "V2", "V3", and "V4", we report the peak amount of memory measured in MBytes (mem), the overall runtime in seconds (time), and the percentage of the overall runtime spent to perform computations related specifically to p-semiflow generation (PS), non-minimal support elimination (MS), and scaling back (SB), respectively. We also run each model using GreatSPN [1], which provides an explicit p-semiflow generation capability. The performance for GreatSPN is shown in the rows labeled "GS". The percentage of time spent performing p-semiflow generation, non-minimal support elimination, and scaling back is not shown for GreatSPN, as the tool does not report this detailed information. Finally, "om" means "out-of-memory" and "ot" means "out-of-time", i.e., the runtime exceeds eight hours.

## 3.1 Models Requiring Non-minimal P-Semiflow Elimination

We first analyze the models requiring non-minimal support elimination (Fig. 10). It is apparent that the results for V1 and V2 are nearly identical. Petri nets often have two features resulting in similar performance for these two variants: only arcs with cardinality one, and a sparse flow matrix. When the flow matrix is sparse and contains only entries with value $-1$, 1, or 0, the first column to be eliminated requires that only a single linear combination be performed because the MDD encoding of the flow matrix has at most a single positive entry, 1, and a single negative entry, $-1$, at its root. Performing this linear combination requires scalar multipliers equal to 1, so the scaled rows retain their original values. Due to the sparsity of the matrix, the MDD encoding the linear combination is likely to still retain the property of having (most) entries with value $-1$, 1, or 0, thus the next column to be eliminated also will most likely require that only a single linear combination be performed, and so on. For the models in Fig. 10 this

| model | trans | p-semiflows | nodes | edges | | mem | PS | MS | SB | time |
|---|---|---|---|---|---|---|---|---|---|---|
| trains10 | 100 | 37 | 642 | 678 | V1 | 6 | 5.02% | 94.95% | 0.03% | 4.70 |
| | | | | | V2 | 6 | 4.99% | 94.98% | 0.03% | 4.81 |
| | | | | | V3 | 7 | 4.76% | 95.17% | 0.07% | 4.05 |
| | | | | | V4 | om | | | | |
| | | | | | GS | 0.06 | - | - | - | 0.03 |
| trains15 | 255 | 99 | 3,533 | 3,620 | V1 | 19 | 4.86% | 95.13% | 0.01% | 210.60 |
| | | | | | V2 | 19 | 4.80% | 95.17% | 0.03% | 210.05 |
| | | | | | V3 | 21 | 4.98% | 95.01% | 0.01% | 216.30 |
| | | | | | V4 | om | | | | |
| | | | | | GS | 0.242 | - | - | - | 0.33 |
| slot20 | 160 | 42 | 1,597 | 1,798 | V1 | 58 | 0.20% | 99.79% | 0.01% | 57.44 |
| | | | | | V2 | 58 | 0.20% | 99.79% | 0.01% | 57.46 |
| | | | | | V3 | om | | | | |
| | | | | | V4 | 5 | 98.46% | 0.01% | 1.53% | 0.29 |
| | | | | | GS | 3 | - | - | - | 0.08 |
| slot2000 | 16,000 | 4,002 | 31,997 | 35,998 | V1 | om | | | | |
| | | | | | V2 | om | | | | |
| | | | | | V3 | om | | | | |
| | | | | | V4 | 242 | 99.64% | 0.01% | 0.36% | 126.12 |
| | | | | | GS | 876 | - | - | - | 189.02 |
| robin4 | 24 | 30 | 78 | 96 | V1 | 0.198 | 15.11% | 83.45% | 1.44% | 0.012 |
| | | | | | V2 | 0.198 | 15.06% | 83.51% | 1.42% | 0.012 |
| | | | | | V3 | 0.187 | 10.76% | 88.14% | 1.10% | 0.015 |
| | | | | | V4 | 26 | 99.9% | 0% | 0% | 11.06 |
| | | | | | GS | 3 | - | - | - | 0.01 |
| robin90 | 540 | $1.24 \times 10^{27}$ | 1,798 | 2,160 | V1 | 57 | 1.02% | 99.97% | 0.01% | 64.89 |
| | | | | | V2 | 57 | 0.36% | 99.62% | 0.01% | 64.81 |
| | | | | | V3 | om | | | | |
| | | | | | V4 | om | | | | |
| | | | | | GS | ot | | | | |
| aloha15 | 60 | 32,771 | 78 | 96 | V1 | 0.325 | 30.17% | 68.99% | 0.83% | 0.02 |
| | | | | | V2 | 0.326 | 30.31% | 68.88% | 0.81% | 0.02 |
| | | | | | V3 | 0.362 | 17.25% | 82.26% | 0.49% | 0.03 |
| | | | | | V4 | om | | | | |
| | | | | | GS | 4 | - | - | - | 33.20 |
| aloha100 | 400 | $1.27 \times 10^{30}$ | 503 | 606 | V1 | 12 | 43.42% | 56.43% | 0.14% | 1.00 |
| | | | | | V2 | 12 | 43.91% | 55.95% | 0.14% | 1.02 |
| | | | | | V3 | 14 | 6.21% | 93.76% | 0.03% | 4.96 |
| | | | | | V4 | om | | | | |
| | | | | | GS | ot | | | | |

**Fig. 10.** Models requiring minimal-support elimination.

process repeats throughout the entire process of p-semiflow generation. It is for this reason that the performance of V1 and V2 is identical for these models: V1 calls *MinSuppExt* after each linear combination for a column, but this is the same as calling *MinSuppExt* after all linear combinations for a column, as done in V2, if only one linear combination is performed per column.

V1 and V2 outperform V3, which typically requires more time and memory, running out of memory in some cases. The higher memory requirements arise from not refining the result of a linear combination before unioning it with $a$. V3 requires more time also because *MinSuppInt* performs non-minimal support elimination on the entire MDD *a* instead of on a smaller MDD produced as the result of a linear combination (for symbolic algorithms, higher memory requirements typically imply longer runtimes).

V4 performs poorly on all models except *slot*, because many non-minimal p-semiflows are found at each step of the algorithm. These extra p-semiflows greatly increase the time and memory requirements for V4, making it feasible for only the smallest model configurations. The only exception is *slot*, as it results in very few non-minimal p-semiflows (only about half of the generated p-semiflows are non-minimal); here, V4 actually outperforms V1, V2, and V3.

GreatSPN tends to perform better than our symbolic techniques only for models with relatively few p-semiflows, while our tool greatly outperforms GreatSPN for models with many p-semiflows. This is especially apparent with *slot* and *robin*. For the smaller version of these models, GreatSPN generates the results much more quickly. However, for the larger version of these models GreatSPN was either slower than at least one of our variants, or ran out of memory.

## 3.2 Models Not Requiring Non-minimal Support Elimination

Fig. 11 reports results for the models not requiring non-minimal support elimination. Among these, *classic*, *mmarch*, and *phil* do not use arcs with cardinality greater than one. As such, V1, V2, and V3 exhibit results similar to the models discussed in the previous section: V1 and V2 have nearly identical performance while V3 is somewhat less efficient. However, V4 really shines when applied to these three models. Non-minimal p-semiflow elimination is an expensive operation, and V1, V2, and V3 spend the majority of their runtime performing the useless task of looking for and attempting to eliminate non-minimal support p-semiflows when none such p-semiflows exist.

Models *power* and *classicX* contain arcs with cardinality greater than one. This added challenge does not significantly increase the overhead for *power*, which has a single p-semiflow: each variant exhibits comparable performance. The same does not hold for *classicX*, however. V1 performs very poorly on it because, for the parameters used in our tests, *classicX* ends up requiring hundreds of linear combination steps for each column in the flow matrix. This has a profound impact on performance as V1 is the only variant that performs elimination after each linear combination. V2, V3, and V4 have similar performance for *classicX*, because, with the higher cost of performing p-semiflow computation on a model having arc cardinalities greater than one, the minimization steps represent a relatively insignificant portion of the total computation time.

GreatSPN was unable to finish computation on *classic* and *classicX* because generating the large number of p-semiflows for these models is infeasible using explicit methods. Our tool was also better suited for generating the p-semiflows for the larger version of the *mmarch* model. However, GreatSPN outperformed our approach on *phil* and *power* due to their small number of p-semiflows.

| model | trans | p-semiflows | nodes | edges | | mem | PS | MS | SB | time |
|---|---|---|---|---|---|---|---|---|---|---|
| classic10 | 10 | $1 \times 10^{10}$ | 100 | 190 | V1 | 0.055 | 16.19% | 75.17% | 8.64% | 0.0027 |
| | | | | | V2 | 0.055 | 15.99% | 75.69% | 8.31% | 0.0028 |
| | | | | | V3 | 0.058 | 8.51% | 87.15% | 4.33% | 0.0054 |
| | | | | | V4 | 0.031 | 80.66% | 0.81% | 18.53% | 0.0014 |
| | | | | | GS | ot | | | | |
| classic250 | 250 | $3.05 \times 10^{599}$ | 62,500 | 124,750 | V1 | 613 | 0.58% | 99.00% | 0.37% | 54.20 |
| | | | | | V2 | 613 | 0.58% | 99.05% | 0.37% | 53.59 |
| | | | | | V3 | om | | | | |
| | | | | | V4 | 8 | 84.97% | 0.01% | 15.03% | 0.92 |
| | | | | | GS | ot | | | | |
| mmarch10 | 1,400 | 404 | 1,200 | 1,603 | V1 | 40 | 0.82% | 99.12% | 0.06% | 4.76 |
| | | | | | V2 | 40 | 0.83% | 99.11% | 0.06% | 4.79 |
| | | | | | V3 | om | | | | |
| | | | | | V4 | 2 | 95.85% | 0.05% | 4.93% | 0.0056 |
| | | | | | GS | 3 | - | - | - | 0.01 |
| mmarch20 | 5,600 | 1,604 | 4,800 | 6,403 | V1 | 198 | 3.84% | 96.15% | 0.01% | 122.15 |
| | | | | | V2 | 198 | 3.91% | 96.08% | 0.01% | 121.99 |
| | | | | | V3 | om | | | | |
| | | | | | V4 | 6 | 96.00% | 0% | 3.99% | 0.32 |
| | | | | | GS | 110 | - | - | - | 4.29 |
| phil30 | 120 | 90 | 239 | 328 | V1 | 11 | 2.35% | 97.59% | 0.06% | 1.04 |
| | | | | | V2 | 11 | 2.36% | 97.59% | 0.06% | 1.04 |
| | | | | | V3 | 11 | 0.43% | 99.53% | 0.04% | 1.61 |
| | | | | | V4 | 0.346 | 94.45% | 0.17% | 5.38% | 0.011 |
| | | | | | GS | 0.1 | - | - | - | 0.01 |
| phil100 | 400 | 300 | 799 | 1,098 | V1 | 50 | 0.19% | 99.79% | 0.01% | 30.90 |
| | | | | | V2 | 50 | 0.19% | 99.80% | 0.01% | 30.85 |
| | | | | | V3 | 50 | 0.13% | 99.86% | 0.01% | 47.64 |
| | | | | | V4 | 2 | 96.82% | 0.04% | 3.13% | 0.077 |
| | | | | | GS | 1 | - | - | - | 0.04 |
| power50 | 2,600 | 1 | 51 | 51 | V1 | 2 | 51.25% | 48.68% | 0.06% | 0.21 |
| | | | | | V2 | 2 | 51.17% | 48.76% | 0.06% | 0.21 |
| | | | | | V3 | 2 | 24.03% | 75.94% | 0.03% | 0.45 |
| | | | | | V4 | 2 | 99.84% | 0.02% | 0.13% | 0.11 |
| | | | | | GS | 0.6 | - | - | - | 0.08 |
| power100 | 10,200 | 1 | 101 | 101 | V1 | 14 | 51.39% | 48.58% | 0.01% | 2.05 |
| | | | | | V2 | 14 | 51.37% | 48.62% | 0.01% | 2.04 |
| | | | | | V3 | 14 | 23.51% | 76.49% | 0.01% | 4.50 |
| | | | | | V4 | 14 | 99.98% | 0.00% | 0.01% | 1.05 |
| | | | | | GS | 4 | - | - | - | 0.64 |
| classicX8 | 8 | $1.67 \times 10^{6}$ | 5,193 | 9,402 | V1 | 14 | 0.31% | 99.87% | 0.01% | 273.42 |
| | | | | | V2 | 2 | 68.60% | 17.26% | 13.94% | 0.27 |
| | | | | | V3 | 2 | 70.50% | 16.33% | 13.17% | 0.29 |
| | | | | | V4 | 4 | 86.47% | 0.01% | 13.53% | 0.28 |
| | | | | | GS | ot | | | | |
| classicX12 | 12 | $8.92 \times 10^{12}$ | 61,584 | 114,648 | V1 | om | | | | |
| | | | | | V2 | 18 | 84.43% | 9.54% | 6.02% | 29.73 |
| | | | | | V3 | 19 | 82.78% | 11.05% | 6.16% | 29.07 |
| | | | | | V4 | 18 | 93.71% | 0% | 6.29% | 28.31 |
| | | | | | GS | ot | | | | |

**Fig. 11.** Models not requiring minimal-support elimination

### 3.3 Scaling Back

For each variant of our algorithm for p-semiflow computation, we choose to scale back the p-semiflows at the very end. This avoids performing this operation multiple times; at least one pass of *SymScalePsemiflows* is required at the end of the p-semiflow computation anyway. For models where total runtime is greater than one second, less than 1% of time is spent scaling back. The only exception is *classicX*, which requires many passes of *SymScalePsemiflows*, because it results in rows with large numbers due to its many high-cardinality arcs. Even for this model, though, less than 14% of the time is spent scaling back the p-semiflows.

## 4 Summary and Conclusions

The symbolic method for p-semiflow computation we presented offers vast time and space improvements thanks to its use of ZMDDs for storage and computation. The most dramatic example comes from model *classic250*, for which it was able to generate $3.05 \times 10^{599}$ p-semiflows in under a second using only 8MBytes of memory. Our symbolic method benefits from the structure of many Petri net models that use only arcs with cardinality one. Such models often require a single pass of the *SymLinComb* algorithm per column.

Two types of models appear to have larger time and memory requirements using the proposed symbolic method. One includes models with a dense flow matrix, which cannot take as great an advantage of the properties of ZMDDs. The other includes models with arc cardinalities greater than one, as revealed by comparing the performance of the *classic* and *classicX* models. Despite the increased resource requirements for this second type of models, the symbolic method still greatly outperforms explicit approaches; for example, it can generate the $8.92 \times 10^{12}$ p-semiflows of the *classicX12* model in under 30 seconds.

We presented four variants of our algorithm, and at least one of either V2 or V4 was the most efficient (or very close to being the most efficient) for each model. V2 works best for models which add many non-minimal support p-semiflows at each step, while V4 works best for models where few non-minimal support p-semiflows are generated. It should then be possible to heuristically combine these two variants into a more resilient algorithm that starts with V4 and switches to V2 if the number of non-minimal support p-semiflows added at an iteration is above a certain threshold. Alternately, two workstations can be run in parallel, each computing the p-semiflows using either V2 or V4. For comparison, the explicit method implemented by GreatSPN tends to be more efficient for models with relatively few p-semiflows. However, for the majority of our parametric models, at least one variant of our symbolic method was able to outperform GreatSPN for large enough instances.

One topic left unexplored is that of a good variable order, an important issue in any decision diagram manipulation. Reordering the variables prior to p-semiflow computation (statically) or between column eliminations (dynamically) might reduce computation times and memory requirements. For example, a variable with only positive values or only negative values could be moved to the root so that no linear combinations or minimal-support computations would have to be performed for the column it represents. Due to the high cost of variable reordering and the potential growth rate of the MDD, it might be best to explore good static variable reordering heuristics first, in our future research.

## References

1. Chiola, G., Franceschinis, G., Gaeta, R., Ribaudo, M.: GreatSPN 1.7: Graphical Editor and Analyzer for Timed and Stochastic Petri Nets. Performance Evaluation 24, 47–68 (1995)
2. Ciardo, G., Jones, R.L., Miner, A.S., Siminiceanu, R.: Logical and stochastic modeling with SMART. Performance Evaluation 63, 578–608 (2006)
3. Colom, J.M., Silva, M.: Convex geometry and semiflows in P/T nets: A comparative study of algorithms for the computation of minimal p-semiflows. In: Proc. International Conference on Applications and Theory of Petri nets, pp. 74–95 (1989)
4. Colom, J., Silva, M.: Improving the linearly based characterization of P/T nets. In: Rozenberg, G. (ed.) APN 1990. LNCS, vol. 483, pp. 113–145. Springer, Heidelberg (1991)
5. Farkas, J.: Theorie der einfachen ungleichungen. Journal für die reine und andgewandte Mathematik 124, 1–27 (1902)
6. Genrich, H.: Predicate/transition nets. In: Brauer, W., Reisig, W., Rozenberg, G. (eds.) APN 1986. LNCS, vol. 254, pp. 207–247. Springer, Heidelberg (1987)
7. Govindarajan, R., Suciu, F., Zuberek, W.M.: Timed Petri net models of multithreaded multiprocessor architectures. In: Proc. Petri Nets and Performance Models, pp. 153–162 (1997)
8. Graf, S., Steffen, B., Lüttgen, G.: Compositional minimisation of finite state systems using interface specifications. Journal of Formal Aspects of Computing 8(5), 607–616 (1996)
9. Kimura, S., Clarke, E.M.: A parallel algorithm for constructing binary decision diagrams. In: Proc. International Conference on Computer Design, pp. 220–223. IEEE Comp. Soc. Press, Los Alamitos (1990)
10. Minato, S.-I.: Zero-suppressed BDDs and their applications. Software Tools for Technology Transfer 3, 156–170 (2001)
11. Pastor, E., Roig, O., Cortadella, J., Badia, R.M.: Petri net analysis using boolean manipulation. In: Valette, R. (ed.) ICATPN 1994. LNCS, vol. 815, pp. 416–435. Springer, Heidelberg (1994)
12. Wan, M., Ciardo, G.: Symbolic state-space generation of asynchronous systems using extensible decision diagrams. In: Nielsen, M., et al. (eds.) SOFSEM 2009. LNCS, vol. 5404, pp. 582–594. Springer, Heidelberg (2009)

# Orthomodular Lattices in Occurrence Nets

Luca Bernardinello, Lucia Pomello, and Stefania Rombolà

Dipartimento di informatica, sistemistica e comunicazione
Università degli studi di Milano–Bicocca
viale Sarca 336–U14, Milano
luca.bernardinello@unimib.it

**Abstract.** In this paper, we study partially ordered structures associated to occurrence nets. An occurrence net is endowed with a symmetric, but in general non transitive, concurrency relation. By applying known techniques in lattice theory, from any such relation one can derive a closure operator, and then an orthocomplemented lattice. We prove that, for a general class of occurrence nets, those lattices, formed by closed subsets of net elements, are orthomodular. A similar result was shown starting from a simultaneity relation defined, in the context of special relativity theory, on Minkowski spacetime. We characterize the closed sets, and study several properties of lattices derived from occurrence nets; in particular we focus on properties related to K-density. We briefly discuss some variants of the construction, showing that, if we discard conditions, and only keep the partial order on events, the corresponding lattice is not, in general, orthomodular.

## 1 Introduction

In this paper we present some theoretical results in the frame of the partial order theory of concurrency. In particular, we focus on occurrence nets, the basic model in net theory for modelling non-sequential processes (see [1] and [2]).

The origin of this work lies on our interest for the relations between Petri net theory and physics; here we consider in particular the special relativity theory.

Since the beginning of his work on nets, Petri has stressed the role that physics should play in their formal and conceptual development. We can identify a double role; on one hand, any theory of information flow, of processes, and of general systems, must take into account the constraints imposed by physical laws; on the other hand, Petri has always considered physical theories, in their mathematical form, as a sort of model to which a theory of information flows should be inspired.

A crucial difference between the standard physical theories and the framework on which Petri develops his own theory comes from the use of the continuum as the underlying model in physics. Petri proposes a combinatorial representation of a spacetime, in which notions corresponding to the relativistic concepts of world line and causal cone can be defined in occurrence nets by means of the concurrency and causal dependence relations.

The idea of a discrete, combinatorial model for physical phenomena has been developed by numerous authors, and in many different directions. The most successful attempt seems to be that based on cellular automata, with a huge literature, and the ambitious project brought forward by Wolfram (see [3]). Other interesting, and sometimes controversial, approaches were proposed by Fredkin (see his web site on Digital Philosophy) and Zuse, who also cooperated with Petri, (see [4], [5]). More recently, several authors have proposed so-called *causal sets* as a discrete model of spacetime (see, for instance, [6]); a causal set is a partial order of events. See also [7] for a different approach to "discrete physics", based on notions of category theory.

A more specific inspiration for the present paper comes from a couple of papers studying orthomodular lattices derived from Minkowski spacetime ([8]) and more general spacetimes ([9]). In these papers, a lattice of subsets of points of a spacetime is constructed via a closure operator defined on the basis of the *spacelike* and *timelike* relations, which are analogous to concurrency and causal dependence. We apply the same construction, but starting from an occurrence net and the associated concurrency relation, and obtain a similar result: for a general class of occurrence nets, the lattice of closed sets is orthomodular. This result suggests an analogy between the concurrency relation in occurrence nets and the simultaneity relation on Minkowski spacetime. In this sense, our work can be seen as a continuation of Petri's work on a combinatorial representation of spacetime.

In a different context, lattices associated to occurrence nets have been studied also by Fernandez, Merceron, Thiagarajan, and others, who take maximal concurrent subsets of events and conditions as elements of the derived lattices (see, for instance, [10]).

The paper is structured as follows. Section 2 collects basic definitions related to occurrence nets, closure operators, orthomodular posets and lattices. In Section 3, causally closed sets in occurrence nets are defined, and then characterized by a set of structural properties. Then, in Section 4, the global structure of the lattice formed by causally closed sets is analyzed; here we show that, for a general class of occurrence nets, the lattice is orthomodular. In a specific subsection we look at properties of the lattice of causally closed sets which are related to K-density. In 4.2, we apply the same construction to elementary event structures ([11]), and show that, in this case, the lattice of closed sets is not guaranteed to be orthomodular. Finally, in Section 5, we briefly comment on the main results, and suggest further developments.

## 2 Preliminary Definitions

In this section, we recall the basic definitions needed in the following.

### 2.1 Occurrence Nets

**Definition 1.** *A* net *is a triple* $N = (B, E, F)$*, where* $B$ *and* $E$ *are countable sets,* $F \subseteq (B \times E) \cup (E \times B)$*, and*

(i) $B \cap E = \emptyset$
(ii) $\mathrm{dom}(F) \cup \mathrm{ran}(F) = B \cup E$

The elements of $B$ are called *local states* or *conditions*, the elements of $E$ *local changes of state* or *events*, and $F$ is called the *flow relation*. We will use the standard graphical notation for nets.

For each $x \in B \cup E$, define ${}^\bullet x = \{y \in B \cup E \mid (y,x) \in F\}$, $x^\bullet = \{y \in B \cup E \mid (x,y) \in F\}$. For $e \in E$, an element $b \in B$ is a *precondition* of $e$ if $b \in {}^\bullet e$; it is a *postcondition* of $e$ if $b \in e^\bullet$. A net $N = (B,E,F)$ is *simple* iff for each $x, y \in B \cup E$: $({}^\bullet x = {}^\bullet y$ and $x^\bullet = y^\bullet) \Rightarrow x = y$.

Occurrence nets are a special class of nets used to model non-sequential processes ([1], [2]). Note that they are called *causal nets* in [11].

**Definition 2.** *A net* $N = (B,E,F)$ *is an* occurrence net *iff*

(i) $\forall b \in B : |{}^\bullet b| \leq 1 \ \wedge \ |b^\bullet| \leq 1$ *and*
(ii) $\forall x, y \in B \cup E : (x,y) \in F^+ \Rightarrow (y,x) \notin F^+$.

Hence an occurrence net contains neither conflicts nor cycles.

Because of Definition 2(ii), the structure $(X, \sqsubseteq)$ derived from an occurrence net $N$ by putting $X = B \cup E$ and $\sqsubseteq = F^*$ is a partially ordered set (shortly *poset*). We will use $\sqsubset$ to denote the associated strict partial order.

Given a partial order relation $\leq$ on a set A, we can derive the relations $li = \leq \cup \geq$, and $co = (A \times A) \setminus li$. We will be interested in such relations derived from $(X, \sqsubseteq)$. In such case, intuitively, $x \ li \ y$ means that $x$ and $y$ are connected by a causal relation, and $x \ co \ y$ means that $x$ and $y$ are causally independent. The relations $li$ and $co$ are symmetric and not transitive. Note that $li$ is a reflexive relation, while $co$ is irreflexive. Given an element $x \in X$ and a set $S \subseteq X$, we write $x \ co \ S$ if $\forall y \in S : x \ co \ y$. Moreover, given two sets $S_1 \subseteq X$ and $S_2 \subseteq X$, we write $S_1 \ co \ S_2$ if $\forall x \in S_1, \forall y \in S_2 : x \ co \ y$. In the following we will use $x \, co \, y$ or $(x,y) \in co$ indifferently, and similarly for $li$.

For each element $x$ of $X$, we can now define the *causal cone* at $x$ by:

$$\mathrm{Cone}(x) = \{y \in X \mid x \ li \ y\}.$$

For each $x \in X$ we denote by:

$$F^-(x) = \{y \in X \mid y \sqsubset x\} \quad \text{and} \quad F^+(x) = \{z \in X \mid x \sqsubset z\}$$

the *past* and the *future* of $x$, respectively. By generalizing to subsets $S$ of $X$, we denote the *past* and the *future* of $S$ by

$$F^-(S) = \{x \in X \mid x \notin S, \exists y \in S : x \in F^-(y)\} \text{ and}$$
$$F^+(S) = \{x \in X \mid x \notin S, \exists y \in S : x \in F^+(y)\}.$$

Note that an element $x$ belongs neither to its future nor to its past. From the $li$ and $co$ relations one can define *cuts* and *lines* of a poset $\mathcal{A} = (A, \leq)$:

$$\mathrm{Cuts}(\mathcal{A}) = \{c \subseteq A \mid c \text{ is a maximal clique of } co \cup id_A\};$$
$$\mathrm{Lines}(\mathcal{A}) = \{ \ l \subseteq A \mid l \text{ is a maximal clique of } li\}.$$

Given an occurrence net $N = (B, E, F)$, we will denote by $\text{Cuts}(N)$ and $\text{Lines}(N)$, respectively, the set of cuts and the set of lines of the poset associated to $N$. We will always assume the Axiom of Choice, so that any clique of *co* and of *li* can be extended to a maximal clique.

The notion of *K-density* [2] was introduced in order to formalize a property which intuitively should hold for posets corresponding to non-sequential processes which are actually feasible. K-density is based on the idea of interpreting cuts as (global) states and lines as sequential subprocesses. K-density postulates that every occurrence of a subprocess must be in a specific state.

**Definition 3.** $\mathcal{A} = (A, \leq)$ *is* K-dense $\Leftrightarrow \forall c \in \text{Cuts}(\mathcal{A}), \forall l \in \text{Lines}(\mathcal{A}) : c \cap l \neq \emptyset$.

An occurrence net $N$ is K-dense if its associated poset is K-dense.

For posets derived from occurrence nets, K-density can be characterized by the absence of the substructures shown in Fig. 1 [2]. In order to formalize this fact we need to say that a poset $(A', \leq')$ is embeddable into $(A, \leq)$ iff there exists an injection $\gamma : A' \to A$ such that $\forall x, y \in A' : x \leq' y \Leftrightarrow \gamma(x) \leq \gamma(y)$.

**Proposition 1.** *Let* $(X, \sqsubseteq)$ *be the poset associated to an occurrence net* $N = (B, E, F)$, $X = (B \cup E)$. *If none of the posets shown in* Fig. 1 *is embeddable into* $(X, \sqsubseteq)$, *then* $(X, \sqsubseteq)$ *is K-dense.*

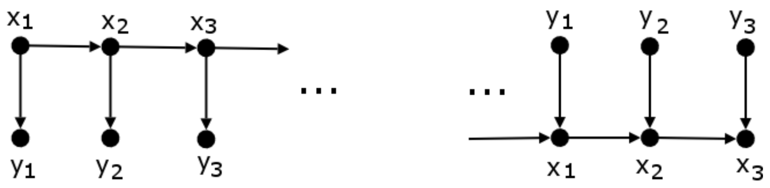


**Fig. 1.** Posets which are not K-dense

In the following examples, we often use a specific, infinite and regular, occurrence net, which we call *Petri grid*, proposed by Petri ([12]), as a discrete representation of bidimensional spacetime. We have chosen the following definition of Petri grid, among all the possible ones, since, with respect to the event coordinates of the net, it preserves the relations which hold on the coordinates of points in a bidimensional Minkowski spacetime.

**Definition 4.** *Let* $N_g = (B_g, E_g, F_g)$ *be an infinite occurrence net, where* $E_g = \{(x, y) \mid x, y \in \mathbb{Z}, x + y \text{ is even}\}$, $B_g = \{(e_1, e_2) \mid e_1, e_2 \in E_g,\ e_1 = (x, y),\ e_2 = (x \pm 1, y + 1)\}$, *and* $F_g$ *is such that* $\forall e_1, e_2 \in E_g$, *if* $(e_1, e_2) \in B_g$ *then*

(i) $(e_1, (e_1, e_2)) \in F_g$,
(ii) $((e_1, e_2), e_2) \in F_g$.

The Petri grid $N_g = (B_g, E_g, F_g)$ is not K-dense. The violation of K-density is witnessed, e.g., by the line $l = \{(x, y) \mid (x, y) \in E_g, x = y\} \cup \{(e_1, e_2) \mid (e_1, e_2) \in B_g, e_1 = (x, y), e_2 = (x + 1, y + 1), x = y\}$ and the cut $c = \{(e_1, e_2) \mid (e_1, e_2) \in B_g, e_1 = (x, y),\ e_2 = (x - 1, y + 1), x = y\}$ (see the left side of Fig. 2, where the blackened elements form $l$, while the dotted line joins the elements of $c$). On the right side of Fig. 2 we show another example of K-density violation.

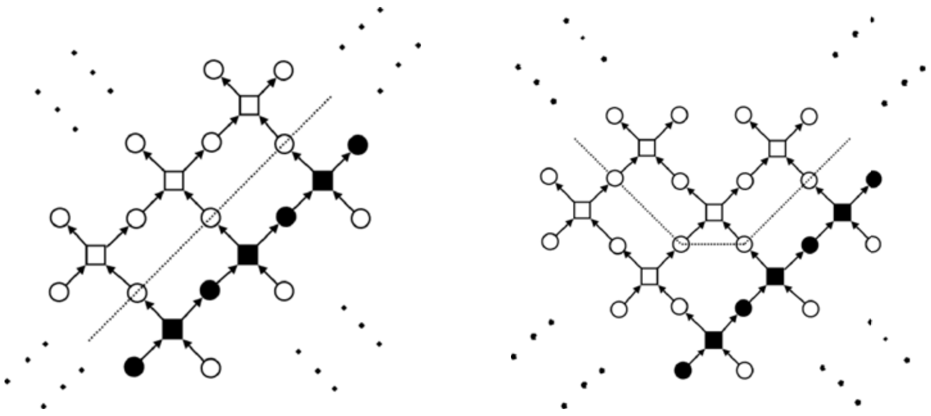


**Fig. 2.** Examples illustrating that the net $N_g$ is not K-dense

## 2.2 Orthomodular Posets and Lattices

In this section we recall the basic definitions related to orthomodular posets and lattices.

**Definition 5.** *An* orthocomplemented poset $P = \langle P, \leq, 0, 1, (\,.\,)' \rangle$ *is a partially ordered set* $\langle P, \leq \rangle$*, equipped with a minimum and a maximum element, respectively denoted by 0 and 1, and with a map* $(\,.\,)' : P \to P$*, such that the following conditions are verified (where* $\vee$ *and* $\wedge$ *denote, respectively, the least upper bound and the greatest lower bound with respect to* $\leq$*, when they exist):* $\forall x, y \in P$

(i) $(x')' = x$;

(ii) $x \leq y \Rightarrow y' \leq x'$;

(iii) $x \wedge x' = 0$ and $x \vee x' = 1$.

The map $(\,.\,)' : P \to P$ is called an *orthocomplementation* in $P$. In an orthocomplemented poset, $\wedge$ and $\vee$, when they exist, are not independent: in fact, the so-called De Morgan laws hold: $(x \vee y)' = x' \wedge y'$, $\quad (x \wedge y)' = x' \vee y'$. In the following, we will sometimes use *meet* and *join* to denote, respectively, $\wedge$ and $\vee$. Meet and join can be extended to families of elements in the obvious way, denoted by $\bigwedge$ and $\bigvee$. In an orthocomplemented poset the notions of orthogonality and compatibility can be introduced. Two elements $x, y \in P$ are called *orthogonal*,

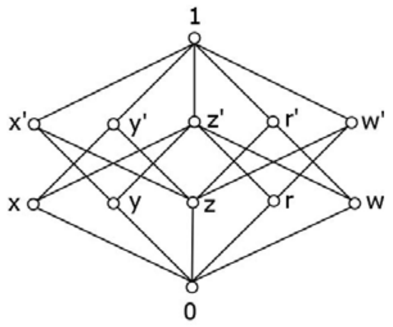


**Fig. 3.** A finite regular orthomodular poset

denoted $x \perp y$, iff $x \leq y'$. Two elements $x, y \in P$ are called *compatible*, denoted $x \$ y$, iff $\exists\, x_0, y_0, z \in P$: $(x_0 \perp y_0 \perp z \perp x_0$ and $x = x_0 \vee z$ and $y = y_0 \vee z)$.

From the previous definitions it follows that [13]:

(i) $x \perp x'$; (ii) $x \perp y \Rightarrow x \$ y$; (iii) $x \leq y \Rightarrow x \$ y$;
(iv) $x \$ y \Rightarrow (x \vee y \in P$ and $x \wedge y \in P)$; (v) $x \$ y \Leftrightarrow x \$ y' \Leftrightarrow x' \$ y'$.

A poset $P$ is called *orthocomplete* when it is orthocomplemented and every pairwise orthogonal countable subset of $P$ has a least upper bound.

A lattice $\mathcal{L}$ is a poset in which for any pair of elements meet and join always exist. A lattice $\mathcal{L}$ is *complete* when the meet and the join of any subset of $\mathcal{L}$ always exist.

**Definition 6.** *[14] An* orthomodular poset $P = \langle P, \leq, 0, 1, (\,.\,)' \rangle$ *is an orthocomplete poset which satisfies the condition:*

$$x \leq y \Rightarrow y = x \vee (y \wedge x')$$

which is usually referred to as the orthomodular law; we will sometimes use the equivalent statement $x \leq y \Rightarrow x = y \wedge (x \vee y')$.

The orthomodular law is weaker than the distributive one. A lattice $\mathcal{L}$ is called *distributive* if and only if $\forall x, y, z \in \mathcal{L}$ the equalities $x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$, $x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$ hold. A distributive lattice is orthomodular. Orthocomplemented distributive lattices are generally called Boolean algebras. Orthomodular posets and lattices can therefore be considered as a generalization of Boolean algebras and have been studied as algebraic models for quantum logic [13].

An orthomodular poset $P = \langle P, \leq, 0, 1, (\,.\,)' \rangle$ is *regular* (or *coherent*) when $\forall x, y, z \in P$ such that $x \$ y \$ z \$ x$, it holds $(x \vee y) \$ z$. Any orthomodular lattice $\mathcal{L}$ is regular [13]. Any regular orthomodular poset can be seen as a family of partially overlapping Boolean algebras. Two elements are compatible if there is a Boolean subalgebra which contains both of them.

*Example 1.* Fig. 3 shows a finite regular orthomodular poset. Fig. 4 shows an infinite regular orthomodular poset.

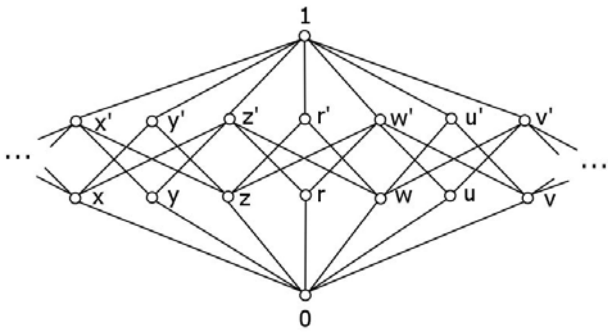


**Fig. 4.** An infinite regular orthomodular poset

## 2.3 Closure Operators

Useful references for this section are [15] and [16].

**Definition 7.** *Let $X$ be a set and $\mathbb{P}(X)$ the powerset of $X$. A map $\mathcal{C} : \mathbb{P}(X) \to \mathbb{P}(X)$ is a* closure operator *on $X$ if, for all $A, B \subseteq X$,*

$$\begin{aligned} &\text{(i)} \quad A \subseteq \mathcal{C}(A), \\ &\text{(ii)} \quad A \subseteq B \Rightarrow \mathcal{C}(A) \subseteq \mathcal{C}(B), \\ &\text{(iii)} \quad \mathcal{C}(\mathcal{C}(A)) = \mathcal{C}(A). \end{aligned}$$

Note that, with this definition, $\mathcal{C}$ is not a topological closure operator. A subset $A$ of $X$ is called *closed* with respect to $\mathcal{C}$ if $\mathcal{C}(A) = A$. If $\mathcal{C}$ is a closure operator on a set $X$, the family $L_{\mathcal{C}} = \{A \subseteq X \mid \mathcal{C}(A) = A\}$ of closed subsets of $X$ forms a complete lattice, when ordered by inclusion, in which

$$\bigwedge\{A_i : i \in I\} = \bigcap_{i\in I} A_i, \quad \bigvee\{A_i : i \in I\} = \mathcal{C}(\bigcup_{i\in I} A_i).$$

The proof of this statement can be found in [15].

We now describe a construction from binary relations to closure operators. Let $X$ be a set of elements, and $\alpha \subseteq X \times X$ be a symmetric relation. Given $A \subseteq X$ we can define an operator $(.)^\perp$ on the powerset of $X$

$$A^\perp = \{x \in X \mid \forall y \in A : (x, y) \in \alpha\}.$$

By applying twice the operator $(\,.\,)^\perp$, we get a new operator $C(\,.\,) = (\,.\,)^{\perp\perp}$. The map $C$ on the powerset of $X$ is a closure operator on $X$ [15]. A subset $A$ of $X$ is called *closed* with respect to $(\,.\,)^{\perp\perp}$, if $A = A^{\perp\perp}$. The family $L(X)$ of all closed sets of $X$, ordered by set inclusion, is then a complete lattice [15].

When $\alpha$ is also irreflexive, the operator $(\,.\,)^{\perp}$, applied to elements of $L(X)$, is an orthocomplementation; the structure $\mathcal{L}(X) = \langle L(X), \subseteq, \emptyset, X, (\,.\,)^{\perp} \rangle$ then forms an orthocomplemented complete lattice [15].

A closure operator $\mathcal{C}(.)$ on a set $X$ is called *algebraic* if, for all $A \subseteq X$,

$$\mathcal{C}(A) = \bigcup\{\mathcal{C}(B) \mid B \subseteq A \text{ and } B \text{ is finite}\}.$$

If $\mathcal{C}(.)$ is an algebraic closure operator on a set $X$ and $\mathcal{L}_{\mathcal{C}}$ the associated complete lattice, then $\mathcal{L}_{\mathcal{C}}$ is an algebraic lattice ([16]).

## 3 Causally Closed Sets Induced by the Concurrency Relation in Occurrence Nets

In this section we give the definition of a closure operator built starting from the *co* relation in occurrence nets, and study the characterization of causally closed sets induced by this operator.

We work on a general class of occurrence nets $N = (B, E, F)$ such that $N$ is simple and

(i) $\forall e \in E : 1 \leq |{}^{\bullet}e| < \infty \;\wedge\; 1 \leq |e^{\bullet}| < \infty$,
(ii) $\forall e_1, e_2 \in E : |[e_1, e_2]| < \infty$, where $[e_1, e_2] = \{x \in B \cup E \mid e_1 \sqsubseteq x \sqsubseteq e_2\}$.

Condition (i) is stronger than "degree finiteness" since each event must have at least one precondition and one postcondition, while condition (ii) is "interval finiteness" [2]. These two properties seem to be natural requirements for models of real processes.

Let $N = (B, E, F)$ be an occurrence net satisfying (i) and (ii) above. We can define an operator on subsets of $X = (B \cup E)$, which corresponds to an orthocomplementation, since *co* is irreflexive, and give a characterization of the causally closed sets generated by this operator.

**Definition 8.** *Let $S \subseteq X$, then*

*(i)* $S^{\perp} = \{x \in X \mid \forall y \in S : x \; co \; y\}$ *is the orthocomplement of $S$;*
*(ii) if $S = (S^{\perp})^{\perp}$, then $S$ is a causally closed set of $N$.*

The set $S^{\perp}$ contains the elements of $X$ which are not in causal relation with any element of $S$. Obviously, $S \cap S^{\perp} = \emptyset$ for any $S \subseteq X$. Notice that causally closed sets are not related to closed sets as defined by Petri for nets ([17]).

A typical example of a closed set in Petri grid is shown in Figure 5, together with the corresponding orthocomplement.

We now introduce some properties of causally closed sets of $N$. In the following, we denote by $b$ and $e$, respectively, an element of $B$ and an element of $E$, and we sometimes denote $(S^{\perp})^{\perp}$ by $S^{\perp\perp}$. First, we present a relation between a causally closed set $S$ and its orthocomplement $S^{\perp}$.

**Proposition 2.** *Let $S = S^{\perp\perp}$. Then $F^{-}(S) = F^{-}(S^{\perp})$ and $F^{+}(S) = F^{+}(S^{\perp})$.*

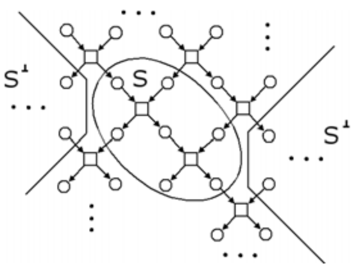


**Fig. 5.** A closed set $S$ and its orthocomplement $S^\perp$

*Proof.* Let $S = S^{\perp\perp}$, and $x \in F^-(S)$. Then $\exists y \in S : x \sqsubset y$. Assume that $x \notin F^-(S^\perp)$. Let us consider two cases:

1. $x$ *co* $S^\perp$. Since $S = S^{\perp\perp}$, $x \in S$ in contradiction with the assumption $x \in F^-(S)$.
2. $x \in F^+(S^\perp)$. Then $\exists y_1 \in S^\perp$, $y_1 \sqsubset x$; from $x \sqsubset y$, it then follows $y_1 \sqsubset y$, which is a contradiction.

Therefore $x \in F^-(S^\perp)$ and $F^-(S) \subseteq F^-(S^\perp)$. Since $(S^\perp)^\perp = S$, by the same argument $F^-(S) \supseteq F^-(S^\perp)$. The proof for $x \in F^+(S)$ is analogous. □

Proposition 2 states that a causally closed set $S$ and its orthocomplement $S^\perp$ share the past and the future.

We now show that if a causally closed set $S$ contains an event $e$ then it contains its preconditions and postconditions.

**Proposition 3.** *For each $e \in E$, for each $x \in X$*

$$(x, e) \in co \Leftrightarrow (\forall y \in {}^\bullet e : (x, y) \in co) \Leftrightarrow (\forall z \in e^\bullet : (x, z) \in co).$$

*Proof.* Assume $(x, e) \in co$. Let $b \in {}^\bullet e$. $(x, b) \in co$ follows immediately from the fact that $e$ is the unique element in $b^\bullet$.

Assume now $(x, b) \in co$ for all $b \in {}^\bullet e$. Any path from $x$ to $e$ should pass through one precondition of $e$; hence there can be no such path. On the other hand, if $e \sqsubset x$, then $b \sqsubset x$ for all $b \in {}^\bullet e$; hence $(x, e) \in co$. We have thus shown $(x, e) \in co \Leftrightarrow \forall b \in {}^\bullet e : (x, b) \in co$.

By the same argument, inverting the order relation, one shows $(x, e) \in co \Leftrightarrow \forall b \in e^\bullet : (x, b) \in co$. □

**Corollary 1.** *Let $S = S^{\perp\perp}$ and $e \in E$. We have:*

(i) $e \in S \Rightarrow {}^\bullet e \cup e^\bullet \subseteq S$,
(ii) ${}^\bullet e \subseteq S \Rightarrow e \in S$,
(iii) $e^\bullet \subseteq S \Rightarrow e \in S$.

The next proposition shows that any causally closed set $S$ is convex.

**Proposition 4.** *Let $b_1, b_2 \in B$ such that $b_1 \sqsubset b_2$; if $b_1, b_2 \in S = S^{\perp\perp}$ then $\forall x \in B \cup E : x \in [b_1, b_2] \Rightarrow x \in S$.*

*Proof.* Let $y \in S^{\perp}$; then $y$ *co* $b_1$ and $y$ *co* $b_2$. Let $x \in [b_1, b_2]$. If $y \in F^-(x)$ then $y \in F^-(b_2)$; so $y \sqsubset b_2$, which is a contradiction. If $y \in F^+(x)$ then $y \in F^+(b_1)$; so $y \sqsupset b_1$, which again is a contradiction. Therefore $y$ *co* $x$ for any $x \in [b_1, b_2]$, and $x \in S^{\perp\perp} = S$. □

Since any condition has at most one pre-event and at most one post-event, from Proposition 4 it follows immediately that, for any $x_1, x_2 \in S$, the interval $[x_1, x_2]$ is contained in $S$ when $S$ is closed.

From Corollary 1 and Proposition 4 it follows that the causally closed sets of $N$ are *convex* sets, namely that if a causally closed set $S$ contains two causally connected elements then it contains any element between them.

We now give a complete characterization of causally closed sets. In order to do this we need some preliminary definitions.

**Definition 9.** *Let $S \subseteq X$. Then*

(i) $Ad_f(S) = \{e \in E \mid e \notin S, {}^{\bullet}e \cap S \neq \emptyset, \forall x \in F^-(e) : [x, e] \cap S = \emptyset \Rightarrow F^-(x) \cap S \neq \emptyset\}$,
(ii) $Ad_p(S) = \{e \in E \mid e \notin S, e^{\bullet} \cap S \neq \emptyset, \forall x \in F^+(e) : [e, x] \cap S = \emptyset \Rightarrow F^+(x) \cap S \neq \emptyset\}$.

We denote by $Ad(S) = Ad_f(S) \cup Ad_p(S)$ the set of *adjacent events* of $S$. Intuitively, events adjacent to a set $S$ are events such that, going backwards along their past (or going along their future), outside $S$, we cannot find an element whose past (or future) is independent from $S$. Note that if a set $S \subseteq X$ contains all preconditions or all postconditions of an event $e$ outside $S$ then $e$ is an adjacent event of $S$.

*Example 2.* An example of an adjacent event is shown on the left side of Fig. 6, where $e_1 \in Ad_f(S_1)$. The event $e_2$ on the right side of Fig. 6 is not adjacent to $S_2$ since $x \in F^-(e_2)$, $[x, e_2]$ is disjoint from $S_2$, but $F^-(x) \cap S_2 = \emptyset$. Intuitively, every element in the past of $e_1$ is in the future of some element in $S_1$, while $x$, which is in the past of $e_2$, is causally independent from any element in $S_2$.

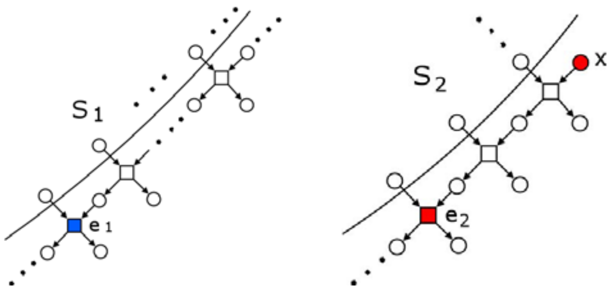


**Fig. 6.** The event $e_1$ is adjacent to $S_1$; $e_2$ is not adjacent to $S_2$

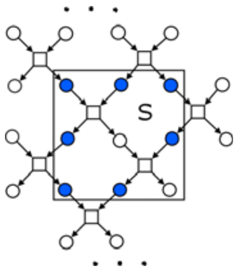


**Fig. 7.** An example of frontier of a set $S$

**Definition 10.** *Let $S \subseteq X$. Then the* frontier *of $S$ is the set*

$$\mu(S) = \{x \in S \mid \exists y \notin S, x\ F\ y \text{ or } y\ F\ x\}.$$

Intuitively, the frontier of $S$ is the set of elements of $S$ which are directly linked to the outside (see Fig. 7 where the frontier of $S$ is graphically represented by the set of grey elements). Note that if $S$ is a causally closed set then, by Corollary 1, $\mu(S) \subseteq B$.

We are now ready for our characterization of causally closed sets of $N$.

**Proposition 5.** *Let $S \subseteq X$. Then*

$$(S \text{ is a convex set, } Ad(S) = \emptyset, \text{ and } \mu(S) \subseteq B) \Leftrightarrow S = S^{\perp\perp}.$$

*Proof.* We first show the "$\Rightarrow$" implication (see the left side of Fig. 8). Let $S \subseteq X$ be a convex set, with $Ad(S) = \emptyset$ and $\mu(S) \subseteq B$. We proceed by contradiction. Take $x \in S^{\perp\perp} \setminus S$. We will show that $x\ li\ w$ for some $w \in S^{\perp}$.

From the hypothesis, $x \notin S^{\perp}$. Suppose $x \in F^{+}(S)$; then $\exists b \in \mu(S), \exists e \in E$ : $e \notin S,\ b \in {}^{\bullet}e,\ e \sqsubseteq x$. Since $Ad(S) = \emptyset$, $e$ cannot be adjacent to $S$; hence

$$\exists y \in F^{-}(e) : [y, e] \cap S = \emptyset \text{ and } F^{-}(y) \cap S = \emptyset.$$

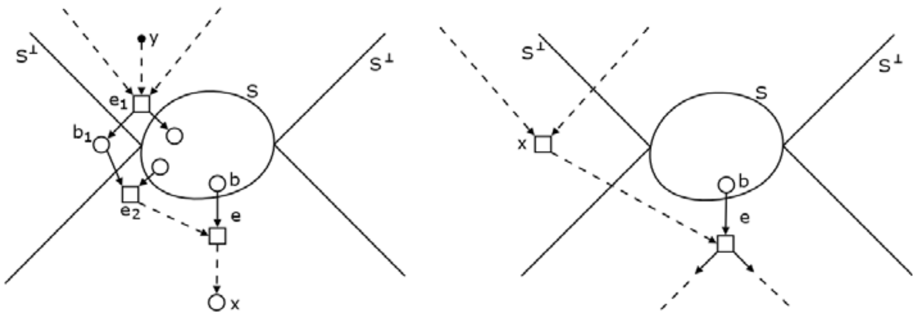


**Fig. 8.** Characterization of causally closed sets of $N$

Choose an arbitrary path from $y$ to $e$, and let $e_1$ be the last event on this path for which $F^-(e_1) \cap S = \emptyset$. If $e_1$ *co* $S$, then $e_1 \in S^\perp$; since $e_1 \in F^-(x)$, and $x \in S^{\perp\perp}$ we have a contradiction.

Assume then that $e_1 \notin S^\perp$. Since $F^-(e_1) \cap S = \emptyset$, $e_1 \notin F^+(S)$. If $e_1 \in F^-(S)$, take a condition $b_1$ such that $b_1 \in e_1^\bullet$, $b_1 \in [e_1, e]$. Since $\{e_1\} = {}^\bullet b_1$ and $e_1 \notin F^+(S)$ it follows that $b_1 \notin F^+(S)$. Assume that $b_1 \in F^-(S)$. Let $\{e_2\} = b_1^\bullet$, then $e_2 \in F^-(S)$. Moreover $e_2 \in F^-(e)$, $e_2 \in [e_1, e]$ and $F^-(e_2) \cap S \neq \emptyset$, because of the choice of $e_1$. Hence $e_2 \in F^+(S)$. But this is in contradiction with the assumption that $S$ is a convex set. Therefore $e_2 \notin F^-(S)$ from which it follows that $b_1 \notin F^-(S)$. Hence $b_1$ *co* $S$, that implies $b_1 \in S^\perp$. But $b_1 \in F^-(x)$, in contradiction with the assumption $x \in S^{\perp\perp}$.

For $x \in F^-(S)$ the proof is analogous. Therefore $S = S^{\perp\perp}$.

We now show the reverse implication (see the right side of Fig. 8). Let $S = S^{\perp\perp}$. From Corollary 1 and Proposition 4 it follows that $S$ is a convex set. Corollary 1(i) implies that $\mu(S) \subseteq B$. Suppose $Ad(S) \neq \emptyset$. Then $\exists e \in E : e \in Ad(S)$. Suppose $e \in Ad_f(S)$; then $\exists b \in {}^\bullet e : b \in S$. Since $S = S^{\perp\perp}$ and $e \notin S$, $\exists x \in S^\perp : x\ li\ e$. If $x \in F^+(e)$ we have $x\ li\ b$ which is a contradiction. Suppose $x \in F^-(e)$; since $x \in S^\perp$, $[x, e] \cap S = \emptyset$ and $F^-(x) \cap S = \emptyset$. By Definition 9 $e \notin Ad_f(S)$, in contradiction with the assumption that $e \in Ad_f(S)$.

For $e \in Ad_p(S)$ the proof is analogous. Therefore $Ad(S) = \emptyset$. □

## 4 The Algebraic Structure of Causally Closed Sets

In this section we study the algebraic structure induced by the closure operator defined above. In particular, we show that, for the general class of occurrence nets here considered, this closure operator gives rise to an orthomodular lattice.

In Section 4.1 some properties of lattices generated in this way are presented, and some relations with K-density are shown.

Finally, in Section 4.2 we discuss the application of the same construction to partial orders of events obtained by deleting conditions from occurrence nets. We show that, in this case, the lattice can be non orthomodular.

We call $L(N)$ the collection of causally closed sets of $N = (B, E, F)$, and $X = B \cup E$. By the results on closure operators recalled in Section 2.3, we know that

$$\mathcal{L}(N) = \langle L(N), \subseteq, \emptyset, X, (\,.\,)^\perp \rangle$$

is an orthocomplemented complete lattice, in which the meet is just set intersection, while the join of a family of elements is given by set union followed by causal closure.

Now we present the main result of this paper, namely, the proof that the collection $L(N)$ of causally closed sets of $N$ forms an orthomodular lattice. In order to do this we need the following proposition.

**Proposition 6.** *Let $S_1, S_2 \in L(N)$ be such that $S_1 \subseteq S_2$. Then $\forall x \in (S_1 \vee S_2^\perp) \setminus (S_1 \cup S_2^\perp)$, $\exists y_1 \in S_1$, $\exists y_2 \in S_2^\perp : (x\ li\ y_1) \wedge (x\ li\ y_2)$.*

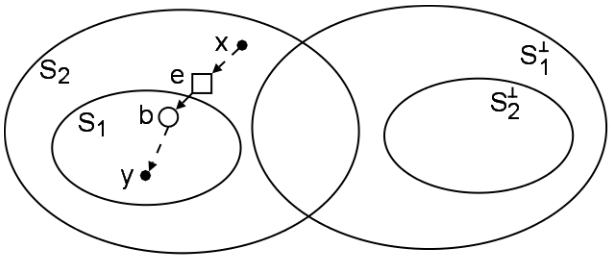


**Fig. 9.** An example of an element $x \notin (S_1 \vee S_2^{\perp}) \setminus (S_1 \cup S_2^{\perp})$

*Proof.* Let $S_1, S_2 \in L(N)$, with $S_1 \subseteq S_2$ (see Fig. 9). Then $S_2^{\perp} \subseteq S_1^{\perp}$, and $S_1 \; co \; S_2^{\perp}$. We show that if an element $x \in (S_1 \vee S_2^{\perp}) \setminus (S_1 \cup S_2^{\perp})$ belongs to $S_1^{\perp} \cup S_2$, then we have a contradiction. Put $M = S_1 \vee S_2^{\perp}$ . Let $x \in M \setminus (S_1 \cup S_2^{\perp})$; suppose that $x \in S_2 \setminus S_1$. The element $x$ cannot belong to $S_2 \cap S_1^{\perp}$, since in that case $x \in (S_1 \cup S_2^{\perp})^{\perp}$, which contradicts $x \in S_1 \vee S_2^{\perp}$. So $x \in S_2 \setminus S_1^{\perp}$. Since $x \notin S_1^{\perp}$, there exists $y \in S_1$ with $x \; li \; y$. Assume that in $N$ there is a path directed from $x$ to $y$ (if the path is directed the other way, the argument is symmetric). Let $(e, b) \in F$ be the arc crossing the border of $S_1$ along the path; $b \in B$, since $S_1$ is closed. We will show that $e$ cannot actually belong to $S_2$ by deriving a contradiction. By Proposition 5, $e \notin Ad(S_1)$; hence, there is $z \in F^+(e)$ such that $[e, z] \cap S_1 = \emptyset$ and $F^+(z) \cap S_1 = \emptyset$ (see Fig. 10). Choose a path from $e$ to $z$. Let $e_i$ be the last event along this path whose future crosses $S_1$. Hence there is $w \in \mu(S_1)$ such that $w \in F^+(e_i)$. By hypothesis $e \in M$ ($x$ and $y$ belong to

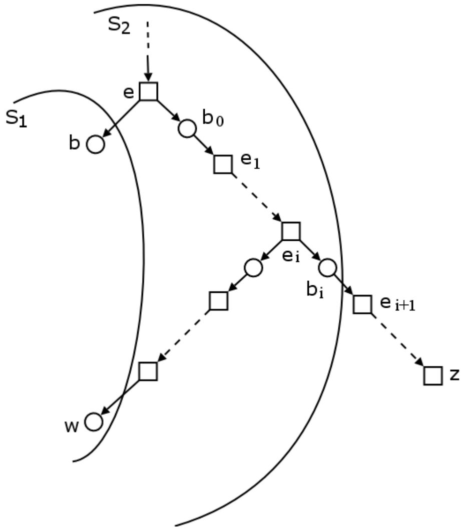


**Fig. 10.** An example of an event $e \notin (S_1 \vee S_2^{\perp}) \setminus (S_1 \cup S_2^{\perp})$

$M$ and $M$ is convex); but also $w \in M$, hence also $e_i \in M$. Let us call $b_i$ the postcondition of $e_i$ on the selected path from $e$ to $z$; $b_i$ must belong to $M$. By the convexity of $S_2$, $e_i \in S_2$, hence $b_i \in S_2$. If $b_i \in F^+(S_1)$ also $e_i \in F^+(S_1)$ which is in contradiction with the assumption $S_1 \in L(N)$. By definition of $e_i$, $b_i \notin F^-(S_1)$, hence $b_i \in S_1^\perp$. We have $b_i \in (S_2 \cap S_1^\perp)$, from which follows $b_i \in (S_1 \cup S_2^\perp)^\perp$ in contradiction with assumption $b_i \in (S_1 \vee S_2^\perp)$. Therefore $e \notin S_2$, and we already know that $e \notin S_2^\perp$. By a similar proof $e \notin S_1^\perp$ and $e \notin S_1$. Hence $\exists y_1 \in S_1,\ \exists y_2 \in S_2^\perp : (e\ li\ y_1) \wedge (e\ li\ y_2)$. □

**Theorem 1.** *$\mathcal{L}(N)$ is orthomodular.*

*Proof.* We show that, for $S_1, S_2 \in L(N)$, if $S_1 \subseteq S_2$, then $S_1 = S_2 \wedge (S_1 \vee S_2^\perp) = S_2 \cap (S_1 \vee S_2^\perp)$.

Let $S_1 \subseteq S_2$. If $x \in S_1$, then $x \in S_2$, which implies $x \in S_2 \wedge (S_1 \vee S_2^\perp)$, hence $S_1 \subseteq S_2 \wedge (S_1 \vee S_2^\perp)$.

On the other hand, let $x \in S_2 \cap (S_1 \vee S_2^\perp)$; then $x \in S_2$ and $x \in (S_1 \vee S_2^\perp)$. From $S_1 \subseteq S_2 \Rightarrow S_1\ co\ S_2^\perp$, it follows that $S_1 \cup S_2$ is convex. We must then consider two cases:

1. $Ad(S_1 \cup S_2^\perp) = \emptyset$, then $S_1 \vee S_2^\perp = S_1 \cup S_2^\perp$. Suppose that $x \in S_2^\perp$; since by hypothesis $x \in S_2$, we obtain a contradiction, thus $x \in S_1$ and $S_1 \supseteq S_2 \wedge (S_1 \vee S_2^\perp)$.
2. $Ad(S_1 \cup S_2^\perp) \neq \emptyset$. Suppose that $x \in (S_1 \vee S_2^\perp) \setminus (S_1 \cup S_2^\perp)$. Then, from Proposition 6 it follows that $\exists y \in S_2^\perp : y\ li\ x$, contradicting our assumption $x \in S_2$. Thus $x \notin (S_1 \vee S_2^\perp) \setminus (S_1 \cup S_2^\perp)$; which implies $x \in S_1$ and $S_1 \supseteq S_2 \wedge (S_1 \vee S_2^\perp)$. □

The following propositions characterize orthogonality and compatibility of causally closed sets.

**Proposition 7.** *Let $S_1, S_2 \in L(N)$. $S_1 \perp S_2 \Leftrightarrow S_1\ co\ S_2$.*

*Proof.* From the definitions, $S_1 \perp S_2 \Leftrightarrow S_1 \subseteq S_2^\perp \Leftrightarrow S_1\ co\ S_2$. □

**Proposition 8.** *Let $S_1, S_2 \in L(N)$. $S_1\ \$\ S_2 \Leftrightarrow S_1 \cup S_2$ is a convex set.*

*Proof.* We first show the "$\Rightarrow$" implication. Let $S_1, S_2 \in L(N)$ and $S_1\ \$\ S_2$; then $\exists A_1, A_2, A \in L(N)$ such that $A_1 \perp A_2 \perp A \perp A_1$ and $S_1 = A_1 \vee A$ and $S_2 = A_2 \vee A$. Suppose that $S_1 \cup S_2$ is not a convex set; then $\exists x \in S_1 \setminus A$, $\exists y \in S_2 \setminus A : x\ li\ y$ and $[x, y] \nsubseteq S_1 \cup S_2$. Without loss of generality, we can assume that $x$ is in the frontier of $S_1$, $y$ in the frontier of $S_2$, and $x \sqsubset y$. Consider two cases:

(i) $x \in A_1$. Then $y \notin A_2$, since $A_1 \perp A_2$; since $y \in A \vee A_2$, by Proposition 6, there is $z \in A_2 : y\ li\ z$. It cannot be $y \sqsubset z$, because in that case, we would have a path from $A_1$ to $A_2$; hence $z \sqsubset y$. Since $y$ is a condition on the frontier of $S_2$, any path from $z$ to $y$ must contain the unique event in ${}^\bullet y$. But this event is outside $S_2$, and then $S_2$ would not be convex, which is a contradiction.

(ii) $x \notin A_1$. If $y \in A_2$, then apply a symmetric argument to that of point (i). If $y \notin A_2$, then, we can apply Proposition 6 both to $x$ and to $y$ to build a path from $A_1$ to $A_2$, in contradiction with the assumption $A_1 \perp A_2$.

Therefore $S_1 \cup S_2$ is a convex set.

We now show the reverse implication. Let $S_1, S_2 \in L(N)$ be such that $S_1 \cup S_2$ is a convex set. Take $A = S_1 \cap S_2$, $A_1 = S_1 \cap A^\perp$ and $A_2 = S_2 \cap A^\perp$. $A, A_1, A_2 \in L(N)$, since they are intersections of closed sets, and by construction $A_1 \perp A$ and $A_2 \perp A$. We now show that $A_1 \perp A_2$. Suppose $\exists x \in A_1, \exists y \in A_2$ such that $x\ li\ y$. Then there is a path from $x$ to $y$, which we call $\alpha$. By convexity of $S_1 \cup S_2$, $\alpha \subseteq S_1 \cup S_2$. From $A_1 \perp A$ it follows $\alpha \cap A = \emptyset$. Therefore there are $v, w$ such that $v \in S_1 \setminus S_2$, $w \in S_2 \setminus S_1$, $(v, w) \in F$ and $\alpha$ passes through the arc $(v, w)$; but $v$ and $w$ are then in the frontier of, respectively, $S_1$ and $S_2$, hence they should be both conditions, which is a contradiction. Hence $A_1 \perp A_2$. To prove that $S_1 \$ S_2$ we show that $S_1 = A_1 \vee A$ and $S_2 = A_2 \vee A$. Since $A_1 \subseteq S_1$, by Theorem 1, $S_1 = A_1 \vee (S_1 \wedge A_1^\perp)$. We now show $S_1 \wedge A_1^\perp = A$.

If $x \in A$, then $x \in S_1$ and, since $A \perp A_1$, $x \in A_1^\perp$. This means that $A \subseteq S_1 \wedge A_1^\perp$.

Let $x \in S_1 \wedge A_1^\perp$; by de Morgan's laws, $A_1^\perp = S_1^\perp \vee A$. Hence $x \in S_1$ and $x \in (S_1^\perp \vee A)$; $x$ cannot be an element of $S_1^\perp$. Suppose $x \in (S_1^\perp \vee A) \setminus (S_1^\perp \cup A)$; then by Proposition 6 $\exists y \in S_1^\perp : x\ li\ y$, in contradiction with the hypothesis $x \in S_1$. Hence $x$ must be in $A$.

Therefore $S_1 = A_1 \vee X$. An analogous proof holds for $S_2 = A_2 \vee X$. □

## 4.1 Some Properties of $\mathcal{L}(N)$ Related to K-Density

In this section we study some properties of lattices derived from occurrence nets related to K-density. The first property we prove is valid in general.

**Theorem 2.** *Let $S \in L(N)$, with $|S| < \infty$. Then, for each line $l$ of $N$, $l \cap (S \cup S^\perp) \neq \emptyset$.*

*Proof.* We proceed by contradiction. Suppose there is a line $l$ such that $l \cap (S \cup S^\perp) = \emptyset$. Then, each element of $l$ must be in relation $li$ with at least one element of $S$ (otherwise, it would belong to $S^\perp$).

Let $D_1 \subseteq l$ be the set of those elements of $l$ which lie in the past of $S$. $D_1$ must be a proper subset of $l$, since otherwise, $S$ being finite, at least one element of $S$ would be in relation $li$ with all of $l$, and $l$ would not be a maximal clique of $li$. Similarly, let $D_2 = \{x \in l \mid \exists y \in S : y \sqsubset x\}$; $D_2$ is also a proper subset of $l$. $D_1$ and $D_2$ are disjoint, since $S$ is convex.

If $D_1 \neq \emptyset$, then there is a greatest element (with respect to $\sqsubseteq$) of $D_1$, and this element must be an event, say $e$. The unique post-condition of $e$ on $l$ cannot belong to $D_2$, hence it is concurrent with every element of $S$; but in that case, it would be an element of $S^\perp$, in contradiction with the initial assumption.

If instead $D_1 = \emptyset$, then, since $D_2$ is a proper subset of $l$, there exist points of $l$ concurrent with all points of $S$, hence again we have a contradiction. □

If, in the statement of the previous theorem, we drop the finiteness condition on $S$, the property does not hold in general. In the proof of the following theorem, we show an example of an infinite closed set $S$ such that its orthocomplement is also infinite, and a line which intesects neither $S$ nor $S^\perp$. Notice that this requires the occurrence net to be non K-dense.

**Theorem 3.** *An occurrence net $N = (B, E, F)$ is K-dense if and only if,*

$$\forall S \in L(N), \forall l \in \mathrm{Lines}(N) : l \cap (S \cup S^\perp) \neq \emptyset$$

*Proof.* We first show that, if $N$ is not K-dense, then there exist a line $\lambda$ and a closed set $S$ such that $\lambda \cap (S \cup S^\perp) = \emptyset$.

Assume that $N$ is not K-dense. Then, by Proposition 1, one of the posets in Figure 1 embeds into $N$. Let $\lambda$ be a line of $N$ extending the set formed by all the $x_i$; such a line exists, because those elements form a clique of $li$. Let $Y_P = \{y_i | i \text{ is even}\}$ and $Y_D = \{y_i | i \text{ is odd}\}$. Clearly, $Y_D \subseteq Y_P^\perp$. Since each element of $\lambda$ is in relation $li$ with at least one element of $Y_P$, we also have $\lambda \cap Y_P^\perp = \emptyset$ and $\lambda \cap Y_P^{\perp\perp} = \emptyset$. Since $Y_P^\perp$ is closed, the proof is done.

Let us now prove the implication in the other direction. Assume there exist a line $\lambda$ and a closed set $S$ such that $\lambda \cap (S \cup S^\perp) = \emptyset$. Note first that $\lambda$ must be infinite: if it were finite, it should end with a condition (since $|e^\bullet| \geq 1$ for all events), which should be concurrent with all elements of $S$. Since $S = (S^\perp)^\perp$, from the hypothesis it follows that each element of $\lambda$ is in relation $li$ with at least one element of $S$. Hence each element of $\lambda$ is either in $F^-(S)$ or in $F^+(S)$, but not in both, since $S$ is convex. Moreover, if $x \in \lambda$ is in $F^-(S)$, then the same holds for each $y \in \lambda$, since, otherwise, the postcondition of the last element of $\lambda$ lying in $F^-(S)$ would be in $S^\perp$. Assume, then, that all elements of $\lambda$ lie in the past of $S$ (the other case is dealt with in a dual way). Consider the set $H$ formed by all those conditions in the frontier of $S$ belonging to some path coming from $\lambda$ (see Figure 11). We claim that this set is infinite, and that it is a clique of the

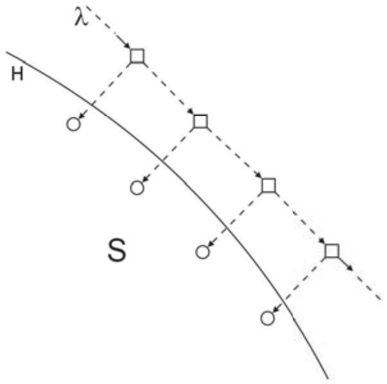


**Fig. 11.** Pattern of non K-dense occurrence nets

concurrency relation in $N$. To see that $H$ is infinite, take $x \in H$; since $[z, x]$ is finite for all $z \in B \cup E$, there must be a finite number of elements of $\lambda$ preceding $x$. Since $\lambda$ is infinite, also $H$ must be infinite. Suppose now that there are two elements, $v, w \in H$, linearly ordered: $v \sqsubseteq w$; a path from $v$ to $w$ must pass through the unique incoming arc of $w$, hence must leave $S$ and then re-enter it, but this is impossible, because $S$, being closed, is convex.

Now, by using elements on $\lambda$ and elements in $H$, it is easy to see that the poset on the left side of Figure 1 embeds into $N$, and $N$ is not K-dense. □

We now introduce a relation between K-dense occurrence nets and algebraic lattices.

**Proposition 9.** *If $N = (B, E, F)$ is not K-dense, then $\mathcal{L}(N)$ is not algebraic.*

*Proof.* Suppose $N$ is not K-dense; then there is a line $l$, and a cut $c$ such that $l \cap c = \emptyset$, and $l$ and $c$ are infinite. By the definition of cut, $c^{\perp\perp} = B \cup E$, since $c^{\perp} = \emptyset$; hence $l \subset c^{\perp\perp}$. Take an arbitrary $M \subset c, |M| < \infty$; we will show that $M^{\perp\perp} \cap l = \emptyset$. From this it follows that the closure operator $(.)^{\perp\perp}$ is not algebraic, hence $\mathcal{L}(N)$ is not algebraic.

Let us consider two distinct cases.

1. $\exists x \in l$ such that $x \ co\ M$; then $x \in M^{\perp}$, which implies that no element of $l$ can belong to $M^{\perp\perp}$.
2. $\forall x \in l, \exists y \in M : x\ li\ y$.
   Suppose first that in $l$ there are both elements in the past of $M$ and in the future of $M$. No such element can be in both, because otherwise there would be either a cycle in $N$ or a directed path between two distinct elements of $M$. Hence there is a greatest (with respect to $\sqsubseteq$) element, which is both in $l$ and in the past of $M$. The unique postcondition $b$ of such event on $l$ must then be concurrent with $M$. Hence $b \in M^{\perp}$ and so no element of $l$ can belong to $M^{\perp\perp}$.

   Suppose now that all elements of $l$ lie in the past of $M$. If $l$ had a last element $x$, then $x \sqsubset z$ for some $z \in M$, and the same would hold for all of $l$, which could not be a line. Hence $l$ does not have a last element; then, since $M$ is finite, there is at least one $w \in M$ which lies in the future of each element of $l$, and this contradicts the hypothesis that $l$ is a maximal clique of $li$.

   If all elements of $l$ lie in the future of $M$, a similar argument applies. □

It is still an open problem whether the reverse implication of Proposition 9 holds.

## 4.2 Lattices of Causally Closed Sets on Event Structures

In this section we briefly report some results on lattices generated by the concurrency relation in the elementary event structures associated to occurrence nets [11]. These structures are posets where $\sqsubseteq = F^*|_{E \times E}$ and the concurrency relation corresponds to the relation *co* in $N$ restricted to $E \times E$; obviously, also in this case, *co* is symmetric and irreflexive. We can then apply the constructions recalled in Section 2.3, obtaining the corresponding orthocomplemented complete lattices formed by the collection of causally closed sets.

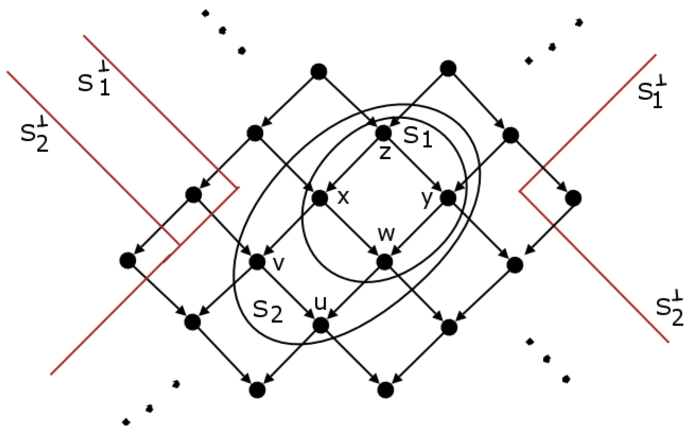


**Fig. 12.** The lattice constructed on event structures is not orthomodular

These causally closed sets have properties analogous to closed sets defined on occurrence nets; in particular, any closed set is convex. However, by dropping conditions, in general we lose orthomodularity, as shown by the following example.

*Example 3.* Fig. 12 shows a fragment of the event structure derived from Petri grid. In it we have, for instance, the closed sets $S_1 = \{x, y, z, w\}$, $S_2 = \{x, y, z, w, v, u\}$. They are such that: $S_1 \subset S_2$ and $S_2 \neq S_1 \vee (S_2 \wedge S_1^\perp)$, since $S_2 \wedge S_1^\perp = \emptyset$; therefore the orthomodularity law is not valid.

## 5 Conclusions

We have studied a closure operator induced by the *co* relation in an occurrence net, and the algebraic structure of the corresponding causally closed sets, showing that it is a complete orthomodular lattice.

The structure of closed sets has a sort of density property; in fact, as proved in Theorem 2, if we consider a finite closed set, then any line, which can be interpreted as a possible world line, crosses either the given set or its orthocomplement (which could then be interpreted as a negation). This property is actually valid for any closed set in the case of K-dense occurrence nets.

Starting from these first results, several further developments are possible. We are working on a further characterization of the causally closed sets of an occurrence net in terms of subprocesses of the process described by the net itself.

In [9], Casini suggests an interpretation of the causally closed sets of a space-time in terms of a logical language, whose formulas express properties of the history of a particle. The resulting logic is non classical, since the related lattice is orthomodular, but non distributive. We will investigate how to define a similar interpretation for our case.

On a more technical side, we intend to study the properties of lattices generated from nets, and in particular the problem to decide when a given orthomodular lattice corresponds to the structure of the causally closed sets of an occurrence net and how to construct it.

Kummer and Stehr have considered and studied causality and concurrency on cyclic processes ([18]); if we consider causally closed subsets of system nets with cyclic behaviour, in general we do not get an orthomodular structure. It is therefore interesting to characterize the class of system nets such that the lattices of closed sets are orthomodular, and to study the relations between the lattices obtained from the processes and from the systems.

## Acknowledgments

Work partially supported by MIUR, Regione Lombardia – FSE, and CNR-PAN. The authors thank the anonymous referees for several useful suggestions.

## References

1. Petri, C.A.: Non-sequential processes. Technical Report ISF-77–5, GMD Bonn (1977) Translation of a lecture given at the IMMD Jubilee Colloquium on 'Parallelism in Computer Science', Universität Erlangen–Nürnberg (June 1976)
2. Best, E., Fernandez, C.: Nonsequential Processes–A Petri Net View. EATCS Monographs on Theoretical Computer Science, vol. 13. Springer, Heidelberg (1988)
3. Wolfram, S.: A New Kind of Science. Wolfram Media (2002)
4. Petri, C.A.: Rechnender netzraum. Spektrum der Wissenschaft, Spezial 3/07: Ist das Universum ein Computer? 16–19 (2007)
5. Petri, C.A.: On the physical basis of information flow – abstract. In: van Hee, K.M., Valk, R. (eds.) PETRI NETS 2008. LNCS, vol. 5062, p. 12. Springer, Heidelberg (2008)
6. Bombelli, L., Lee, J., Meyer, D., Sorkin, R.: Spacetime as a causal set. Phys. Rev. Lett. 60, 521–524 (1985)
7. Abramsky, S.: Petri nets, discrete physics, and distributed quantum computation. In: Degano, P., De Nicola, R., Meseguer, J. (eds.) Concurrency, Graphs and Models. LNCS, vol. 5065, pp. 527–543. Springer, Heidelberg (2008)
8. Cegła, W., Jadczyk, Z.: Causal logic of Minkowski space. Commun. Math. Phys. 57, 213–217 (1977)
9. Casini, H.: The logic of causally closed spacetime subsets. Class. Quantum Grav. 19, 6389–6404 (2002)
10. Fernandez, C., Thiagarajan, P.S.: A lattice theoretic view of k-density. Arbeitspapiere der GMD, n. 76 (1983)
11. Nielsen, M., Plotkin, G.D., Winskel, G.: Petri nets, event structures and domains, part I. Theoretical Computer Science 13, 85–108 (1981)
12. Petri, C.A.: Nets, time and space. Theoretical Computer Science 153, 3–48 (1996)
13. Pták, P., Pulmannová, P.: Orthomodular Structures as Quantum Logics. Kluwer Academic Publishers, Dordrecht (1991)
14. Beltrametti, E.G., Cassinelli, G.: The logic of quantum mechanics. Encyclopedia of Mathematics and its Applications, vol. 15. Addison-Wesley, Reading (1981)

15. Birkhoff, G.: Lattice Theory, 3rd edn. American Mathematical Society (1979)
16. Davey, B.A., Priestley, H.A.: Introduction to Lattices and Order. Cambridge University Press, Cambridge (1990)
17. Petri, C.A.: Concepts of net theory. In: Mathematical Foundations of Computer Science: Proc. of Symposium and Summer School, High Tatras, September 3–8, 1973, pp. 137–146. Math. Inst. of the Slovak Acad. of Sciences (1973)
18. Kummer, O., Stehr, M.O.: Petri's axioms of concurrency: A selection of recent results. In: Azéma, P., Balbo, G. (eds.) ICATPN 1997. LNCS, vol. 1248, pp. 195–214. Springer, Heidelberg (1997)

# Hasse Diagram Generators and Petri Nets

Mateus de Oliveira Oliveira

School of Computer Science, Tel Aviv University, Tel Aviv, Israel
mateusde@tau.ac.il

**Abstract.** In [LJ06] Lorenz and Juhás raised the question of whether there exists a suitable formalism for the representation of infinite families of partial orders generated by Petri nets. Restricting ourselves to bounded $p/t$-nets, we propose *Hasse diagram generators* as an answer. We show that Hasse diagram generators are expressive enough to represent the partial order language of any bounded $p/t$ net. We prove as well that it is decidable both whether the (possible infinite) family of partial orders represented by a given Hasse diagram generator is included on the partial order language of a given $p/t$-net and whether their intersection is empty. Based on this decidability result, we prove that the partial order languages of two given Petri nets can be effectively compared with respect to inclusion. Finally we address the synthesis of $k$-safe $p/t$-nets from Hasse diagram generators.

**Keywords:** Causality/partial order theory of concurrency.

## 1 Introduction

When dealing with the development and analysis of concurrent systems, some questions may arise naturally. We may want to know, for example, whether the behavior of a given system subsumes a specified collection of desired scenarios or whether it contains some, out of a collection of undesired scenarios. Given two systems $A$ and $B$, we could ask as well whether the behavior of $A$ is included in the behavior of $B$. Finally, we could be interested in synthesizing a system whose behavior minimally includes a given set of scenarios. In this paper, we address these questions under an unifying perspective. We formalize our systems by means of $p/t$-nets and our scenarios by means of partial orders.

In order to finitely represent possible infinite families of partial orders, we introduce *Hasse diagram generators*. With this aim, we define *slices* as the building blocks of directed acyclic graphs (DAGs). We specify languages over slices by means of what we call *slice graphs*. Composing the slices on the strings of these languages, we get infinite families of *DAGs*, the transitive closure of which, gives rise to infinite families of partial orders. A Hasse diagram generator is a slice graph which generates exclusively transitive reduced DAGs. For this special type of slice graph, each generated *DAG* is the Hasse diagram of the partial order it induces.

Partial orders can be associated to $p/t$-nets via the notion of Petri net process [GR83, BW00]. They may carry two possible semantics. The causal and the execution semantics. Accordingly to the causal semantics, a vertex $v$ is connected to a vertex $v'$ in a partial order if and only if the event represented by $v'$ causally depends on the occurrence the of event represented by $v$. The absence of an edge between two events indicates independence. The execution semantics is not aimed to indicate a causal dependence between events, but rather the order in which they are executed. An edge connecting $v$ to $v'$ indicates that $v'$ can be executed either after or at the same time as $v$, but not before. The absence of edge between $v$ and $v'$ indicates that they can be executed in any order. For a given Petri net $N$ we say that a partial order $PO$ is a *causal order* of $N$ if it carries the causal semantics and an *execution* of $N$ if it carries the execution semantics. We denote $\mathcal{L}_{cau}(N)$, the set of causal orders of $N$, and $\mathcal{L}_{ex}(N)$, its set of executions.

Our main result states that for any Hasse diagram generator $\mathcal{HG}$, and any bounded $p/t$-net $N$, it is decidable whether the language $\mathcal{L}_{PO}(\mathcal{HG})$ of partial orders generated by $\mathcal{HG}$ is included on $\mathcal{L}_{sem}(N)$, as well as whether the intersection of $\mathcal{L}_{PO}(\mathcal{HG})$ with $\mathcal{L}_{sem}(N)$ is empty. We use the variable *sem* to indicate that these decidability results are valid for both the causality semantics ($sem = cau$) and for the execution semantics ($sem = ex$). Previously, decidability was stated for single ( and consequently finite families of ) partial orders carrying both semantics [JLD05]. In order to provide supporting evidence that Hasse diagram generators are indeed a suitable formalism for the representation of the partial order language generated by $p/t$-nets, we show that for each $p/t$-net $N$ there exist effectively computable Hasse diagram generators $\mathcal{HG}_{ex}(N)$ and $\mathcal{HG}_{cau}(N)$, whose partial order languages match respectively the set of executions of $N$ and the set of causal orders of $N$.

In the reverse direction, we study the synthesis of Petri nets from Hasse diagram generators. We restrict ourselves to $k$-safe $p/t$-nets for a given $k$. Here, the term $k$-safe is used in order to emphasize that the bound $k$ is given as a parameter for the synthesis. We develop the notion of $k$-safe region for slice graphs. Using these regions, we are able to synthesize from a given Hasse diagram generator $\mathcal{HG}$, a Petri net with minimal execution language with relation to the partial order language represented by $\mathcal{HG}$. For the causal semantics we impose as well a bound $r$ on the number of copies of each place of the synthesized net. With this additional care, we are able to synthesize the set of minimal nets whose causal language is minimal with respect to the partial order language defined by $\mathcal{HG}$. It is worth noting that this work deals for the first time with the synthesis of (unlabeled) $p/t$-nets from partial order languages carrying the causal semantics. Synthesis of Petri nets from many types of automata and sequential or step languages had previously been achieved through the theory of regions [Dar98, Dar00, BD96, ED96]. In [LJ06] an abstract notion of region was defined for partial order languages. This notion was effected in [LBDM07, BDLM08] for finite sets of partial orders carrying the execution semantics and in [LBD08] for a class of partial orders which is not expressive enough to represent the partial order behavior of arbitrary bounded $p/t$-nets.

The rest of the paper is organized as follows. In section 2 we define slices, slice graphs and Hasse diagram generators. In section 3 we describe $p/t$-nets, their processes and their execution and causal order languages. In section 4 we introduce the notion of *interlaced flow* in order to characterize Hasse diagrams of $p/t$-net executions and processes. Interlaced flows can be regarded as a generalization of token flows defined in [JLD05, BDJL06]. In sections 5 and 6 we introduce respectively *seasoners* and *filters*, which provide a nice framework for the statement of our main results in section 7. In section 8 we address the synthesis of $k$-safe $p/t$-nets from Hasse diagram generators and in section 9 we make some final remarks.

## 2 Slices, Slice Graphs and Hasse Diagram Generators

A *slice* is a DAG $\mathbf{S} = (V, E, l)$ where $V = I \dot\cup C \dot\cup O$ and $l : V \rightarrow \{\mathcal{I}_1, ..., \mathcal{I}_{|I|}\} \cup T \cup \{\mathcal{O}_1, ..., \mathcal{O}_{|O|}\}$. The vertex set $V$ is partitioned into three subsets: A non-empty center $C$ labeled by $l$ with elements of a set of transitions $T$ and the in- and out-frontiers, $I$ and $O$ respectively, which are injectively numbered by $l$ in such a way that $l(I) = \{\mathcal{I}_1, ..., \mathcal{I}_{|I|}\}$ and $l(O) = \{\mathcal{O}_1, ..., \mathcal{O}_{|O|}\}$. Furthermore an unique edge touches each frontier vertex $v \in I \dot\cup O$. This edge is outgoing if $v$ lies on the in-frontier $I$ and incoming if $v$ lies on the out-frontier $O$. It will be convenient to write simple $j$ to denote the unique edge which touches in the $j$-th vertex $v$ of a frontier, i.e, for which $l(v) \in \{\mathcal{I}_j, \mathcal{O}_j\}$. Whether $v$ is an in or out-frontier vertex will be clear by the context. For such an edge $j$, $j^s$ denotes its source vertex and $j^t$ its target vertex. Along all this paper, for an edge $e$ of a graph, $e^s$ will denote its source vertex and $e^t$ its target vertex.

In drawings, we surround slices by dashed rectangles, and implicitly direct their edges from left to right. In and out frontiers vertices are determined respectively by the intersection of edges with the left and right sides of the rectangle. Frontier vertices are implicitly numbered from up to down. Center vertices are indicated by their labels (Fig: 1.$I$). A slice $\mathbf{S}$ can be composed with a slice $\mathbf{S}'$ whenever the out-frontier of $\mathbf{S}$ is of the same size as the in-frontier of $\mathbf{S}'$. In this case, the resulting slice $\mathbf{S} \circ \mathbf{S}'$ is obtained by gluing the single edge touching $j$-th out-frontier vertex of $\mathbf{S}$ to the corresponding edge touching the $j$-th in-frontier vertex of $\mathbf{S}'$ ( Fig. 1.$II$). We note that as a result of the composition, multiple edges may arise, since the vertices on the glued frontiers disappear. Formally we have

$$\mathbf{S} \circ \mathbf{S}' = [(\mathbf{S} - O) \dot\cup (\mathbf{S}' - I')] + \{(j^s, j^t) | 1 \leq j \leq |O|\}.$$

Where $\dot\cup$ stands for the disjoint union of multigraphs. The minus operation stands for the usual deletion of vertices and the + operation for the usual addition of edges on multigraphs. It is easy to see that any DAG, even containing multiple edges, can be cast as the composition of a sequence of unit slices.

We say a slice is *initial* if its in-frontier is empty and *final* if its out-frontier is empty. A slice with a unique vertex in the center is called a *unit slice*. A unit slice is standard if there is at least one edge connecting its center vertex to an in(out)-frontier vertex, whenever the in(out)-frontier of $\mathbf{S}$ is not empty. If both

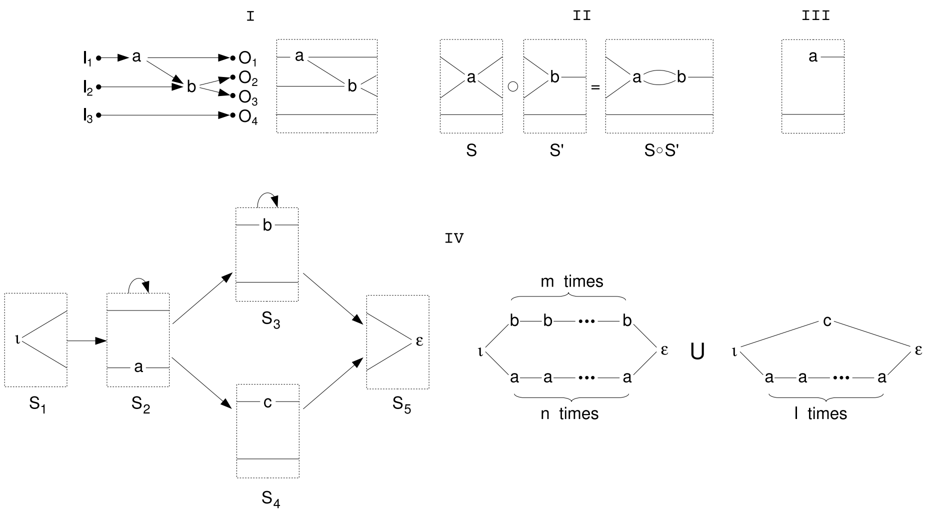


**Fig. 1.** *I*: A slice and its representation accordingly to our convention. *II*: Composition of slices. *III*: A slice which is not standard. *IV*: A slice graph labeled with unit slices, and an intuitive representation of its graph language. $\mathbf{S}_1$ is initial and $\mathbf{S}_5$, final.

the in- and out-frontiers of a slice are empty, we say that it is a pure DAG. In this paper we assume that unit initial slices have its center vertex labeled by $\iota$ and that unit final slices have its center vertex labeled by $\epsilon$.

**Definition 1 (Slice Graph).** *A* slice graph *over a slice alphabet $\Sigma_{\mathbb{S}}$ is a directed graph $\mathcal{SG} = (\mathcal{V}, \mathcal{E}, \mathcal{S})$ possible containing loops but free of multiple edges. The function $\mathcal{S} : \mathcal{V} \rightarrow \Sigma_{\mathbb{S}}$ satisfies the following condition: $(v_1, v_2) \in \mathcal{E}$ implies that $\mathcal{S}(v_1)$ can be composed with $\mathcal{S}(v_2)$. We say that a vertex on a slice graph is initial if it is labeled with an initial slice and final if it is labeled with a final slice.*

A graph is simple if it doesn't contain multiple edges. The simplification of a multigraph $G$ is the simple graph obtained from it by letting on it a single copy of each edge. A *DAG* is a directed and acyclic graph. A vertex of a *DAG* is minimal (maximal) if there is no edge in which $v$ is the tail (source). A partial order is a simple *DAG* where $E$ is irreflexive and transitive. The partial order induced by a DAG $G$ is the transitive closure $G^*$ of the simplification of $G$. The Hasse diagram of $G$ is the Hasse diagram of its induced partial order.

**Definition 2 (Languages Generated by Slice Graphs)**

*(i) Slice language:* $\mathcal{L}(\mathcal{SG}) = \{\mathcal{S}(v_1)\mathcal{S}(v_2)...\mathcal{S}(v_n) : v_1v_2...v_n$ *is a walk on* $\mathcal{SG}$ *from an initial to a final vertex*$\}$
*(ii) Graph language:* $\mathcal{L}_G(\mathcal{SG}) = \{\mathbf{S}_1 \circ \mathbf{S}_2 \circ ... \circ \mathbf{S}_n | \mathbf{S}_1\mathbf{S}_2...\mathbf{S}_n \in \mathcal{L}(\mathcal{SG})\}$
*(iii) PO language:* $\mathcal{L}_{PO}(\mathcal{SG}) = \{H^* | H \in \mathcal{L}_G(\mathcal{SG})\}$.

**Definition 3 (Hasse Diagram Generator).** *A hasse diagram generator is a slice graph $\mathcal{SG}$ in which all elements of $\mathcal{L}_G(\mathcal{SG})$ are Hasse diagrams.*

## 3 p/t-Nets and Their Partial Order Semantics

Let $T$ be a set of transitions. Then a place over $T$ is a triple $p = (p_0, \check{p}, \hat{p})$ where $p_0$ denotes the initial number of tokens on $p$ and $\check{p}, \hat{p} : T \to \mathbb{N}$ are functions which denote the number of tokens that a transition $t$ respectively puts on and takes from $p$. A $p/t$-net over $T$ is a pair $N = (P, T)$ where $T$ is a set of transitions ant $P$ a multiset of places over $T$. We assume through this paper that for each transition $t \in T$, there exists places $p_1, p_2 \in P$ for which $\check{p}_1(t) \geq 0$ and $\hat{p}_2(t) \geq 0$. A marking of $N$ is a function $m : P \to \mathbb{N}$. A transition $t$ is enabled at marking $m$ if $m(p) \geq \hat{p}(t)$ for each $p \in P$. The occurrence of an enabled transition at marking $m$ gives rise to a new marking $m'$ defined as $m'(p) = m(p) - \hat{p}(t) + \check{p}(t)$. The initial marking $m_0$ of $N$ is given by $m_0(p) = p_0$ for each $p \in P$. A sequence of transitions $t_0 t_1 ... t_{n-1}$ is an occurrence sequence of $N$ if there exists a sequence of markings $m_0 m_1 ... m_n$ such that $t_i$ is enabled at $m_i$ and if $m_{i+1}$ is obtained by the firing of $t_i$ at marking $m_i$. A $p/t$-net $N$ is $k$-safe, for $k \geq 1$, if for each occurrence $t_0 t_1 ... t_{n-1}$ of $N$, $m_i(p) \leq k$ for each $0 \leq i \leq n$. $N$ is bounded if it is $k$-safe for some $k$. The union of two $p/t$-nets $N_1 = (P_1, T), N_2 = (P_2, T)$ over $T$ is the $p/t$-net $N_1 \cup N_2 = (P_1 \dot{\cup} P_2, T)$.

**Definition 4 (Process).** *A process of a p/t-net $N = (P, T)$ is a DAG $\pi = (B \dot{\cup} V, F, \rho)$ where $B \dot{\cup} V$ is the set of vertices. We cal the vertices in $B$* conditions *and the vertices in $V$* events*. The edge set $F$ is included on $(B \times V) \cup (V \times B)$ and $\rho : (B \cup V) \to (P \cup T) \cup \{\iota, \epsilon\}$ is a labeling function satisfying*

*(i)* *$\pi$ has an unique minimal vertex $v_\iota \in V$ and an unique maximal vertex $v_\epsilon \in V$.*
*(ii)* $\rho(B) \subseteq P$, $\rho(V \backslash \{v_\iota, v_\epsilon\}) \subseteq T$, $\rho(v_\iota) = \iota$, $\rho(v_\epsilon) = \epsilon$.
*(iii)* $\forall v \in V, \forall p \in P : |\{(b, v) \in F : \rho(b) = p\}| = \hat{p}(\rho(v))$
*(iv)* $\forall v \in V, \forall p \in P : |\{(v, b) \in F : \rho(b) = p\}| = \check{p}(\rho(v))$.
*(v)* $\forall p \in P : |\{(v_\iota, b) : \rho(b) = p\}| = p_0$.

The only point our definition of process differs from the usual definition of $p/t$-net process [GR83] is the addition of a minimal event $v_\iota$ which is labeled with a letter $\iota \notin T$ and a maximal event $v_\epsilon$ which is labeled with a letter $\epsilon \notin T$. Intuitively, $\iota$ loads the initial marking of $N$, and $\epsilon$ empties the marking of $N$ after the occurrence of all events of the process. We call attention to the fact that the number of conditions connected to $v_\epsilon$ varies accordingly to the process. These minimal and maximal events will be useful to avoid the consideration of particular cases on theorem 10.

A partial order $po = (V, <, l)$ is a simple graph where $<$ is a transitive and irreflexive relation over $V \times V$. A prefix of $po$ is a partial order $po' = (V', <', l')$ where $V' \subseteq V, <' \subseteq <, l' = l|_V$ and for all $v' \in V', v \in V$, $v < v'$ implies that $v \in V'$ and $v <' v'$. A sequentialization of $po$ is a partial order $po' = (V, <', l)$

where $<\subseteq<'$. The causal order of a process $\pi$ is obtained from it by abstracting its conditions and by taking the partial order induced by its events. An execution is a sequentialization of a causal order.

**Definition 5 (Causal order of a $p/t$-net Process and Executions).** *The causal order of a process $\pi = (B\dot{\cup}V, F, \rho)$ of a $p/t$-net $N$ is the partial order $po_\pi = (V, F^*|_{V\times V}, \rho|_V)$. An execution of $\pi$ is a partial order $po' = (V, F', \rho)$ where $F \subseteq F'$.*

We denote $\mathcal{L}_{cau}(N)$ the set of all causal orders derived from processes of $N$ and $\mathcal{L}_{ex}(N)$ the set of all its executions. When talking about a prefix of a causal order (execution) of a process, we take the additional care of adding a maximum vertex labeled with $\epsilon$, so that it is itself a causal order (execution) of $N$. We notice that with this proviso, $\mathcal{L}_{cau}(N)$ is prefix closed and $\mathcal{L}_{ex}(N)$ is both prefix and sequentialization closed.

## 4 Interlaced Flows, Executions and Causal Orders

In [JLD05, BDJL06] executions and causal orders of Petri nets were characterized in terms of token flows. These flows however cannot be transposed to Hasse diagram generators, since a flow value must be assigned to each edge of a given partial order. In this section we introduce the notion of *p-interlaced flow*, which non-trivially generalizes token flows in the sense that they allow us to characterize Petri net partial orders by attaching values only to the edges of their Hasse diagrams. Instead of a single value, our flow assigns four values to each edge. Establishing an interlacing relation on these values, we are able to coherently derive Petri net processes without considering edges which are not in the transitive reduction of the partial order. As it will be clear in sections 5 and 7, interlaced flows can be easily "sliced" and then transposed to Hasse diagram generators.

**Definition 6 (Interlaced Flow).** *Let $H = (V, E, l)$ be a Hasse diagram. Then a four-tuple $f = (\boldsymbol{bb}, \boldsymbol{bf}, \boldsymbol{pb}, \boldsymbol{pf})$ of functions of type $E \to \mathbb{N}$ is called an interlaced flow on $H$ if the following equation is satisfied for each vertex $v \in V$:*

$$\sum_{e^t=v} \boldsymbol{bf}(e) + \boldsymbol{pf}(e) = \sum_{e^s=v} \boldsymbol{pb}(e) + \boldsymbol{pf}(e) \tag{1}$$

Intuitively, for each $e \in E$, $\boldsymbol{pb}(e)$ counts some of the tokens produced in the **p**ast of $e^s$ and consumed **b**y $e^t$; $\boldsymbol{pf}(e)$, some of the tokens produced in the **p**ast of $e^s$ and consumed in the **f**uture of $e^t$, and $\boldsymbol{bf}(e)$, some of the tokens produced **b**y $e^s$ and consumed in the future of $e^t$. Thus equation 1 states that on interlaced flows, the total number of tokens produced in the past of a vertex $v$, that arrives at it without being consumed, will eventually be consumed in the future of $v$. $\boldsymbol{bb}(e)$, which does not appears on equation 1 counts the totality tokens produced **by** $e^s$ and consumed **by** $e^t$. This component of the flow will be used below in the definition of unit flows as well as in the definition of $p$-interlaced flows.

As we shall see in lemma 8 any interlaced flow can be decomposed on a sum of unit flows. Each unit flow keeps track of the trajectory of a unique token, from the event in which it is created until the event in which it is consumed. The sum of flows is performed componentwise. We recall that the value associated by a flow $f$ to an edge $e$ of a DAG is the four-tuple $(\boldsymbol{bb}(e), \boldsymbol{bf}(e), \boldsymbol{pb}(e), \boldsymbol{pf}(e))$.

**Definition 7 (Unit flow).** *Let $H = (V, E, l)$ be a DAG and $w = v_1 e_1 v_2 \ldots v_k e_k v_{k+1}$ be a path on $H$. Then the* unit flow *of $w$ on $H$, is the function $f_w : E \to \mathbb{N}^4$ where $f_w(e) = (0,0,0,0)$ for $e \in E \backslash \{e_1 \ldots e_k\}$ and $f_w(e_i)$ is defined as follows for $1 \leq i \leq k$:*

$$f_w(e_i) = \begin{cases} (1,0,0,0) & \textit{if} \quad i = 1 \quad \textit{and} \;\; k = 1 \\ (0,1,0,0) & \textit{if} \quad i = 1 \quad \textit{and} \;\; k > 1 \\ (0,0,0,1) & \textit{if} \;\; 1 < i < k \;\; \textit{and} \;\; k > 1 \\ (0,0,1,0) & \textit{if} \quad i = k \quad \textit{and} \;\; k > 1 \end{cases} \tag{2}$$

Intuitively $f_w$ represents a unique token that is produced at the first vertex of $w$ and is consumed at its last vertex, after traveling untouched along all intermediary vertices. At each edge, one component of the interlaced flow is equal to 1 and the other three are 0. If $w$ has a unique edge, i.e. $k = 1$, the token produced by $e_1^s$ is consumed by $e_1^t$, thus $\boldsymbol{bb}(e_1) = 1$. For $k > 1$, the token produced by $e_1^s$ travels to the future of $e_1^t$. In this case, $\boldsymbol{bf}(e_1) = 1$. For $1 < i < k$, the token is produced in the past of $e_i^s$ and consumed in its future, hence $\boldsymbol{pf}(e_i) = 1$. Finally, for $e_k$ we have $\boldsymbol{pb}(e_k) = 1$ since the token is produced in the past of $e_k^s$ and consumed by $e_k^t$. It is easy to verify that for any path $w$ of $H$, $f_w$ is an interlaced flow on $H$. This and some other properties of interlaced flows are described in the following lemma, which will be used in the proof of theorem 10.

**Lemma 8.** *Let $H = (V, E, l)$ be a Hasse diagram, $w$ a path in $H$ and $f, f' : E \to \mathbb{N}^4$ be two interlaced flows on $H$ where $f = (\boldsymbol{bb}, \boldsymbol{bf}, \boldsymbol{pb}, \boldsymbol{pf})$ and $f' = (\boldsymbol{bb}', \boldsymbol{bf}', \boldsymbol{pb}', \boldsymbol{pf}')$. Then*

1. *If $f(e) = (0,0,0,0)$ for every $e \in E$ then $f$ is an interlaced flow on $H$.*
2. *$f_w$ is an interlaced flow on $H$.*
3. *$f + f'$ is an interlaced flow on $H$.*
4. *Let $w = v_1 e_1 v_2 \ldots v_k e_k v_{k+1}$ be a path on $H$ and suppose $f = f' + f_w$, then*
   (a) $\boldsymbol{bb}(e_1) + \boldsymbol{bf}(e_1) = \boldsymbol{bb}'(e_1) + \boldsymbol{bf}'(e_1) + 1$
   (b) $\boldsymbol{pb}(e_k) + \boldsymbol{bb}(e_k) = \boldsymbol{pb}'(e_k) + \boldsymbol{bb}'(e_k) + 1$
5. *Let $e \in E$. Then*
   (a) $\boldsymbol{bb}(e) + \boldsymbol{bf}(e) \geq 1$ *if and only if there exists a path $w = v_1 e_1 v_2 .. v_k e_k v_{k+1}$ and an interlaced flow $f'$ on $H$ such that $e_1 = e$ and $f = f' + f_w$.*
   (b) $\boldsymbol{pb}(e) + \boldsymbol{bb}(e) \geq 1$ *if and only if there exists a path $w = v_1 e_1 v_2 .. v_k e_k v_{k+1}$ and an interlaced flow $f'$ on $H$ such that $e_k = e$ and $f = f' + f_w$.*
6. *There exists a multiset $M$ of paths of $H$ such that $f = \sum_{w \in M} f_w$.*

*7. Let $M$ be a multiset of paths of $H$ and $f = \sum_{w \in M} f_w$. Then for each $v \in V$*
*(a) $\sum_{e^s=v} \boldsymbol{bb}(e) + \boldsymbol{bf}(e) = |\{w \in M | v \text{ is the first vertex of } w\}|$*
*(b) $\sum_{e^t=v} \boldsymbol{bb}(e) + \boldsymbol{pb}(e) = |\{w \in M | v \text{ is the last vertex of } w\}|$*

*Proof.* Items 1-3 follow from the definitions of interlaced flow (6) and of unit flow (7 ). Items 4.a and 4.b follow from the definition of unit flow and holds even if $f$ is not interlaced. For the proof of one direction of item 5.a, suppose that there is a path $w = v_1 e_1 v_2 ... v_k e_k v_{k+1}$ where $e_1 = e$, $f = f' + f_w$ and $f'$ is interlaced. Then by item 4.*a* $\boldsymbol{bb}(e) + \boldsymbol{bf}(e)$ must be greater than 1, since $\boldsymbol{bb}'(e) + \boldsymbol{bf}'(e) \geq 0$. For the other direction, suppose $\boldsymbol{bb}(e) + \boldsymbol{bf}(e) \geq 1$. If $\boldsymbol{bb}(e) \geq 1$ then the path has length 1 and consists in $e$ itself. Now let $\boldsymbol{bb}(e) = 0$ and $\boldsymbol{bf}(e) \geq 1$. Then by the equation of definition 6, there exists an edge $e_2$ in $H$ with $e^t = e_2^s$ for which $\boldsymbol{pb}(e_2) + \boldsymbol{pf}(e_2) \geq 1$. If $\boldsymbol{pb}(e_2) \geq 1$ take $w = (e^s)e(e^t)e_2(e_2^t)$. Otherwise also by the equation of definition 6 there exists an edge $e_3$ with $e_2^t = e_3^s$ and $\boldsymbol{pb}(e_3) + \boldsymbol{pf}(e_3) \geq 1$ and so on. For some $k$ we must have $\boldsymbol{pb}(e_k) \geq 1$ since $H$ is finite. This $e_k$ will be the last edge of the path. The proof of item 5.b is analogous. For the first direction we use 4.b instead of 4.a. For the other direction we construct the path $w$ in the reverse order, starting from the last edge. Item 6 follows from items 5.a and 5.b: decompose $f$ sucessively until the zero interlaced flow is reached. Indeed, item 5 assures that while the flow is non-zero, such decomposition is possible. Item 7.a follows from items 5.a and 4.a and item 7.b from 5.b and 4.b. □

**Definition 9 ($p$-interlaced flow).** *Let $N = (P, T)$ be a $p/t$-net $H = (V, E, l)$ a Hasse diagram with $l : V \to T$ and $p \in P$ a place of $N$. Then a $p$-interlaced flow is an interlaced flow $f : E \to \mathbb{N}^4$ which satisfies the two following additional equations around each vertex:*

$$(IN) \qquad In(v) = \sum_{e^t=v} \boldsymbol{bb}(e) + \boldsymbol{pb}(e) = \hat{p}(l(v))$$

$$(OUT) \qquad Out(v) = \sum_{e^s=v} \boldsymbol{bb}(e) + \boldsymbol{bf}(e) = \check{p}(l(v))$$

### 4.1 Characterization of Executions and Causality Diagrams in Terms of $p$-Interlaced Flows

In this subsection we enunciate and prove theorem 10, which characterizes Petri net partial orders in terms of interlaced flows. Item $(i)$ of theorem 10 states that each Hasse diagram $H$ whose induced partial order $H^*$ is an execution of a $p/t$-net $N$ can be characterized by a set of $p$-interlaced flows, one for each place of $N$. The proof is constructive. Given that $H^*$ is an execution of $N$, we construct a set of interlaced flows on $H$, one for each place of N. Conversely, if such a set of $p$-interlaced flows is given, we construct a process $\pi$ of $N$ for which $H^*$ is one of its executions. Making some additional considerations on each direction of the proof, we prove as well item $(ii)$ of theorem 10 which characterizes Hasse diagrams of causal orders of $N$.

**Theorem 10 (Interlaced flow theorem).** *Let $N = (P,T)$ be a (not necessarily bounded) p/t-net and $H = (V,E,l)$ be a Hasse diagram. Then*

*(i) The partial order induced by $H$ is an execution of $N$ iff there exists a p-interlaced flow $f_p : E \to \mathbb{N}^4$ on $H$ for each place $p$.*
*(ii) The partial order induced by $H$ is a causal order of $N$ iff there exists a set $\{f_p\}_{p\in P}$ of p-interlaced flows such that for at least one $p$, the component of $f_p$ which denotes the direct transmission of tokens is strictly greater than $1$.*

**From Processes to Flows.** Let $\pi = (B\dot{\cup}V, F, \rho)$ be a process of $N$ and suppose that $H^* = (V,<,l)$ is an execution of $N$. For each place $p \in N$ we extend the functions $\hat{p}, \check{p} : T \to \mathbb{N}$ to $\check{p} : T \cup \{\iota, \epsilon\} \to \mathbb{N}$ and $\hat{p} : T \cup \{\iota\} \to \mathbb{N}$ by making $\check{p}(\iota) = p_0$, $\check{p}(\epsilon) = 0$ and $\hat{p}(\iota) = 0$ . $\hat{p}(\epsilon)$ represents the number of tokens on $p$ after the occurrence of every events of $\pi$. For each $b \in B$ with $(v,b),(b,v') \in F$ for some $v,v' \in V$ we choose an arbitrary path $w_b$ in $H$ (not in $H^*$) from $v$ to $v'$. Since $H$ is the Hasse diagram of a sequentialization of the causal order $po_\pi$ of $\pi$, such a path always exists. We claim that for each $p \in P$, $f_p = \sum_{\rho(b)=p} f_{w_b}$ is a $p$-interlaced flow of $N$. By lemma 8.2, each $f_{w_b}$ is interlaced. Thus by lemma 8.3, $f_p$ is interlaced as well. By the definition of process (4), for each $v \in V$ we have $|\{(v,b) \in F : \rho(b)=p\}| = \check{p}(\rho(v))$ and thus exactly $\check{p}(l(v))$ chosen paths whose first vertex is $v$. It implies, by lemma 8.4 that $\sum_{e^s=v} \boldsymbol{bb}_p(e) + \boldsymbol{bf}_p(e) = \check{p}(l(v))$. Thus condition $(OUT)$ of definition 9 is satisfied. Analogously $|\{(b,v) \in F : \rho(b)=p\}| = \hat{p}(\rho(v))$ and thus exactly $\hat{p}(l(v))$ chosen paths whose last vertex is $v$, what implies $\sum_{e^t=v} \boldsymbol{pb}_p(e) + \boldsymbol{bb}_p(e) = \hat{p}(l(v))$. Thus condition $(IN)$ of definition 9 is satisfied as well. This proves one direction of item $i$ of theorem 10. In order to prove the same direction of item $ii$, suppose that $H$ is the Hasse diagram of the causal order of $\pi$, and let $e = (v,v') \in E$ be one of the edges of $H$. Then for some $p \in P$ there exists a condition $b$ in $\pi$ whose label is $p$ for which $(v,b),(b,v') \in F$. Thus $f_{w_b}(e) = (1,0,0,0)$ which implies that $\boldsymbol{bb}(e) \geq 1$.

**From Flows to Processes.** Suppose that there is a $p$-interlaced flow $f_p : E \to \mathbb{N}^4$ on $H$ for each $p \in P$. We construct a process $\pi = (B\dot{\cup}V, F, \rho)$ of $N$ for which $H^*$ is one of its executions. First we set $\rho(v) = l(v)$ for each $v \in V$. By lemma 8.6, for each $f_p$, there exists a multiset $M_p$ of paths of $H$ for which $f_p = \sum_{w\in M_p} f_w$. For each path $w = v_1e_1v_2...v_ke_kv_{k+1}$ in $M_p$ we create a condition $b_w$ in $B$ labeled by $p$, i.e. $\rho(b_w) = p$, and put $(v_1, b_w)$ and $(b_w, v_{k+1})$ into $F$. We claim that for each $v \in V$, $|\{(b,v) \in F : \rho(b)=p\}| = \hat{p}(\rho(v))$ and $|\{(v,b) \in F : \rho(b)=p\}| = \check{p}(\rho(v))$. Since each $f_p$ is a $p$-interlaced flow, we have $\sum_{e^t=v} \boldsymbol{pb}_p(e) + \boldsymbol{bb}_p(e) = \hat{p}(l(v))$ and $\sum_{e^s=v} \boldsymbol{bb}_p(e) + \boldsymbol{bf}_p(e) = \check{p}(l(v))$ for each $v \in V$. By lemma 8.7, for each $p \in P$, there exists exactly $\check{p}(l(v))$ paths in $M_p$ whose first vertex is $v$ and exactly $\hat{p}(l(v))$ paths in $M_p$ whose last vertex is $v$. Furthermore for $v = v_\iota$, there are $\check{p}(\iota) = p_0$ minimal conditions, which correspond to paths whose first vertex is $v_\iota$.

It remains to check that $H^*$ is indeed a sequentialization of the causal order $po_\pi = (V, <_\pi, l)$ derived from $\pi$. For that, let $v, v' \in V$ with $v <_\pi v'$. Then by the definition of causal order (5) there exists at least a condition $b$ in $B$ for which $(v, b), (b, v') \in F$. Let $\rho(b) = p$ for some $p \in P$. Then by our construction of $\pi$ from $f$, this condition $b = b_w$ corresponds to a path $w$ in $H$ whose first vertex is $v$ and last is $v'$, what implies that $v < v'$ in $H^* = (V, <, l)$. This proves that $<_\pi \subseteq <$, and thus $H^*$ is a sequentialization of $po_\pi$. This proves the converse direction of item $(i)$. In order to prove the same direction of item $ii$, suppose that for each $e = (v, v') \in E$, we have $\boldsymbol{bb}(p) \geq 1$ for at least one place $p \in P$. Then by the construction of process described above, we have a condition $b_w$ such that $(v, b_w), (b_w, v') \in F$. This implies that the edge $(v, v')$ is also in the causal order derived from $\pi$. And thus $<$ which is the transitive closure of $E$ is included on $<_\pi$. Since the inclusion in the other sense was already proved, we have $< = <_\pi$. □

## 5 Slicing Graph Properties via Seasonings

Any DAG can be cast as the composition of a sequence of unit slices. Conversely, we can use slice graphs in order to compose unit slices and define infinite families of *DAGs*. It would be interesting to generate infinite families of *DAGs* carrying some given property. Being a Hasse diagram is an example of an useful property, since Hasse diagrams are in one to one correspondence with the partial orders they induce. Given a $p/t$-net $N$ we could be interested in generating as well, sets of Hasse diagrams which induced partial order is an execution or a causal order of $N$.

In order to address these problems we introduce in this section the notion of *seasoner*. A seasoner is a second order predicates $Q(\mathbf{S}, R)$ where the first variable $\mathbf{S}$, ranges over unit slices and the second $R$, over relations defined on the edges of $\mathbf{S}$. Roughly speaking, we use seasoners in order to transpose *DAG* properties to the unit slices they are constituted of. This notion will be made more precise below.

A *seasoning* of a slice $\mathbf{S} = (V, E, l)$ is a relation $R : E^\alpha \times X$ which associates values of an arbitrary fixed set $X$ to $\alpha$-tuples of edges of $\mathbf{S}$. A *seasoned slice* is a pair $(\mathbf{S}, R)$ where $\mathbf{S}$ is an slice and $R$ a seasoning of $\mathbf{S}$. A seasoned slice $(\mathbf{S}, R)$ can be composed with a seasoned slice $(\mathbf{S}', R')$ if $\mathbf{S}$ can be composed with $\mathbf{S}'$ and furthermore the values associated by $R$ to each $\alpha$-tuple of out-edges of $\mathbf{S}$ agrees with the values associated by $R'$ to the corresponding $\alpha$-tuple of in-edges of $\mathbf{S}'$ (Fig. 2.*III* and 3). A seasoning of a sequence $\mathbf{S}_1\mathbf{S}_2...\mathbf{S}_n$ of slices is a sequence of relations $R_1R_2...R_n$ where $R_i$ is a seasoning of $\mathbf{S}_i$ for $1 \leq i \leq n$ and such that $(\mathbf{S}_i, R_i)$ can be composed with $(\mathbf{S}_{i+1}, R_{i+1})$ for $1 \leq i \leq n-1$. In this paper we restrict our seasonings to partial functions (view as relations).

**Definition 11 (Seasoner).** *A* seasoner *is a decidable second order predicate $Q(\mathbf{S}, R)$ in which the first variable $\mathbf{S}$ ranges over unit slices, the second $R$ over seasonings of $\mathbf{S}$, and for each $\mathbf{S}$ the set $\{R | Q(\mathbf{S}, R)\}$ is finite and computable. A sequence of unit slices $\mathbf{S}_1\mathbf{S}_2...\mathbf{S}_n$ is $Q$-seasonable, if $\mathbf{S}_1\mathbf{S}_2...\mathbf{S}_n$ has a seasoning $R_1R_2...R_n$ satisfying $Q(\mathbf{S}_i, R_i)$ for each $i$. A DAG $G$ is $Q$-seasonable if each of its unit decompositions $\mathbf{S}_1\mathbf{S}_2...\mathbf{S}_n$ is $Q$-seasonable. A seasoner $Q$ is coherent if for every DAG $G$, the $Q$-seasonability of a unit decomposition $S_1S_2...S_n$ of $G$ implies the $Q$-seasonability of $G$.*

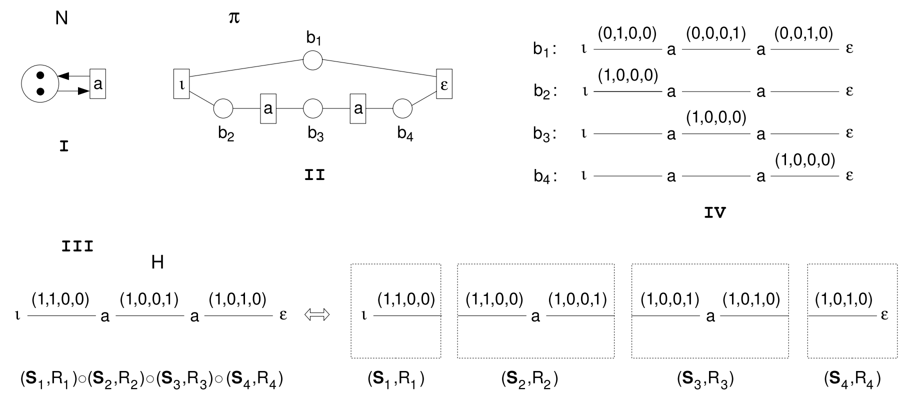


**Fig. 2.** *I*) A $p/t$-net $N$. *II*) A process $\pi$ of $N$. *III*) The hasse diagram $H$ of the causal order derived from $\pi$ toguether with an interlaced flow. $H$ with the flow attached on it can be cast as the composition of a sequence of seasoned slices (section 5). *IV*) The flow attached to $H$ is decomposed into unit flows. One for each condition $b_i$ of $\pi$.

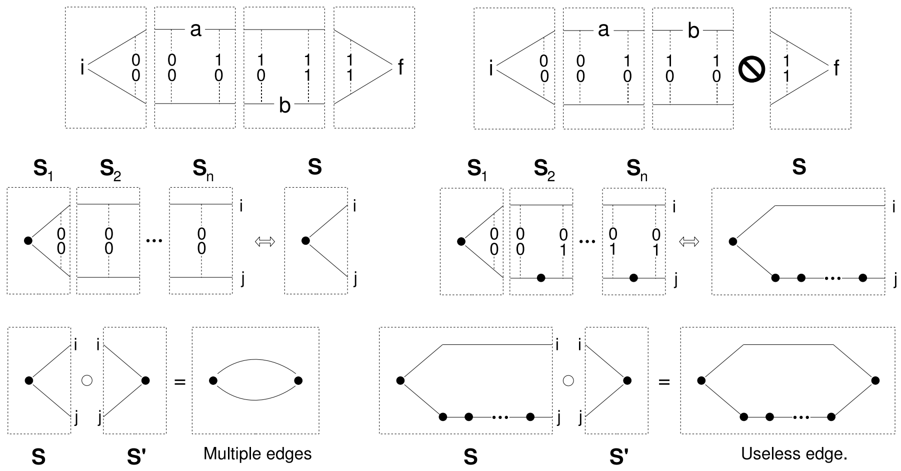


**Fig. 3.** (Left) Hasse seasoning of a unit decomposition Hasse diagram. (Right) If a graph is not a Hasse diagram, none of its unit decompositions can be coherently Hasse-seasoned.

Our interest in seasoners is justified in lemma 12. The first three items provide a sliced characterization of $p$-interlaced flows, executions and causal orders. The fourth provides a sliced characterization of Hasse diagrams.

**Lemma 12 (Seasoners and DAG properties).** *Let $H$ be a Hasse diagram, $G$ a DAG, $N$ a bounded $p/t$-net with bound $k$ and $p$ a place of $N$. Then there exist seasoners $Q_p^k$, $Q_{ex}^N$, $Q_{cau}^N$ and $Q_{\mathcal{H}}$ such that*

*(i) The PO induced by $H$ has a p-interlaced flow with relation to $N$ iff $H$ is $Q_p^k$-seasonable.*
*(ii) The PO induced by $H$ is an execution of $N$ iff $H$ is $Q_{ex}^N$-seasonable.*
*(iii) The PO induced by $H$ is a causal order of $N$ iff $H$ is $Q_{cau}^N$-seasonable.*
*(iv) $G$ is a Hasse diagram with a unique minimal vertex iff $G$ is $Q_{\mathcal{H}}$-seasonable.*

The proof of the first three items of lemma 12, will follow directly from theorem 10 combined with proposition 14 and the auxiliary lemma 15. The proof of item $(iv)$ will be postponed to subsection 5.1.

**Definition 13.** *Let $N = (P, T)$ be a bounded p/t-net with bound $k$, $p$ a place of $N$ and $\mathbf{S} = (V, E, l)$ a standard unit slice with $V = I\dot{\cup}\{v\}\dot{\cup}O$ and $l(v) = t$ for some $t \in T$. Then*

1. *(p-seasoner) Define $Q_p^k(\mathbf{S}, R)$ to be true if $\mathbf{S}$ is standard and $R$ is a function $R : E \to \mathbb{N}^4$ where for each $e \in E$, $R(e) = (\boldsymbol{bb}(e), \boldsymbol{bf}(e), \boldsymbol{pb}(e), \boldsymbol{pf}(e))$ and*
   *(a) $\sum_{e^t=v} \boldsymbol{pb}(e) + \boldsymbol{bb}(e) = \hat{p}(l(v))$,*
   *(b) $\sum_{e^s=v} \boldsymbol{bb}(e) + \boldsymbol{bf}(e) = \check{p}(l(v))$,*
   *(c) $\sum_{e^t=v} \boldsymbol{bf}(e) + \boldsymbol{pf}(e) = \sum_{e^s=v} \boldsymbol{pb}(e) + \boldsymbol{pf}(e)$,*
   *(d) $\sum_{e^s\in I} \boldsymbol{bb}(e) + \boldsymbol{bf}(e) + \boldsymbol{pb}(e) + \boldsymbol{pf}(e) \le k$,*
   *(e) $\sum_{e^t\in O} \boldsymbol{bb}(e) + \boldsymbol{bf}(e) + \boldsymbol{pb}(e) + \boldsymbol{pf}(e) \le k$.*
2. *(Execution seasoner) Define $Q_{ex}^N(\mathbf{S}, R)$ to be true if $R$ is a function $R : P \to (E \to \mathbb{N}^4)$ where for each $p \in P$ $Q_p^k(\mathbf{S}, R(p))$ holds. For each $e \in E$ we let $R(p)(e) = (\boldsymbol{bb}_p(e), \boldsymbol{bf}_p(e), \boldsymbol{pb}_p(e), \boldsymbol{pf}_p(e))$.*
3. *(Causality seasoner) Define $Q_{cau}^N(\mathbf{S}, R)$ to be true iff $Q_{ex}^N(\mathbf{S}, R)$ holds and if for each $e \in E$ there exists a $p \in P$ with $\boldsymbol{bb}_p(e) \ge 1$.*

**Proposition 14.** *Let $N = (P, T)$ be a bounded p/t-net with bound $k$, $p$ a place of $N$, $\mathbf{S} = (V, E, l)$ a unit slice with $V = I\dot{\cup}\{v\}\dot{\cup}O$ and $R$ a $Q_p^k$-seasoning of $\mathbf{S}$. Then*

$$\sum_{e^t\in O} (\boldsymbol{bb} + \boldsymbol{bf} + \boldsymbol{pb} + \boldsymbol{pf})(e) = \check{p}(l(v)) - \hat{p}(l(v)) + \sum_{e^s\in I} (\boldsymbol{bb} + \boldsymbol{bf} + \boldsymbol{pb} + \boldsymbol{pf})(e).$$

**Lemma 15.** *Let $N = (P, T)$ be a bounded p/t-net with bound $k$, $p$ a place of $N$ , $H = (V, E, l)$ a DAG whose induced partial order is an execution of $N$, $\mathbf{S}_1\mathbf{S}_2...\mathbf{S}_n$ an unit decomposition of $H$ where $S_i = (V_i, E_i, l_i)$ with $V_i = I_i \cup \{v_i\} \cup O_i$ and $R_1R_2...R_n$ a $Q_p^k$-seasoning of $\mathbf{S}_1\mathbf{S}_2...\mathbf{S}_n$. Then for $1 \le i \le n-1$ the following equality holds:*

$$\sum_{e^t\in O_i} (\boldsymbol{bb}_i + \boldsymbol{bf}_i + \boldsymbol{pb}_i + \boldsymbol{pf}_i)(e) = \sum_{1\le j\le i} (\check{p}(l_j(v_j)) - \hat{p}(l_j(v_j)))$$

*Proof.* The proof is by induction on $i$. Since the partial order induced by $H$ is an execution of $N$, $H$ has a unique minimal vertex which is labeled by $\iota$. Thus the in-frontier of $S_1$ is empty and its unique center vertex is $v_\iota$ which is labeled by $\iota$. By equation (c) and (b) of definition 13.1 (p-seasoner), $\sum_{e^s=v_\iota}(\boldsymbol{bb}_1 + \boldsymbol{bf}_1 + \boldsymbol{pb}_1 + \boldsymbol{pf}_1)(e) = \sum_{e^s=v_\iota}(\boldsymbol{bb} + \boldsymbol{bf})(e) = \check{p}(\iota) = m_0(p)$. Thus the lemma is valid for the base case. Now suppose it is valid for $i$ with $1 \leq i < n-1$. We prove it is valid for $i+1$ as well. By the rule of composition of seasoned slices, the sum over the edges touching the in-frontier of $S_{i+1}$ must be equal to the sum over the edges touching out-frontier of $S_i$. Thus we have

$$\sum_{e^s \in I_{i+1}} (\boldsymbol{bb}_{i+1} + \boldsymbol{bf}_{i+1} + \boldsymbol{pb}_{i+1} + \boldsymbol{pf}_{i+1})(e) = \sum_{1 \leq j \leq i} (\check{p}(l_j(v_j)) - \hat{p}(l_j(v_j))).$$

This equation together with proposition 14 imply that

$$\sum_{e^t \in O_{i+1}} (\boldsymbol{bb}_{i+1} + \boldsymbol{bf}_{i+1} + \boldsymbol{pb}_{i+1} + \boldsymbol{pf}_{i+1})(e) = \sum_{1 \leq j \leq i+1} (\check{p}(l_j(v_j)) - \hat{p}(l_j(v_j)))$$

what proves the lemma. □

**Proof of items (i)-(iii) of lemma 12.** Let $\mathbf{S}_1\mathbf{S}_2...\mathbf{S}_n$ be a unit decomposition of a Hasse diagram $H$ and $R_1R_2...R_n$ be a $Q_p^k$-seasoning of $\mathbf{S}_1\mathbf{S}_2...\mathbf{S}_n$. Then $(\mathbf{S}, R_1) \circ (\mathbf{S}_2, R_2) \circ ... \circ (\mathbf{S}_n, R_n)$ is equal to $H$ with a $p$-interlaced flow associated to its edges (Fig. 2.*III*). Conversely, if it is possible to associate a $p$-interlaced flow $f_p$ to a Hasse diagram $H$, then to any of its unit decompositions $\mathbf{S}_1\mathbf{S}_2...\mathbf{S}_n$, we can associate a seasoning $R_1R_2...R_n$ where $R_i(e^i) = f(e)$ if and only if $e^i$ is the sliced part of $e$ which lies on $\mathbf{S}_i$ (Fig. 2.*III*). Since $f$ is interlaced and satisfies conditions $(IN)$ and $(OUT)$ of definition 9, equations (a),(b) and (c) of definition 13 are satisfied. Since $N$ is bounded by $k$, lemma 15 assures that equations (d) and (b) of definition 13.*I* are satisfied as well. This last claim follows from the fact that $\sum_{1 \leq j \leq i}(\check{p}(l_j(v_j)) - \hat{p}(l_j(v_j)))$ corresponds to the number of tokens at place $p$ after the firing of transitions $l_1(v_1)l_2(v_2)...l_k(v_i)$. Items (ii) and (iii) of lemma 12 follow analogously. The unique difference is that instead of single interlaced flows we attach sets of interlaced flows to each edge of $H$: one flow for each place $p \in P$. □

## 5.1 Slicing Hasse Diagrams

**Definition 16 (Hasse Seasoning).** *Let* $\mathbf{S} = (V, E, l)$ *be a unit standard slice whose unique center vertex is* $v$*. Then a partial function* $\mathcal{H} : E^2 \rightarrow \{0,1\}^2$ *is a* Hasse seasoning *of* $\mathbf{S}$ *if the following conditions can be verified for every* $e_1, e_2 \in E$*:*

1. $\mathcal{H}(e_1e_2)$ *is not defined if and only if* $(e_1{=}e_2)$ *or* $(e_1^t{=}e_2^s)$ *or* $(e_1^s{=}e_2^t)$
2. *If* $\mathcal{H}(e_1e_2) = xy$ *then* $\mathcal{H}(e_2e_1) = yx, \quad x, y \in \{0,1\}$
3. *If* $e_1^t{=}e_2^t$ *then* $\mathcal{H}(e_1e_2) = 11$
4. *If* $e_1^s{=}e_2^s$ *then* $\mathcal{H}(e_1e_2) = 00$
5. *If* $e_1^s \in I$ *and* $e_1^t \in O$ *and* $e_2^s = v$ *then* $\mathcal{H}(e_1e_2) \in \{01, 11\}$ *and*
   $\mathcal{H}(e_1e_2) = 01$ *iff* $(\exists e, e^t{=}v)(\mathcal{H}(e_1e) \in \{00, 01\})$

*We define the Hasse seasoner* $Q_{\mathcal{H}}(\mathbf{S}, R)$ *to be true if* $\mathbf{S}$ *is a unit slice and* $R$ *is a Hasse seasoning of* $\mathcal{SG}$*. See figure 3.*

We say that edges $i, j$ touching the in-frontier of a slice $\mathbf{S}$ *converge* if $i^t = j^t$. Now let $i, j$ touch the out frontier of $S$. Then we say that $i, j$ diverge if $i^s = j^s$ and that $i$ is shorter than $j$ if there is a path with at least one edge from $i^s$ to $j^s$. Some intuition about definition 16 can be gathered by the understanding of the next three propositions, which we will employ in the proof of lemma 12.$IV$.

**Proposition 17.** *Let $\mathbf{S}_1\mathbf{S}_2...\mathbf{S}_n$ be an unit decomposition of a slice $\mathbf{S}$ with a unique maximal vertex, $\mathcal{H}_1\mathcal{H}_2...\mathcal{H}_n$ a Hasse seasoning of $\mathbf{S}_1\mathbf{S}_2...\mathbf{S}_n$ and $i, j \in O^n$ be two out-frontier vertices of $\mathbf{S}_n$. Then*

– $\mathcal{H}_n(i, j) = 00 \Leftrightarrow i$ *and* $j$ *diverge in* $S$
– $\mathcal{H}_n(i, j) = 01 \Leftrightarrow i$ *is shorter than* $j$ *in* $S$

*See figure 3.*

**Proposition 18.** *Let $S, S'$ be two (not necessarily unit ) slices such that $S$ can be composed with $S'$. And let $i, j$ be two convergent in-frontier vertices in $S'$. If $i, j$ are divergent in $S$ or if $i$ is shorter than $j$ in $S$ then $S \circ S'$ is not transitive reduced.*

**Proposition 19.** *Let $\mathbf{S}$ and $\mathbf{S}'$ be two unit slices such that $\mathbf{S}$ can be composed with $\mathbf{S}'$, $\mathcal{H}$ be an Hasse seasoning of $\mathbf{S}$. If there is no Hasse seasoning $\mathcal{H}'$ of $\mathbf{S}'$ such that $(\mathbf{S}, \mathcal{H})$ can be composed with $(\mathbf{S}', \mathcal{H}')$, then there exist $i, j$ such that $\mathcal{H}(i, j) \in \{00, 01\}$ and $i, j$ converge in $\mathbf{S}'$.*

We finish this section with a proof of item (iv) of lemma 12.

**Proof of item (iv) of lemma 12:** Let $\mathbf{S} = \mathbf{S}_1 \circ \mathbf{S}_2 \circ ... \circ \mathbf{S}_n$ be a slice with a unique minimal vertex. Suppose that $\mathbf{S}$ is not a Hasse diagram and that it is $Q_{\mathcal{H}}$-seasonable. Then there exists a $Q_{\mathcal{H}}$-seasoning $\mathcal{H}_1\mathcal{H}_2...\mathcal{H}_n$ of $\mathbf{S}_1\mathbf{S}_2...\mathbf{S}_n$. Since $\mathbf{S}$ is not a Hasse diagram, there exists two vertices $v$ and $v'$ which are connected either by multiple edges, either by an edge and a path of length greater than one. Let $v'$ be in $\mathbf{S}_{k+1}$. Then there are $i, j$ such that $i, j$ converge in $\mathbf{S}_{k+1}$. Furthermore either $i, j$ diverge or $i$ is shorter than $j$ in $\mathbf{S}_1 \circ \mathbf{S}_2 \circ ... \circ \mathbf{S}_k$. By proposition 17, we have that $\mathcal{H}_k(i, j) \in \{00, 01\}$. By proposition 19, there is no $\mathcal{H}_{k+1}$ such that $(\mathbf{S}_k, \mathcal{H}_k)$ can be composed with $(\mathbf{S}_{k+1}, \mathcal{H}_{k+1})$. This contradicts the assumption that $\mathbf{S}_1 \circ \mathbf{S}_2 \circ ... \circ \mathbf{S}_n$ is $Q_{\mathcal{H}}$-seasonable.

In order to prove the converse, let $\mathbf{S}_1\mathbf{S}_2...\mathbf{S}_k$ be the greatest prefix of $\mathbf{S}_1\mathbf{S}_2...\mathbf{S}_n$ which is $Q_{\mathcal{H}}$-seasonable and let $\mathcal{H}_1\mathcal{H}_2...\mathcal{H}_k$ be one of its Hasse seasonings. Suppose that $k < n$. By proposition 19, there exists integers $i, j$ such that $\mathcal{H}_k(i, j) \in \{00, 01\}$ and $i, j$ converge in $\mathbf{S}_{k+1}$. By proposition 17, either $i, j$ diverge, either $i$ is shorter than $j$ in $\mathbf{S}_1 \circ \mathbf{S}_2 \circ ... \circ \mathbf{S}_k$. This implies, by proposition 18 that $\mathbf{S}_1 \circ ... \circ \mathbf{S}_k \circ \mathbf{S}_{k+1}$ is not transitive reduced, and so neither is $S_1 \circ ... \circ S_n$. □

## 6 Filters

In this section we introduce *filters*. A filter is a function $filter(\mathcal{SG}, Q)$ which takes as parameters a slice graph and a coherent seasoner, and returns a slice graph whose graph language consists exactly on the $Q$-seasonable graphs of the graph language of $\mathcal{SG}$. In other words, it filters out from $\mathcal{L}_G(\mathcal{SG})$ every graph that is not $Q$-seasonable. In particular, if $Q = Q_{\mathcal{H}}$ is the Hasse seasoner defined in definition 16, then $\mathcal{L}_G(filter(\mathcal{SG}, Q_{\mathcal{H}}))$ is exactly the set of Hasse diagrams in $\mathcal{L}_G(\mathcal{SG})$. Other concrete examples of filters will be found in section 7 where we state our main results. In this section we study filters and their properties in a rather general setting.

**Definition 20 (Filter).** *Let $\mathcal{SG} = (\mathcal{V}, \mathcal{E}, \mathcal{S})$ be a slice graph where $\mathcal{S} : V \rightarrow \Sigma_{\mathbb{S}}$ and let $Q$ be a seasoner. Then the $Q$-filter of $\mathcal{SG}$ is the slice graph $filter(\mathcal{SG}, Q) = (\mathcal{V}^f, \mathcal{E}^f, \mathcal{S}^f)$ where*

$$\mathcal{V}^f = \bigcup_{v \in \mathcal{V}} \{v_R | Q(\mathcal{S}(v), R)\} \qquad \mathcal{S}^f : \mathcal{V}^f \rightarrow \Sigma_{\mathbb{S}} \qquad \mathcal{S}^f(v_R) = \mathcal{S}(v)$$

$$\mathcal{E}^f = \{(v_R, v'_{R'}) | (v, v') \in \mathcal{E} \textit{ and } (\mathcal{S}(v), R) \textit{ can be composed with } (\mathcal{S}(v'), R')\}$$

**Lemma 21.** *Let $Q$ be a seasoner. Then $\mathbf{S}_1\mathbf{S}_2...\mathbf{S}_n \in \mathcal{L}(filter(\mathcal{SG}, Q))$ if and only if $\mathbf{S}_1\mathbf{S}_2...\mathbf{S}_n \in \mathcal{L}(\mathcal{SG})$ and $\mathbf{S}_1\mathbf{S}_2...\mathbf{S}_n$ is $Q$-seasonable.*

*Proof.* Let $\mathbf{S}_1\mathbf{S}_2...\mathbf{S}_n$ be the label of a walk $v^1_{R_1}v^2_{R_2}...v^n_{R_n}$ from an initial to a final vertex on $filter(\mathcal{SG}, Q)$. Then $R_1R_2..R_n$ is a $Q$-seasoning of $\mathbf{S}_1\mathbf{S}_2...\mathbf{S}_n$, which labels the walk $v^1v^2...v^n$ in $\mathcal{SG}$ as well. Conversely, suppose that $R_1R_2...R_n$ is a $Q$-seasoning of the label $\mathbf{S}_1\mathbf{S}_2...\mathbf{S}_n$ of a walk $v^1v^2...v^n$ in $\mathcal{SG}$. Then $\mathbf{S}_1\mathbf{S}_2...\mathbf{S}_n$ also labels the walk $v^1_{R_1}v^2_{R_2}...v^n_{R_n}$ in $filter(\mathcal{SG}, Q)$.

**Lemma 22 (Filter Lemma).** *Let $Q$ be a coherent seasoner and $\mathcal{SG}$ a slice graph. Then there exists an effectively computable slice graph $filter(\mathcal{SG}, Q)$ such that $H \in \mathcal{L}_G(filter(\mathcal{SG}, Q))$ iff $H \in \mathcal{L}_G(\mathcal{SG})$ and $H$ is $Q$-seasonable.*

*Proof.* It is clear that the filter of definition 20 is computable, since by definition 11, for each $\mathbf{S}$ the set $\{R|Q(\mathbf{S}, R)\}$ is finite and computable. If $H \in \mathcal{L}_G(filter(\mathcal{SG}, Q))$ then one of its unit decompositions $\mathbf{S}_1\mathbf{S}_2...\mathbf{S}_n$ is in $\mathcal{L}(\mathcal{SG}, Q)$. By lemma 21 $\mathbf{S}_1\mathbf{S}_2...\mathbf{S}_n$ is $Q$-seasonable, and since $Q$ is coherent, $H$ is $Q$-seasonable. Now suppose $H$ is $Q$-seasonable and that it is in $\mathcal{L}_G(\mathcal{SA})$. Then one of its unit decompositions $\mathbf{S}_1\mathbf{S}_2...\mathbf{S}_n$ is in $\mathcal{L}(\mathcal{SA})$ and by lemma 21, $\mathbf{S}_1\mathbf{S}_2...\mathbf{S}_n$ is in $\mathcal{L}(filter(\mathcal{SG}, Q))$, what implies that $H \in \mathcal{L}_G(\mathcal{SG}, Q)$. □

The next lemma will be useful in the proofs our main results in the next section.

**Lemma 23 (Decidability of Equality and Emptiness).** *Let $\mathcal{SG}$ be a slice-graph and $Q$ a seasoner. Then it is decidable whether $\mathcal{L}_G(filter(\mathcal{SG}, Q)) = \emptyset$ as well as whether $\mathcal{L}_G(\mathcal{SG}) = \mathcal{L}_G(filter(\mathcal{SG}, Q))$.*

*Proof.* It is well known that the set of labels of walks on a given graph is a regular set. Thus for any slicegraph $\mathcal{SG}$, $\mathcal{L}(\mathcal{SG})$ is a regular language over an alphabet of slices. The idea of our proof is to reduce the emptiness and inclusion tests of graph languages of filtered slice graphs, to the emptiness and inclusion tests of their slice languages, and then take advantage of well known decidability results for regular sets. In this sense, the decidability of emptiness follows easily, since $\mathcal{L}_G(filter(\mathcal{SG}, Q)) = \emptyset \Leftrightarrow \mathcal{L}(filter(\mathcal{SG}, Q)) = \emptyset$. With relation to the equality test, we claim that $\mathcal{L}_G(\mathcal{SG}) = \mathcal{L}_G(filter(\mathcal{SG}, Q)) \Leftrightarrow \mathcal{L}(\mathcal{SG}) = \mathcal{L}(filter(\mathcal{SG}, Q)$. First we note that $\mathcal{L}(filter(\mathcal{SG}, Q)) \subseteq \mathcal{L}(\mathcal{SG})$ for any $Q$. Thus we just have to prove that $\mathcal{L}(\mathcal{SG}) \subseteq \mathcal{L}(filter(\mathcal{SG}, Q)) \Leftrightarrow \mathcal{L}_G(\mathcal{SG}) \subseteq \mathcal{L}_G(filter(\mathcal{SG}, Q)$. It is easy to see that for any slice graphs $\mathcal{SG}', \mathcal{SG}''$, $\mathcal{L}(\mathcal{SG}') \subseteq \mathcal{L}(\mathcal{SG}'')$ implies that $\mathcal{L}_G(\mathcal{SG}') \subseteq \mathcal{L}_G(\mathcal{SG}'')$, what proves the $\rightarrow$ direction. The converse however is not valid for any two slice graphs, but as we show now it is whenever one of them is a filtered version of the other. Let $H$ be a DAG in $\mathcal{L}_G(\mathcal{SG})$ and suppose that $\mathcal{L}_G(\mathcal{SG}) \subseteq \mathcal{L}_G(filter(\mathcal{SG}, Q))$. Thus by lemma 22, any unit decomposition of $H$ is $Q$-seasonable. Let $\mathbf{S}_1\mathbf{S}_2...\mathbf{S}_n$ be a unit decomposition of $H$ in $\mathcal{L}(\mathcal{SG})$. Then since $\mathbf{S}_1\mathbf{S}_2...\mathbf{S}_n$ is $Q$-seasonable, lemma 21 implies that $\mathbf{S}_1\mathbf{S}_2...\mathbf{S}_n \in \mathcal{L}(filter(\mathcal{SG}, Q))$, what proves that $\mathcal{L}(\mathcal{SG}) \subseteq \mathcal{L}(filter(\mathcal{SG}, Q))$. □

## 7 Main Results

In this section we will use seasoners and filters defined in the previous sections in order to prove our main results. We start with the verification theorem 24, which states that for any bounded $p/t$-net $N$, we are able to decide either whether the partial order language generated by a Hasse diagram generator is included on the partial order language of a $p/t$-net $N$, or whether their intersection is empty. These tests are decidable for both the execution and for the causal semantics. The variable *sem* assumes the value *ex* for the execution semantics and the value *cau* for the causality semantics.

**Theorem 24 (Verification Theorem).** *Let $N$ be a bounded $p/t$-net and $\mathcal{HG}$ a Hasse diagram generator. Then*

*(i)* $\mathcal{L}_{PO}(\mathcal{HG}) \subseteq \mathcal{L}_{sem}(N) \Leftrightarrow \mathcal{L}_{PO}(\mathcal{HG}) = \mathcal{L}_{PO}(filter(\mathcal{HG}, Q^N_{sem}))$
*(ii)* $\mathcal{L}_{PO}(\mathcal{HG}) \cap \mathcal{L}_{sem}(N) = \emptyset \Leftrightarrow \mathcal{L}_{PO}(filter(\mathcal{HG}, Q^N_{sem})) = \emptyset$

*Furthermore the equalities on the right hand side of the $\Leftrightarrow$ symbol are decidable.*

*Proof.* Since $\mathcal{HG}$ is a hasse diagram generator, questions about its partial order language can be safely translated to analogous questions about its graph language. Formally, it is enough to prove that

(i) $\mathcal{L}_G(\mathcal{HG}) \subseteq \mathcal{L}_{sem}(N) \Leftrightarrow \mathcal{L}_G(\mathcal{HG}) = \mathcal{L}_G(filter(\mathcal{HG}, Q^N_{sem}))$
(ii) $\mathcal{L}_G(\mathcal{HG}) \cap \mathcal{L}_{sem}(N) = \emptyset \Leftrightarrow \mathcal{L}_G(filter(\mathcal{HG}, Q^N_{sem})) = \emptyset$

and that both the equality test and the emptiness test on the right side of the $\Leftrightarrow$ symbols of statements $(i)$ and $(ii)$ respectively, are decidable. The decidability of these tests was already proved on lemma 23 for general filters. Now by lemma 12, the partial order induced by any Hasse diagram $H$ in $\mathcal{L}_G(\mathcal{SG})$, is an execution of $N$ if and only if $H$ is $Q^N_{ex}$-seasonable. Analogously, $H$ is a causality diagram of $N$ if and only if it is $Q^N_{cau}$ seasonable. Thus, using the filter lemma (22),

$H \in \mathcal{L}_G(filter(\mathcal{HG}, Q_{ex}^N))$ iff $H \in \mathcal{L}_G(\mathcal{HG})$ and the partial order induced by $H$ is an execution of $N$. Similarly, $H \in \mathcal{L}_G(filter(\mathcal{SG}, Q_{cau}^N))$ iff $H \in \mathcal{L}_G(\mathcal{SG})$ and the partial order induced by $H$ is a causal order of $N$. Thus the statements above hold and the verification theorem is proved. $\square$

**Definition 25 (Slicewidth).** *We define the* slicewidth $sw(\mathbf{S})$ *of a slice* $\mathbf{S} = (V, E, l)$ *as the size of its greatest frontier, i.e. if* $V = I \cup C \cup O$ *then* $sw(\mathbf{S}) = \max\{|I|, |O|\}$. *The slicewidth of an unit decomposition* $\mathbf{S}_1\mathbf{S}_2...\mathbf{S}_n$ *is the slice width of its widest slice:* $sw(\mathbf{S}_1\mathbf{S}_2...\mathbf{S}_n) = \max\{sw(\mathbf{S}_i)\}$. *The slicewidth of a DAG* $H$ *is the slicewidth of its thinnest unit decomposition:* $sw(H) = \min\{sw(S_1S_2...S_n) | S_1S_2...S_n \in ud(H)\}$.

We contrast our notion of slicewidth with that of width of a partial order which is defined in terms of chains. A chain in a partial order *po* is a totally ordered subset of elements of *po*. A cut of *po* is a set of pairwise non-comparable elements of *po*. The width of a partial order is defined as the minimal number chains in which its elements can be partitioned.

**Lemma 26 (Dilworth 1950 [Dil50]).** *In a partial order the maximal cardinality of a cut is equal to the minimal number of chains into which the partial order can be partitioned.*

The slicewidth of partial orders can be arbitrarily greater than their width. Chains for example, always have width equal to 1, while their slicewidth grows quadratically with their length. However, the slicewidth of the Hasse diagram $H$ of a partial order *po* is always equal to the width of *po*, i.e, $sw(H) = w(po)$. The following proposition follows from this observation, from lemma 26 and from the fact that at any marking of a bounded $p/t$-net $N = (P, T)$ with bound $\beta_N$, at most $\beta_N|P|$ transitions can be fired concurrently.

**Proposition 27.** *Let* $N = (P, T)$ *be a bounded p/t-net. Then the slicewidth* $sw(H)$ *of any Hasse diagram which induced partial order is an execution (or causal order) of* $H$ *is bounded by* $\beta_N|P|$.

**Definition 28 (Bounded Width Slicegraph).** *Let* $n \in \mathbb{N}$. *Then we denote* $\mathcal{SG}_n$ *the slice graph* $(\mathcal{V}, \mathcal{E}, \mathcal{S})$ *over* $\Sigma_{\mathbb{S}}$ *where*

$$\Sigma_{\mathbb{S}} = \{\mathbf{S} | \mathbf{S} \textit{ is a unit standard slice over } T \cup \{\iota, \epsilon\}\ , w(S) \leq n\}$$

$$\mathcal{V} = \{v_{\mathbf{S}} | \mathbf{S} \in \Sigma_{\mathbb{S}}\} \qquad \mathcal{S} : \mathcal{V} \to \Sigma_{\mathbb{S}} \qquad \mathcal{S}(v_{\mathbf{S}}) = \mathbf{S}$$

$$\mathcal{E} = \{(v_{\mathbf{S}}, v'_{\mathbf{S}'}) \in \mathcal{V} \times \mathcal{V} \mid \mathbf{S} \textit{ can be composed with } \mathbf{S}'\}$$

**Proposition 29.** *A DAG* $H$ *belongs to* $\mathcal{L}_G(\mathcal{SG}_n)$ *if and only if it has an unique minimal vertex which is labeled with* $\iota$, *an unique maximal vertex which is labeled with* $\epsilon$, *and its slice width* $sw(H)$ *is less than or equal to* $n$.

Our second main result 30, the expressibility theorem, states that for any given bounded Petri net $N = (P, T)$ there exist effectively computable Hasse diagram generators whose partial order languages match the causal behavior and the execution behaviors of $N$. In order to prove this theorem we will make use of the Hasse seasoner and of the Hasse filter, which where not employed in the verification theorem because there we assumed that the slice graphs we where working with were Hasse diagram generators.

Let $\beta_N$ ne the bound of $N$. Then any Hasse diagram whose induced partial order is an execution or causal order of $N$ can be decomposed into slices of width at most $\beta_N|P|$. In order to prove theorem 30, first we consider a slice graph $\mathcal{SG}^N$ which generates all *DAGs* of width bounded by $\beta_N|P|$. Then we filter $\mathcal{SG}^N$ using the Hasse seasoner $Q_{\mathcal{H}}$, in order to obtain the Hasse diagram generator $filter(\mathcal{SG}^N, Q_{\mathcal{H}})$ which generates precisely the Hasse diagrams of width bounded by $\beta_N|P|$. Subsequently we apply another filter, using $Q^N_{sem}$ in order to get a Hasse diagram generator whose graph language consists precisely on the Hasse diagrams of executions of $N$ ($sem = ex$), or of causal orders of $N$ ($sem = cau$).

**Theorem 30 (Expressibility Theorem).** *Let $N = (P, T)$ be a bounded p/t-net and $\mathcal{HG}^N_{sem} = filter(filter(\mathcal{SG}^N, Q_{\mathcal{H}}), Q^N_{sem})$. Then $\mathcal{L}_{sem}(N) = \mathcal{L}_{PO}(\mathcal{HG}^N_{sem})$.*

*Proof.* Let $n = \beta_N|P|$ and $\mathcal{SG}^N = \mathcal{SG}_n$. By lemma 12, $H$ is a Hasse diagram iff it is $Q_{\mathcal{H}}$-seasonable. Thus by proposition 29, and by the filter lemma 22, $H$ belongs to $\mathcal{L}_G(filter(\mathcal{SG}^N, Q_{\mathcal{H}}))$ iff $H$ is a Hasse diagram whose width is bounded by $n$. Filtering again we have that $H \in \mathcal{L}_G(filter(filter(\mathcal{SG}^N, Q_{\mathcal{H}}), Q_{sem}))$ if and only if $H$ is the Hasse diagram of an execution ($sem = ex$) or causal order ($sem = cau$) of $N$ with $sw(H) \leq n$. Since by proposition 27 the slicewidth of the Hasse diagram of any execution or causal order of $N$ is bounded by $n$, it follows that every execution ($sem = ex$) or causal order ($sem = cau$) of $N$ belongs to $\mathcal{L}_G(filter(filter(\mathcal{SG}^N, Q_{\mathcal{H}}), Q_{sem}))$. This is enough to prove the theorem, since $filter(filter(\mathcal{SG}^N, Q_{\mathcal{H}}), Q_{sem})$ is a Hasse diagram generator. □

As a corollary of theorems 24 and 30. we have the decidability of the inclusion of the causal and execution languages of any two given bounded nets $N_1, N_2$.

**Corollary 31 (Comparison of $p/t$-nets Partial Order Langauges).** *Let $N_1, N_2$ be two bounded Petri nets. Then it is decidable whether $\mathcal{L}_{sem}(N_1) \subseteq \mathcal{L}_{sem}(N_2)$.*

*Proof.* Using the expressibility theorem 30, compute $\mathcal{SG}_{sem}(N_1)$. Then employing the verification theorem 24, test whether $\mathcal{L}_{PO}(\mathcal{SG}_{sem}(N_1)) \subseteq \mathcal{L}_{sem}(N_2)$. □

## 8 Synthesis

Another potential useful application for our slice graphs is the synthesis of $p/t$-nets from infinite families of scenarios. Here we restrict ourselves to the synthesis of $k$-bounded Petri nets for a given $k$. The synthesis problem can be formally stated as follows: Given a Hasse diagram generator $\mathcal{HG}$ and a semantics $sem \in \{ex, cau\}$ construct a $k$-safe $p/t$-net $N$ whose language $\mathcal{L}_{sem}(N)$ minimally includes $\mathcal{L}_{PO}(\mathcal{HG})$. Here minimal inclusion means that given any other

$k$-safe $p/t$-net $N'$, the chain of inclusions $\mathcal{L}_{PO}(\mathcal{HG}) \subseteq \mathcal{L}_{sem}(N') \subseteq \mathcal{L}_{sem}(N)$ implies that $\mathcal{L}_{sem}(N) = \mathcal{L}_{sem}(N')$. In this work we do not address the question of whether the behavior of the constructed net actually matches the partial order behavior specified by the slice graph. Our approach for the synthesis problem starts with the definition of $k$-safe regions for slice graphs in connection with the abstract regions for possible infinite partial order languages defined in [LBD08].

**Definition 32 ($k$-safe Region for Hasse Diagram Generators).** *Let $\mathcal{HG}$ be a Hasse diagram generator and $T$ a set of transitions. Then a $k$-safe region of $\mathcal{HG}$ with relation to $T$ is a place $p = (p_0, \hat{p}, \check{p})$ over $T$ such that $\mathcal{L}(\mathcal{HG}) = \mathcal{L}(filter(\mathcal{HG}, Q_p^k))$. We denote $\mathcal{R}_k(\mathcal{HG})$ the set of all $k$-safe regions of $\mathcal{HG}$.*

Intuitively, a $k$-safe region of $\mathcal{HG}$ is a $k$-safe place for which all partial orders generated by $\mathcal{HG}$ have a $p$-interlaced flow. We note that the set of $k$-safe places is finite for a given $k$. It turns out that the union of all these regions gives rise to a $k$-safe $p/t$-net with the unique minimal $k$-safe behavior with relation to $\mathcal{HG}$.

**Theorem 33 (Synthesis for the Execution Semantics).** *Let $\mathcal{HG}$ be a Hasse diagram generator and $k \geq 1$. Then if $\mathcal{R}_k(\mathcal{HG}) \neq \emptyset$, the net*

$$\mathcal{N}_{ex}^k(\mathcal{HG}) = \bigcup_{p \in \mathcal{R}_k(\mathcal{HG})} (\{p\}, T)$$

*is a $k$-safe net with minimal execution behavior with relation to $\mathcal{L}_{PO}(\mathcal{HG})$.*

*Proof.* By lemmas 12.(i) and 22, for each region $p$, each Hasse diagram in $\mathcal{L}_G(\mathcal{HG})$ has a $p$-interlaced flow. Thus by theorem 10, each partial order in $\mathcal{L}_{PO}(\mathcal{HG})$ is an execution of $\mathcal{N}_{ex}^k(\mathcal{HG})$. Since the addition of a region can at most restrict the execution behavior of a net, and since all possible regions are contained in $\mathcal{N}_{ex}^k$, $\mathcal{L}_{PO}(\mathcal{N}_{ex}^k)$ includes $\mathcal{L}_{PO}(\mathcal{HG})$ minimally.

We note that our notion of minimal behavior depends on the bound $k$. Thus theorem 33 does not exclude the existence of a $k'$-safe $p/t$-net $N'$ with $k' \geq k$ whose partial order behavior is strictly included on the partial order behavior of $N_{ex}^k(\mathcal{HG})$ but for which $\mathcal{L}_{PO}(\mathcal{HG}) \subseteq \mathcal{L}_{ex}(N')$.

With relation to the causal semantics, we impose a bound on the number of copies of each place on the net. We say that a $p/t$-net $N$ is $(k,r)$-safe if it is $k$-safe and if it contains at most $r$ copies of each place of $N$. This restriction is not necessary for the execution semantics because repeated places do not interfere in the execution behavior of $p/t$-nets. However they do interfere on the causal behavior of $p/t$-nets. Another particularity of the causal semantics is the fact that the uniqueness of the minimal behavior, is not assured.

**Theorem 34.** *Let $\mathcal{HG}$ be a slice graph and $k, r \geq 1$. Then the set of $(k,r)$-safe $p/t$-nets, with minimal causal behavior with relation to $\mathcal{L}_{PO}(\mathcal{HG})$, is computable.*

*Proof.* Any $(k, r)$-safe net which causal behavior includes $\mathcal{L}_{PO}(\mathcal{HG})$ must be a multiset of $k$-safe regions of $\mathcal{HG}$ in which each region appears at most $r$ times. Let $\mathcal{M}_{k,r}(\mathcal{HG})$ be the set of all such multisets whose unions give rise to legal nets, i.e., for which every transition in $T$ takes tokens of some place and puts on some other. Since $\mathcal{R}_k(\mathcal{HG})$ is finite, so is $\mathcal{M}_{k,r}(\mathcal{HG})$. Using the verification theorem 24 we can compute the subset $\mathcal{C}_{k,r}(\mathcal{HG})$ of all $(k, r)$-safe $p/t$-nets of $\mathcal{M}_{k,r}(\mathcal{HG})$ whose causal behavior includes $\mathcal{L}_{PO}(\mathcal{HG})$. By using corollary 31 we can partially order $\mathcal{C}_{k,r}(\mathcal{HG})$ with relation to inclusion of their causal behavior. The nets with minimal causal behavior with relation to $\mathcal{L}_{PO}(\mathcal{HG})$ are the minimal elements of this ordering. □

## 9 Final Comments and Future Directions

Most of our results where stated in terms of Hasse diagram generators. It turns out that for a matter of flexibility, it might be convenient to allow in the graph language of a slice graph the presence of DAGs which are not transitive reduced. Edges which do not belong to the transitive reduction of such DAGs, could be used to highlight particular causal dependences between some of their events. But since the ordering relation between events is what matters when leading with formal analysis, it would be convenient to develop a method to transform an arbitrary slice graph into a Hasse diagram generator with identical partial order language. This problem, which is formalized below, is topic of our current research.

*Problem 35.* Given an arbitrary slice graph $\mathcal{SG}$ compute a Hasse diagram generator $\mathcal{HG}$ with $\mathcal{L}_{PO}(\mathcal{SG}) = \mathcal{L}_{PO}(\mathcal{HG})$.

By the results in [BW00] any bounded $p/t$-net can be transformed in a one-safe labeled $p/t$-net with the same partial order behavior. There, the result is stated with relation to the causal semantics, but it holds for the execution semantics as well. It would be interesting to reverse the direction of this question. That means, given a one safe labeled $p/t$-net $N_O$ we would be interested in synthesizing an unlabeled $p/t$-net with identical partial order behavior, if it exists. By an adaptation of theorem 30, this question can be reduced to the synthesis of $p/t$-nets from slice graphs. Indeed from $N_O$, we derive its Hasse diagram generator $\mathcal{HG}_{sem}(N_O)$ ignoring the labels of the transitions. After, on each of its slices, we relabel the center vertices with the labels of the transitions. Then we ask for the synthesis of an unlabeled net $N$ whose causal or execution behavior minimally includes the set of partial orders generated by $\mathcal{HG}_{sem}(N_O)$. The advantage of this approach is that $\mathcal{L}_{PO}(\mathcal{HG}_{cau}(N_O))$ is prefix closed and $\mathcal{L}_{PO}(\mathcal{HG}_{ex}(N_O))$ is prefix and sequentialization closed. These closedness conditions are essential if we are aiming to test whether the partial order language of the synthesized net precisely matches the partial order language of a given Hasse diagram generator respectively with the causal and the execution semantics.

## References

[BD96] Badouel, E., Darondeau, P.: Theory of regions. In: Reisig, W., Rozenberg, G. (eds.) APN 1998. LNCS, vol. 1491, pp. 529–586. Springer, Heidelberg (1998)

[BDJL06] Bergenthum, R., Desel, J., Juhás, G., Lorenz, R.: Can I execute my scenario in your net? viptool tells you! In: Donatelli, S., Thiagarajan, P.S. (eds.) ICATPN 2006. LNCS, vol. 4024, pp. 381–390. Springer, Heidelberg (2006)

[BDLM08] Bergenthum, R., Desel, J., Lorenz, R., Mauser, S.: Synthesis of petri nets from scenarios with viptool. In: van Hee, K.M., Valk, R. (eds.) ICATPN 2008. LNCS, vol. 5062, pp. 388–398. Springer, Heidelberg (2008)

[BW00] Best, E., Wimmel, H.: Reducing k-safe petri nets to pomset-equivalent 1-safe petri nets. In: Nielsen, M., Simpson, D. (eds.) ICATPN 2000. LNCS, vol. 1825, pp. 63–82. Springer, Heidelberg (2000)

[Dar98] Darondeau, P.: Deriving Unbounded Petri Nets from Formal Languages. In: Sangiorgi, D., de Simone, R. (eds.) CONCUR 1998. LNCS, vol. 1466, pp. 533–548. Springer, Heidelberg (1998)

[Dar00] Darondeau, P.: Region based synthesis of P/T-nets and its potential applications. In: Nielsen, M., Simpson, D. (eds.) ICATPN 2000. LNCS, vol. 1825, pp. 16–23. Springer, Heidelberg (2000)

[Dil50] Dilworth, R.P.: A decomposition theorem for partially ordered sets. Annals of Mathematics 51, 161–166 (1950)

[ED96] Badouel, E., Darondeau, P.: On the synthesis of general petri nets. Technical Report PI-1061, IRISA (November 1996)

[GR83] Goltz, U., Reisig, W.: Processes of place/transition nets. In: Díaz, J. (ed.) ICALP 1983. LNCS, vol. 154, pp. 264–277. Springer, Heidelberg (1983)

[JLD05] Juhás, G., Lorenz, R., Desel, J.: Can I execute my scenario in your net? In: Ciardo, G., Darondeau, P. (eds.) ICATPN 2005. LNCS, vol. 3536, pp. 289–308. Springer, Heidelberg (2005)

[LBD08] Lorenz, R., Bergenthum, R., Desel, J.: Synthesis of petri nets from infinite partial languages. In: ACSD, pp. 170–179 (2008)

[LBDM07] Lorenz, R., Bergenthum, R., Desel, J., Mauser, S.: Synthesis of petri nets from finite partial languages. In: Proceedings of ACSD, pp. 157–166 (2007)

[LJ06] Lorenz, R., Juhás, G.: Towards synthesis of petri nets from scenarios. In: Donatelli, S., Thiagarajan, P.S. (eds.) ICATPN 2006. LNCS, vol. 4024, pp. 302–321. Springer, Heidelberg (2006)

# Modeling and Analysis of Transportation Networks Using Batches Petri Nets with Controllable Batch Speed

Isabel Demongodin

Laboratoire des Sciences de l'Information et des Systèmes (LSIS),
University of Aix-Marseille, 13397 Marseille, France
isabel.demongodin@lsis.org

**Abstract.** In road transportation networks, such as freeways, motorways or highways, variable speed limit (VSL) control is one of the most efficient strategies to substantially improve traffic flow and solve or reduce the congestion problem. One way to design control strategies applicable in real time is to represent the vehicle flows with a hybrid dynamic model. In this context, Batches Petri Nets (BPN) and their extensions, are well adapted for the modeling of such systems as they intend to model variable delays on continuous flows by adding special nodes, called batch nodes, to the continuous and hybrid Petri nets. Using BPN with controllable batch speed, this paper studies a portion of the A12 highway in The Netherlands. This hybrid representation leads to the evaluation of several real time VSL control laws in accordance with the accumulation front of vehicles.

**Keywords:** transportation network, real-time control, variable speed limit, Batches Petri nets.

## 1 Introduction

The control of the congestion, meaning the maximization of the vehicle flow with real-time actions, becomes one of the most delicate problems in the urban and road traffic domain [9]. To analyze the congestion problem, there is already a large variety of models, methods and architectures [6]. Among the class that consists of dynamic discrete events and hybrid models, continuous and hybrid Petri nets [2] are well adapted to the modeling and analysis of performance and control of flow systems. This class of nets combines the advantages of discrete modeling to represent the control using timed discrete Petri nets, and of continuous modeling using continuous Petri nets to represent flows. Several authors use these hybrid formalisms for the modeling and analysis of traffic systems, either with a macroscopic representation or microscopic representation: continuous Petri nets with constant speed have been used in [5] to study the traffic in the city of Turin in Italy, the authors of [13] use continuous Petri nets with variable speed to represent the congestion phenomenon, while in [7] the authors have studied the optimal traffic light control of an urban transportation system. Batches Petri nets [3] extend the hybrid Petri nets class by defining a new type of node, the batch node, and the concept of batch, i.e. a group of entities moving through a transfer zone at a certain speed. These Petri nets allow by their hybrid dynamic formalization to represent in a very detailed manner transfer elements with the possibility of accumulation of entities. Since its first definition and application to high throughput manufacturing systems, batches PN has been extended and widespread by several authors in different domains: [14] defines the stochastic batch Petri nets while [8] extends the model to evaluate multi-modal hubs. Adopting a suitable spatial decomposition and a mesoscopic representation of transportation systems, several elementary models have been defined [4] by means of the generalized batches Petri net (GBPN): one way sections, two ways sections, convergent and divergent intersections. Dedicated to transportation systems, GBPN has proved to be an accurate and powerful formalism to analyze vehicles flows. Thanks to this Petri net model, the accumulation levels can be characterized in real time. It is now possible to consider the control of traffic congestion.

The off-line optimal control could be constructed for any section of the traffic net, while its real time solution and its realization present a certain number of difficulties. Some advanced control theories have been developed on continuous and hybrid Petri net formalisms but most of those works have been dedicated to traffic light control [7] and to their application to urban transportation systems [5]. In transportation systems such as freeways, motorways or highway, ramp metering and variable speed limit (VSL) control are the most efficient strategies to substantially improve traffic flow and to solve or reduce the congestion problem [10]. While the control by ramp metering or coordinated ramp metering can be established thanks to the traffic light control studies, few works have been dedicated to VSL control strategies. The main impact of VSL on traffic flow is deemed to be the reduction of the mean speed under critical densities, and the homogenization of speeds, i.e. reduction of speed differences among vehicles.

In this context, this paper proposes a model using GBPN with controllable batch speed for analyzing control laws of a real transportation system. Section 2 presents the concepts of batches Petri nets by focusing on the hybrid dynamics of controllable batches. With such extensions, it is now conceivable to regulate the speed of vehicles according to the level of the accumulation front. Furthermore, a real infrastructure, a 10 km segment of the A12 highway in the Netherlands, is studied. Two VSL control strategies on this highway are discussed in section 3.

## 2 Concepts of Batches Petri Nets

A Batches Petri Net (BPN) intends to model variable delays on continuous flows by adding special nodes, called batch nodes, to a hybrid Petri net [2] combining transition timed discrete Petri nets and constant continuous Petri nets. BPN [3] is thus composed of three kinds of places and three kinds of transitions: discrete place and discrete transition, continuous place and continuous transition, batch place and batch transition (see Fig.1). Batch nodes combine both a discrete event and a linear continuous dynamics in a single structure: a batch transition acts like a continuous transition while a batch place integrates a hybrid formalism of the circulation flow.

## 2.1 Definitions and Notations

**Definition 1.** A Generalized Batches Petri net (GBPN) [3], is defined by a 5-tuple $B = (R, f, c, Tempo, M_0)$ where:

1) $R$ is a Petri net defined by $R = <P, T, Pre, Post>$ with $P$: finite set of places, $T$: finite set of transitions, $Pre\ (P_i, T_j)$ is a function defining the weight of an arc from a place to a transition and $Post\ (P_i, T_j)$ is a function defining the weight of an arc from a transition to a place.
2) $f$: $P \cup T \rightarrow \{D, C, B\}$, called the "batch function", indicates for every node if it is a discrete, continuous or batch node.
3) $c$: if $f(P_i) = B$: $P_i \rightarrow \{V_i, d_{maxi}, s_i\} \in \Re^+ \times \Re^+ \times \Re^+$,
$c$, called the "characterized batch function", associates three continuous characteristics (speed, maximum density and length) to every batch place.
4) $Tempo$ is an application that associates a rational positive or null number to every transition:
- if $f(T_j) = D$, then $Tempo\ (T_j) = d_j$ is the delay associated with the discrete transition $T_j$, expressed in time unit.
- if $f(T_j) = C$ or $B$, then $Tempo\ (T_j) = \Phi(T_j) = \Phi_j$ is the maximum firing flow associated with transition $T_j$, expressed in entities/time unit. With every continuous or batch transition $T_j$, is also associated an instantaneous firing flow, noted $\varphi_j(t)$, representing the quantity of markings by time unit that fires transition $T_j$. The instantaneous firing flow of $T_j$ is a piecewise constant function with a value lower or equal to the maximum firing flow of $T_j$: $0 \leq \varphi_j(t) \leq \Phi_j$.
5) $M_0 = M(t_0) = (m_1^0, m_2^0, \ldots, m_n^0)$ is an initial marking. We denote by $M(t) = (m_1(t), m_2(t), \ldots, m_n(t))$, the marking at time $t$. According to the type of places, the marking is defined in different sets:
- if $f(P_i) = D$, then $m_i \in \aleph$: the marking of a discrete place is a natural integer.
- if $f(P_i) = C$, then $m_i \in \Re^+$: the marking of a continuous place is a non negative real.
- if $f(P_i) = B$, then $m_i = \{B_{1i}, ..., B_{pi}, ..., B_{ni}\}$: the marking of a batch place is a series of batches. □

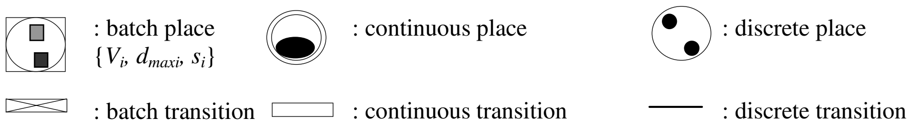


**Fig. 1.** Nodes of Batches Petri nets

In the basic definition found in [3], a batch, i.e. an internal coherent batch, represents a series of entities with the same distribution characteristics during a time interval. At time $t$, an internal coherent batch ($ICB_p$) is characterized by three continuous variables: a length ($l_p \in \Re^+$), a density ($d_p \in \Re^+$), and a head position ($x_p \in \Re^+$). With this concept, the marking of a batch place $P_i$, is a series of batches, ordered on their position, moving forward at the same speed $V_i$. For controlling the speed of batches moving at different speeds, we define here the concept of controllable batch, which is an internal coherent batch with a speed characteristic.

**Definition 2.** At time $t$, a *controllable batch* is defined by a quadruple: $CtB_k(t) = [l_k(t), d_k(t), x_k(t), v_k(t)] \in \mathfrak{R}^+ \times \mathfrak{R}^+ \times \mathfrak{R}^+ \times \mathfrak{R}^+$ where $(l_k)$ is a length, $(d_k)$ a density , $(x_k)$ a head position and $(v_k)$ is a driving speed. The *instantaneous flow* of a controllable batch is defined as: $\varphi(x_k, t) = \varphi_k(t) = v_k(t).d_k(t)$. A controllable batch that composes the marking of place $P_i$, is noted $CtB_{ki}(t) = [l_k(t), d_k(t), x_k(t), v_k(t)]_i$. □

At time $t$, a batch is called an *output batch* of $P_i$, noted $OB_{ki}$, if its head position is equal to the length associated with $P_i$: $x_k(t) = s_i$. If the density of a batch is equal to the maximal density of the batch place, $d_k(t) = d_{maxi}$, this batch is totally accumulated. Due to the bounded characteristics of a batch place, some constraints on batches characteristics have to be respected: $0 \le l_k \le x_k \le s_i$ (position and length constraints) and $0 \le d_k \le d_{maxi}$ (density constraint).

For instance, let us consider a batch place $P_i$ with $c(P_i) = \{V_i, d_{maxi}, s_i\}$, two controllable batches $CtB_{pi}(t)$, $CtB_{qi}(t)$, and one output controllable batch $OCtB(t)$, defined as follow (see Fig. 2): $CtB_{pi}(t) = [l_p(t), d_p(t), x_p(t), v_p(t)]_i$, $CtB_{qi}(t) = [l_q(t), d_q(t), x_q(t), v_q(t)]_i$ and $OCtB_{0i}(t) = [l_0(t), d_0(t), s_i, v_0(t)]_i$. Thus, the marking of place $P_i$ is: $m_i = \{CtB_{pi}(t), CtB_{qi}(t)_i, OCtB_{0i}(t)\}$. We note $\varphi_{in}(t)$ and $\varphi_{out}(t)$, the input flow and the output flow, respectively, of the batch place, i.e.: $\varphi_{in}(t) = \varphi(0, t)$ and $\varphi_{out}(t) = \varphi(s_i, t)$.

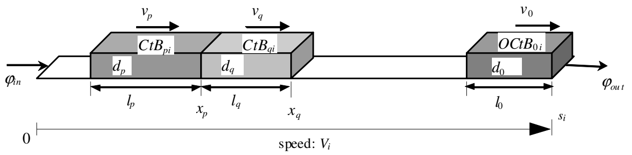


**Fig. 2.** Controllable batches

Various static functions, previously defined on internal coherent batches, have been adapted for controllable batches.

**Definition 3.** At time $t$, let us consider a controllable batch $CtB_{ki}(t) = [l_k(t), d_k(t), x_k(t), v_k(t)]_i$ of place $P_i$.

- if the length is null ($l_k(t) = 0$) and it is not a creating batch ($x_k(t) \neq 0$), this batch can be destroyed. The *destruction* function is noted by $CtB_{ki}(t) = \emptyset$.
- it is always possible to split a batch into two batches. This *splitting* can be either on the length with the same density or on the density with the same length.
- if two batches are in contact with the same density and the same speed, they can be merged. Let us consider $CtB_{qi}(t) = [l_q(t), d_k(t), x_q(t), v_k(t)]_i$, with $x_k(t) = x_q(t) + l_k(t)$. In this case, the downstream batch of $CtB_{qi}(t)$, $CtB_{ki}(t)$, becomes $CtB_{ki}(t) = [l_k(t)+l_q(t), d_k(t), x_k(t), v_k(t)]_i$ and batch $CtB_{qi}(t)$ is destroyed, $CtB_{qi}(t) = \emptyset$. □

The dynamics of GBPN with controllable batch speed is realized by the firing of transitions and the hybrid dynamic of batch places, i.e. the hybrid dynamics of controllable batches that can be resumed as follow.

## 2.2 Dynamics of Controllable Batches

Batch places describe the transfer of batches according to a switching dynamic between two behaviors: the free behavior and the accumulation behavior. Both dynamics of a batch place are governed by the state of the batches composing it.

**Definition 4.** At time $t$, a controllable batch $CtB_{ki}(t) = [l_k(t), d_k(t), x_k(t), v_k(t)]_i$ is in *a free type behavior* (noted $R_{ki}(t)$ = F) if its elements move freely at the driving speed $v_k(t)$, while it is in *an accumulation type behavior* (noted $R_{ki}(t)$ = A) if its elements are not transferred at the driving speed but move according to an output flow which has a lower value than its instantaneous flow. Let us consider a given $\varphi_{outBki}(t)$:

- if $\varphi_{ki}(t) \leq \varphi_{outBki}(t) \Rightarrow R_{ki}(t)$ = F or,
- if $\varphi_{ki}(t) > \varphi_{outBki}(t) \Rightarrow R_{ki}(t)$ = A. □

For instance, with $\varphi_{outBqi}(t) = v_q(t).d_{maxi}$, batch $CtB_{qi}(t)$ of Fig.2 is in free behaviour ($R_q(t)$ = F) as $\varphi_{qi}(t) = v_q(t).d_q(t)$ under density constraint, $d_q(t) \leq d_{maxi}$.

**Definition 5.** A batch place has an *accumulation* at the exit, if there exists an accumulated output batch or the output batch is in an accumulated behavior. □

Various equations govern the characteristics of batches according to their position and behavior: creation, moving and exit.

### 2.2.1 Creation of a Batch

At time $t_1$, if the input flow of a batch place $P_i$ is not null, a batch is created and added to the marking of $P_i$ such that: $CtB_{ki}(t_1) = [0, d_k(t_1), 0, v_k(t_1)]_i$ with $d_k(t_1) = \varphi_{in}(t_1)/V_i(t_1)$ and $v_k(t_1) = V_i(t_1)$. From this time on, the moving of this batch is governed by:

$$\begin{cases} \dfrac{dd_k(t)}{dt} = \dfrac{dv_k(t)}{dt} = 0 \\ \dfrac{dl_k(t)}{dt} = \dfrac{dx_k(t)}{dt} = v_k(t_1) \end{cases} \tag{1}$$

### 2.2.2 Moving of a Batch

Equations that govern the moving of a batch inside a batch place $P_i$ depend on the state of this batch (see definition 4).

*1) Batch in free behavior*

From time $t_1$ on, a batch, $CtB_{ki}(t_1) = [l_k(t_1), d_k(t_1), x_k(t_1), v_k(t_1)]_i$, which has a free behavior, $R_{ki}(t_1)$ = F, evolves according to the following equations.

$$\begin{cases} \dfrac{dl_k(t)}{dt} = \dfrac{dd_k(t)}{dt} = \dfrac{dv_k(t)}{dt} = 0 \\ \dfrac{dx_k(t)}{dt} = v_k(t_1) \end{cases} \tag{2}$$

*2) Batch in accumulated behavior*

At time $t_1$, a batch of $P_i$, $CtB_{pi}(t_1) = [l_p(t_1), d_p(t_1), x_p(t_1), v_p(t_1)]_i$, has an accumulated behavior $R_{pi}(t_1)$ = A, if:

(i) $CtB_{pi}(t_1)$ is in contact with another (downstream) batch $CtB_{qi}(t_1) = [l_q(t_1), d_q(t_1), x_q(t_1), v_q(t_1)]_i$ i.e. $x_q(t_1) = x_p(t_1) + l_q(t_1)$,

(ii) the sum of both batch densities is greater than the maximal density of $P_i$ i.e. $d_p(t_1) + d_q(t_1) > d_{maxi}$, and $d_q(t_1) < d_{maxi}$,

(iii) $CtB_{qi}(t_1)$ has a lower speed i.e. $v_q(t_1) < v_p(t_1)$.

*Remark*: for all other cases, i.e. if $[d_p(t) + d_q(t) \le d_{maxi}]$, or $[d_p(t) + d_q(t) > d_{maxi}$ and $v_q(t_1) \ge v_p(t_1)]$, both batches are in free type behavior, and their evolution is governed by (2). In case that $d_q(t_1) = d_{maxi}$, batch $CtB_{pi}$ cannot pass batch $CtB_{qi}$ and, thus the speed of batch $CtB_{pi}$ is reduced to the speed of $CtB_{qi}$, i.e. $v_p(t_1) = v_q(t_1)$.

From time $t_1$ on, batch $CtB_{pi}$ can pass batch $CtB_{qi}$, which has a slower speed. For representing this phenomenon, at time $t_1$, batch $CtB_{pi}$ is split into two batches $CtB_{p'i}$ and $CtB_{p''i}$, and an empty batch $CtB_{p'''i}$ is introduced with $CtB_{p'''i}(t_1) = [0, d_q(t_1), x_p(t_1), v_q(t_1)]_i$ such that:

- $CtB_{p'i}(t_1)=[l_p(t_1), d_{p'}(t_1), x_p(t_1), v_p(t_1)]_i$ with $d_{p'}(t_1) = d_{maxi} - d_q(t_1)$ and from $t_1$, its behavior is governed by (2)
- $CtB_{p''i}(t_1) = [l_p(t_1), d_{p''}(t_1), x_p(t_1), v_p(t_1)]_i$ with $d_{p''}(t_1) = d_p(t_1) - d_{p'}(t_1)$ i.e., $d_{p''}(t_1) = d_p(t_1) + d_q(t_1) - d_{maxi}$

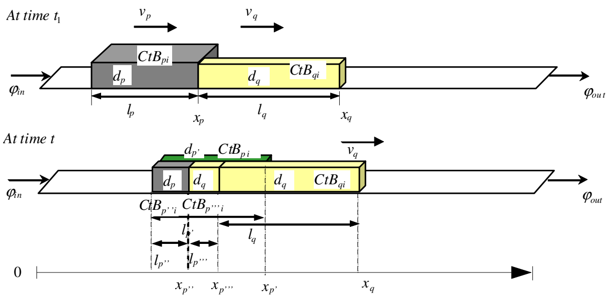


**Fig. 3.** Dynamics of two batches in accumulation behavior

Between two timed events, in other words during time interval $[t_1, t_2]$, batch densities do not vary, i.e. $d_{p'''}(t) = d_q(t_1)$ and $d_{p''}(t) = d_p(t_1) + d_q(t_1) - d_{maxi}$, with $t_1 < t < t_2$. Thus, according to the conservation law and the distance covered by batches, the equations that govern both batches $CtB_{p''i}$ and $CtB_{p'''i}$, from time $t_1$ on, are:

$$\begin{cases} \dfrac{dd_{p''}(t)}{dt} = \dfrac{dv_{p''}(t)}{dt} = 0 \\ \dfrac{dx_{p''}(t)}{dt} = \dfrac{d_p(t_1)}{d_{\max i} - d_q(t_1)} v_q(t_1) - \dfrac{(d_p(t_1) + d_q(t_1) - d_{\max i})}{d_{\max i} - d_q(t_1)} v_p(t_1) \\ \dfrac{dl_{p''}(t)}{dt} = -\dfrac{d_p(t_1)}{d_{\max i} - d_q(t_1)} (v_p(t_1) - v_q(t_1)) \end{cases} \tag{3}$$

$$\begin{cases} \dfrac{dd_{p'''}(t)}{dt} = \dfrac{dv_{p'''}(t)}{dt} = 0 \\ \dfrac{dx_{p'''}(t)}{dt} = v_q(t_1) \\ \dfrac{dl_{p'''}(t)}{dt} = -\dfrac{(d_p(t_1) + d_q(t_1) - d_{\max i})}{d_{\max i} - d_q(t_1)}(v_p(t_1) - v_q(t_1)) \end{cases} \tag{4}$$

Finally, at time $t_2$ and by definition 3, a merging is applied to batches $CtB_{qi}$ and $CtB_{p'''i}$ as they get in contact with the same speed and the same density.

### 2.2.3 Exit of a Batch

At time $t_1$, let us consider an output batch of $P_i$, $OCtB_{0i}(t_1) = [l_0(t_1), d_0(t_1), s_i, v_0(t_1)]_i$.

*1) Output batch in free behavior*

In this case, $\varphi_{out}(t_1) \geq \varphi_{0i}(t_1)$, with $\varphi_{0i}(t_1) = v_0(t_1).d_0(t_1)$, and from time $t_1$ on, the evolution is governed by:

$$\begin{cases} \dfrac{dx_0(t)}{dt} = \dfrac{dd_0(t)}{dt} = \dfrac{dv_0(t)}{dt} = 0 \\ \dfrac{dl_0(t)}{dt} = -v_0(t_1) \end{cases} \tag{5}$$

*2) Output batch in accumulated behavior*

We assume that the output flow of the batch place is constant during a certain delay, i.e. at time $t' > t_1$, $\varphi_{out}(t') = \varphi_{out}(t_1)$. Two cases can be distinguished: the output batch is totally accumulated or not, at time $t_1$.

*a)* $d_0(t_1) = d_{maxi}$

As the output batch is already accumulated, it must reduce its speed, i.e. $v_0(t_1) = \varphi_{out}(t_1)/d_0(t_1) = \varphi_{out}(t_1)/d_{maxi}$. Thus, at time $t_1$, $OCtB_{0i}(t_1) = [l_0(t_1), d_{maxi}, s_i, \varphi_{out}(t_1)/d_{maxi}]_i$. From this time on, the evolution is governed by:

$$\begin{cases} \dfrac{dx_0(t)}{dt} = \dfrac{dd_0(t)}{dt} = \dfrac{dv_0(t)}{dt} = 0 \\ \dfrac{dl_0(t)}{dt} = -\dfrac{\varphi_{out}(t_1)}{d_{\max i}} \end{cases} \tag{6}$$

*b)* $d_0(t_1) < d_{maxi}$

In this case, the output batch starts its accumulation as its density is strictly inferior to the maximal one of the batch place. For representing such a phenomenon, at time $t_1$, this batch is split into two batches in contact such that: $OCtB_{A0i}(t_1) = [0, d_{maxi}, s_i, v_{A0}(t_1)]_i$ and $CtB_{0i}(t_1) = [l_0(t_1), d_0(t_1), s_i, v_0(t_1)]_i$, with $v_{A0}(t_1) = \varphi_{out}(t_1)/d_{maxi}$.

Consequently to the conservation law, the distance covered by the end of the output batch from time $t_1$ on, and as $d_0(t) = d_0(t_1)$ and $d_{A0}(t) = d_{maxi}$, for time $t > t_1$, the evolution of batches $OCtB_{A0i}$ and $CtB_{0i}$ is governed by:

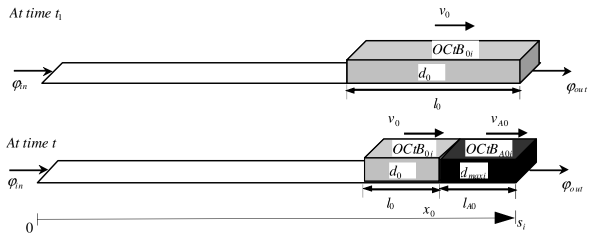


**Fig. 4.** Dynamic of an output batch in accumulation behavior

$$\begin{cases} \dfrac{dx_{A0}(t)}{dt} = \dfrac{dd_{A0}(t)}{dt} = \dfrac{dv_{A0}(t)}{dt} = 0 \\ \dfrac{dl_{A0}(t)}{dt} = \dfrac{v_0(t_1)d_0(t_1) - \varphi_{out}(t_1)}{d_{\max i} - d_0(t_1)} \end{cases} \tag{7}$$

$$\begin{cases} \dfrac{dd_0(t)}{dt} = \dfrac{dv_0(t)}{dt} = 0 \\ \dfrac{dl_0(t)}{dt} = \dfrac{\varphi_{out}(t_1) - v_0(t_1)d_{\max i}}{d_{\max i} - d_0(t_1)} \\ \dfrac{dx_0(t)}{dt} = -\dfrac{v_0(t_1)d_0(t_1) - \varphi_{out}(t_1)}{d_{\max i} - d_0(t_1)} \end{cases} \tag{8}$$

## 2.3 Dynamics of Batches Petri Nets

Enabling conditions and firing rules are those of timed transition discrete PN (with a preselection policy), constant speed continuous PN, and batch PN. All definitions and concepts, such that incidence matrix, place and transition invariants, are preserved when considering controllable batches instead of internal coherent batches (see [3] for more details). The behavior algorithm of a batch PN is based on a discrete event approach with linear or constant continuous evolutions between timed events. It calculates the states of the system only when it undergoes discontinuity. The first phase of the algorithm determines the enabled transitions, next reserves the marks in discrete and continuous places, computes instantaneous firing flows, and finally, establishes batch place states. From these states and their values, all timed events that change the global state of the system are determined, as described below. The date of the nearest event, which is the nearest in time, becomes the current date and at this instant, and new markings are deduced from the firing rules. Thus, between two events or two dates, the state of the hybrid model has an invariant behavior state (IB-state), which corresponds to a period of time such that:

- the marking in discrete places is constant and,
- the instantaneous firing flow of continuous and batch transitions is constant and,
- the reserved marking of discrete and continuous places is constant.

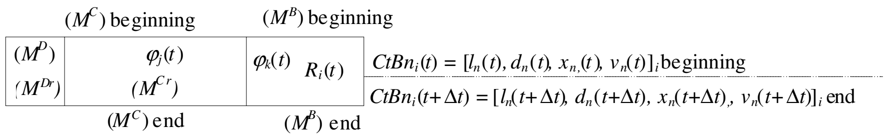
The IB-state changes if and only if one (or possibly several at the same time) of the following kinds of events occurs:

1. firing of a discrete transition: $T_j$.
2. the non reserved marking of a continuous place becomes equal to zero: $m^n_i = 0$.
3. the non reserved marking of a continuous place which is a pre- place of a discrete transition becomes equal to the weight of the corresponding arc: $m^n_i = a$.
4. a batch of a batch place becomes an output batch: $CtB_{0i} = OCtB_{0i}$.
5. an output batch of a batch place is destroyed: $OCtB_{0i} = \varnothing$.

Inside a batch place, several timed events have to be taken into account in the dynamic evolution of batches: (i) a batch becomes an output batch (i.e. event 4 above); (ii) meeting of two batches; (iii) end of the total accumulation of a batch; (iv) destruction of a batch; (v) end of over taking of a batch.

As in hybrid PN, the behavior of a BPN can be represented by an evolution graph where a node represents an IB-state of the dynamic model. The nodes are linked by arcs with a labeled transition that determines the occurred events and the past delay between two consecutive IB-states. The label associated to this transition is noted as following: {occur events / $\Delta t$} where $\Delta t$ is the past delay between events. As shown in Fig.5, and illustrated in Fig.9, a node is decomposed into three parts:

1. The first part represents discrete markings $M^D$ and reserved marking of discrete places $M^{Dr} = \{m^r_i / f(P_i) = D\}$.
2. The second part represents the instantaneous firing flows of continuous transitions $\{\varphi_j / f(T_j) = C\}$, and reserved markings $M^{Cr} = \{m^r_i / f(P_i) = C\}$ of continuous places. The total marking of continuous places $M^C$ at the beginning of the phase and at the end of the phase is also represented.
3. The third part represents the instantaneous firing flows of batch transitions $\{\varphi_j / f(T_j) = B\}$, the behavior of batch places $\{R_i / f(P_i) = B\}$and markings $M^B$ of batch places, at the beginning of the phase and at the end of the phase. The characteristics of batches are described on the right side of the node.



$(M^C)$ beginning
$(M^B)$ beginning
$(M^D)$
$(M^{Dr})$
$\varphi_j(t)$
$(M^{Cr})$
$\varphi_k(t)$
$R_i(t)$
$CtBn_i(t) = [l_n(t), d_n(t), x_n(t), v_n(t)]_i$ beginning
$CtBn_i(t+\Delta t) = [l_n(t+\Delta t), d_n(t+\Delta t), x_n(t+\Delta t), v_n(t+\Delta t)]_i$ end
$(M^C)$ end
$(M^B)$ end

**Fig. 5.** Node of the evolution graph

## 3 Speed Control of a Highway Network Traffic

The traffic network under study is the A12 highway between the sites of Veenendaal and Maarsbergen, in the province of Utrecht, The Netherlands. This study is inspired from the master thesis of Andrea Pinna [11], where the model of this A12 segment is developed in a microscopic approach using the Paramics traffic simulator. The considered system is a simple part of the highway, with a length of 10 kilometers, where the traffic flow of this stretch is modeled only for the east-west direction, that is, from

Arnhem to Den Haag. A petrol station is present just in the middle of the stretch, introducing one off-ramp and one on-ramp, which are separated by 1 km. Thus, the flow of the vehicles in the highway is perturbed by the vehicles that, stopping for a lay over, enter and exit the service station. In particular, the vehicles entering the highway from the on-ramp, adding up to the already strong flow of vehicles in the main stream, could cause congestions and traffic jams. Hence, the chronic problems of traffic congestion appear.

In real highways and during the circumstances of congestion, the control panels advise the drivers about the suggested speed ((30, 40, 50, 60, 70, 80, 90, 100, 110, 120 km/h) or (8, 11, 14, 16, 19, 22, 25, 27, 30, 33 m/s)), in order to maximize the flow of vehicles and to minimize the negative effects of traffic jams. When this dynamic speed limit is applied on a highway section panel, all vehicles will be enforced to follow the suggested speed. Moreover, on this highway, there are detector loops that continuously measure the traffic variables (flow, density, speed, etc.). To each of these detector loops is associated a position in the highway; unfortunately, this position is not always accurate. In order to have a good model of the network, it is necessary to know the exact position of the detector loops. Thanks to the service provided by Google Earth, it has been possible to measure the true reciprocal distances between the detector loops [11]. Detector loops are usually placed few meters after control panels, which are clearly visible in the pictures offered by Google Earth. In the case study segment, we will use 11 detector loops. Finally, the two following data will be considered as limit parameters that the vehicle flows have to be respected: accumulation speed (2.7 m/s) which defines the maximum speed a vehicle could move in order to be counted as being in an accumulation; accumulation length (200 m) sets the minimum length of a group of vehicles so that it could be considered an accumulation.

## 3.1 Modeling of the A12 Highway

The A12 highway portion is characterized by the presence of four zones: two of them are strictly origin zones, while the other two are strictly destination zones. The two main zones are the start of the freeway stretch near Veenendaal at km 91 (zone 1) and the end before the intersection near Maarsbergen at km 81 (zone 4). The entrance at km 85 (zone 2) and the exit at km 84 (zone 3) of the petrol station are the remaining zones.

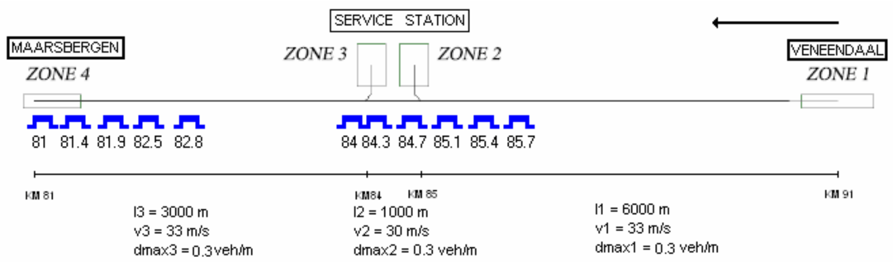


**Fig. 6.** The model of the case study network

A 10 kilometers stretch connects zones 1 - 4, while zones 2 - 3 are linked through an off-ramp and an on-ramp, for which the traffic network model will not consider the curvature of the road. According to the data in [11] (i.e. information received from the Dutch Ministry of Transportations) the traffic demand on this 10 km highway portion between 6:00 and 7:00 a.m., can be considered as follows: 2448 vehicles left zone 1 to get directly to zone 4; 100 vehicles left zone 1 going into the service station (zone 2) and 107 vehicles left the service station in order to reach zone 4.

From this description, three sections of the 10 km highway portion have been defined (see Fig. 6): section 1 from zone 1 to zone 2; section 2 from zone 2 to zone 3; section 3 from zone 3 to zone 4. By using elementary models dedicated to transportation infrastructures (see [4], for more details), the GBPN model (see Fig. 7) is established, where the three sections are represented by the three batch places $P_8$ (section 1), $P_9$ (section 2) and $P_{10}$ (section 3).

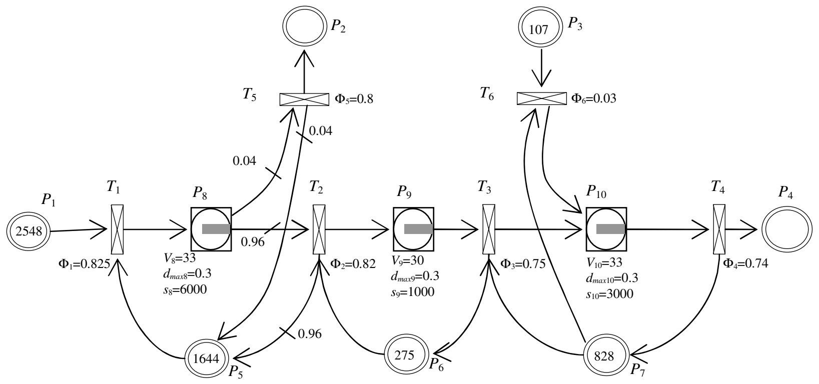


**Fig. 7.** GBPN model of the portion of the A12 highway

The three defined sections of the 10 km highway sector have a normal traffic around 6:00 a.m. Therefore, at the initial time, each batch place contains a batch with the length equal to that of the batch place and moving with the maximum speed of the batch place. Thus, we define an initial state where each place contains an output batch not accumulated: $OCtB_8(t_0)$ = $[6000, 0.026, 6000, 33]_{8,}$ $OCtB_9(t_0)$ = $[1000, 0.025, 1000, 30]_{9,}$ $OCtB_{10}(t_0)$ = $[3000, 0.024, 3000, 33]_{10}$. In other terms, at the initial time, section 1 contains 156 vehicles, section 2 contains 25 vehicles while 72 vehicles are in section 3. Zones 1, 2, 3 and 4 are represented by continuous places $P_1$, $P_2$, $P_3$ and $P_4$, respectively. Places $P_5$, $P_6$ and $P_7$ limit the capacity of each batch place $P_8$, $P_9$ and $P_{10}$, and according to the initial state of the batch places, the marking of these continuous places is respectively: $m_5$ = 1644, $m_6$ = 275 and $m_7$ = 828.

*Remarks:* (i) All the values have been transformed in meters and seconds instead of kilometers and hours, in order to be compatible with the BPN tool. (ii) As each section is represented by a batch place, no distinguishing is established for different lanes of a section. Since the speed limits are sometimes distinct from lane to lane, it has been necessary to reach a compromise: the speed limit applied to several lanes is the average of the speed limits.

## 3.2 Performance Indicators of Transportation Systems

We now focus our attention on the influence of a control law that regulates the maximal speed of sections (or batch places) according to the accumulation front. In the spirit of transportation systems [6], for establishing the influence of a speed control strategy, it is necessary to define performance indicators.

Different time parameter evolutions can be characterized from the evolution graph, such as: number of entities passing through batch places (i.e. quantity of marks), length of accumulation at the exit of a section, batches density, fundamental diagram, etc. When a change of the speed of a batch place occurs as in Controlled BPN [1], this change has to be considered as an external event. In the GBPN with controllable batch speed, a change of the speed of a place will only limit the batch being created.

The notion of a quantity vector, a vector composed of the quantities of marks, has already been defined in GBPN [3]. For discrete and continuous places, the quantity of marks is equal to their marking values. The quantity of marks associated with a batch place represents the number of units composing all the batches, i.e.

$$q_i(t) = \sum_k l_k(t).d_k(t), \forall\, k / CtB_{ki}(t) \in m_i(t).$$

With this notion, we define the mean density of a batch place $P_i$, as the weighted mean of the density of each batch in the same batch place:

$$\bar{d}_i(t) = \frac{\sum_k l_k(t).d_k(t)}{\sum_k l_k(t)}, \forall\, k / CtB_{ki}(t) \in m_i(t)\,.$$

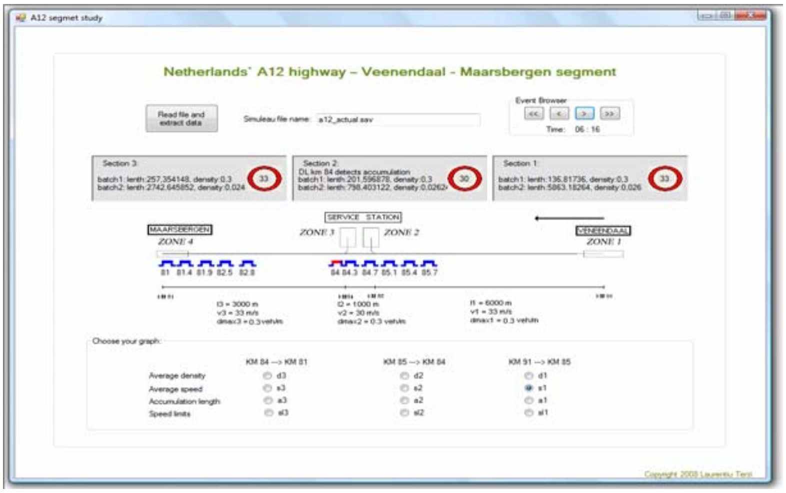
For evaluating the speed control laws, we also define the mean speed of a batch place as the weighted mean of the speed of each batch in the same batch place:

$$\bar{v}_i(t) = \frac{\sum_k v_k(t) \cdot l_k(t).d_k(t)}{\sum_k l_k(t).d_k(t)}, \forall\, k / CtB_{ki}(t) \in m_i(t)\,.$$

Finally, in order to analyze the performance of the studied system, a simulation software tool, named A12segment, has been developed [12]. Based on Simuleau, i.e. a tool dedicated to BPN (linked with Sirphyco that is a tool for HPN), the A12segment tool is in charge of the user interface for the A12 highway portion. More precisely, the tool reads data from a Simuleau output file and extracts the previous indicators. It thus provides a manual managed event driven player that allows the user to see a text description of the state of the traffic model, at each event that occurs.

## 3.3 Evaluation without Control Law

Considering all the initial state information, with a simulation time of 3600 seconds and a precision of 0.01, the evolution graph can be established thanks to Simuleau. However, due to space limits, the complete evolution graph of the highway system is


A12 segmet study

Netherlands' A12 highway – Veenendaal - Maarsbergen segment

Simuleau file name: a12_actual.sav

Event Browser Time: 06 : 16

Section 3:
batch1: lenth:257.354148, density:0.3
batch2: lenth:2742.645852, density:0.024

Section 2:
DL km 84 detects accumulation
batch1: lenth:201.596878, density:0.3
batch2: lenth:798.403122, density:0.0262

Section 1:
batch1: lenth:136.81736, density:0.3
batch2: lenth:5863.18264, density:0.026

l3 = 3000 m
v3 = 33 m/s
dmax3 = 0.3 veh/m

l2 = 1000 m
v2 = 30 m/s
dmax2 = 0.3 veh/m

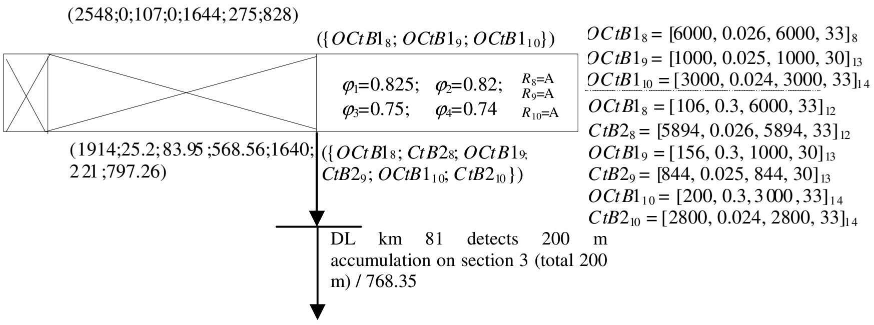
l1 = 6000 m
v1 = 33 m/s
dmax1 = 0.3 veh/m

Choose your graph:

| | KM 84 --> KM 81 | KM 85 --> KM 84 | KM 91 --> KM 85 |
|---|---|---|---|
| Average density | d3 | d2 | d1 |
| Average speed | s3 | s2 | [✓ s1] |
| Accumulation length | a3 | a2 | a1 |
| Speed limits | sl3 | sl2 | sl1 |

**Fig. 8.** A12 segment interface

not presented in this paper. Only the initial state and the event corresponding to an accumulation length equal to 200 m. on section 3 are represented in Fig. 9.

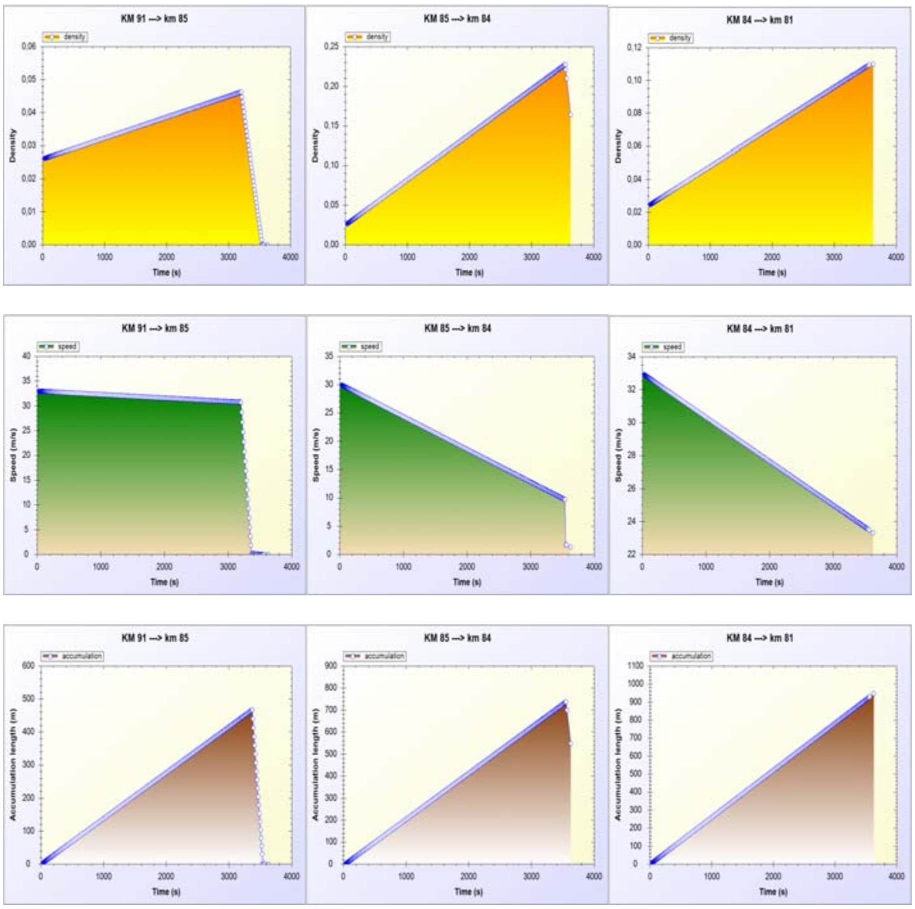


(2548;0;107;0;1644;275;828)
$(\{OCtB1_8; OCtB1_9; OCtB1_{10}\})$
$OCtB1_8 = [6000, 0.026, 6000, 33]_8$
$OCtB1_9 = [1000, 0.025, 1000, 30]_{13}$
$OCtB1_{10} = [3000, 0.024, 3000, 33]_{14}$
$\varphi_1$=0.825; $\varphi_2$=0.82; $R_8$=A $R_9$=A
$\varphi_3$=0.75; $\varphi_4$=0.74 $R_{10}$=A
$OCtB1_8 = [106, 0.3, 6000, 33]_{12}$
$CtB2_8 = [5894, 0.026, 5894, 33]_{12}$
$OCtB1_9 = [156, 0.3, 1000, 30]_{13}$
$CtB2_9 = [844, 0.025, 844, 30]_{13}$
$OCtB1_{10} = [200, 0.3, 3000, 33]_{14}$
$CtB2_{10} = [2800, 0.024, 2800, 33]_{14}$
(1914;25.2;83.95;568.56;1640;221;797.26)
$(\{OCtB1_8; CtB2_8; OCtB1_9; CtB2_9; OCtB1_{10}; CtB2_{10}\})$
DL km 81 detects 200 m accumulation on section 3 (total 200 m) / 768.35

**Fig. 9.** Initial node of the evolution graph

After extracting the information from the Simuleau output file, the A12segment tool provides the following evolutions for the main parameters of the A12 highway, as shown in Fig. 10: average density, average speed and accumulation length.

From these evolutions, it can be observed that an accumulation is created on section 1, continue to increase on section 2 helped by the existence of the service station and maintains considerable dimensions when it reaches section 3. The increasing of the average density and the decreasing of the average speed on the sections are the main consequences of the appearance and increasing of the vehicle accumulation. As a conclusion, it can be inferred that an accumulation of maximum 467 m - 737 m - 946 m on section 1 - 2 - 3 produces an increasing of the average density with 0.019 veh/m - 0.205 veh/m - 0.085 veh/m and a decreasing in speed with 2 m/s - 20 m/s - 10 m/s, respectively.



Section 1 Section 2 Section 3

**Fig. 10.** Parameter evolutions without control law

## 3.4 Evaluation with Speed Control Laws

The two following decision algorithms for choosing the right moment or event to change the speed limits into the sections of the highway have been tested.

### 3.4.1 Decision Algorithm 1

i) If the detector loop at km 85.1 (beginning of section 1) detects that vehicles move with 2.7 m/s (defined accumulation speed) and, if the camera from km 85.1 detects a 200 m length of accumulation, the speed on section 1 will be decreased by 5.55 m/s (20 km/h).

ii) If 300 seconds (5 min.) elapsed from the camera accumulation detection or from the last state check, the information received from the detector loops and the cameras will be analyzed and, depending on the variation of the accumulation lengths, different speed limits will be shown on the information panels.

a- If the accumulation length of a section is over 200 m, the speed limit on this section will be reduced by 5.5 m/s, but not under 8 m/s.

b- If the accumulation length of a section is still over 200 m and if the speed limit on this section is under 14 m/s, the speed limit on the previous section will be reduced by 2.7 m/s, but not under 8 m/s.

c- If the accumulation length of a section is below 200 m, the speed limit on this section will be increased by 2.7 m/s, unless it already reached the maximum speed imposed by the default speed limit of the section.

d- If the accumulation length of a section is still below 200 m, and the speed limit on this section is increased over 14 m/s, the speed limit on the previous section will be increased by 2.7 m/s, unless it already reached the maximum speed imposed by the default speed limit of this other section.

By applying this speed control strategy to the highway, new results have been obtained (see Fig. 11). It can be observed that when the accumulation on section 1 reaches 200 m, the length of accumulation is controlled. The accumulation length on section 2 is better, but still with considerable accumulation dimensions – the maximum length of accumulation reached 650 m. On section 3, the accumulation length reaches a maximum value closer to 400 m, a quite good evolution compared with 930 m in the case without control. Nevertheless, the accumulation length reaches 650 m because the decision algorithm checks the state of the road only at a 300 seconds interval and sometimes an accumulation, not controlled at the incipient phase, can become difficult to control and eliminate. Therefore, an improved decision algorithm will be proposed in order to solve this problem.

### 3.4.2 Decision Algorithm 2

The improved decision algorithm is described below:

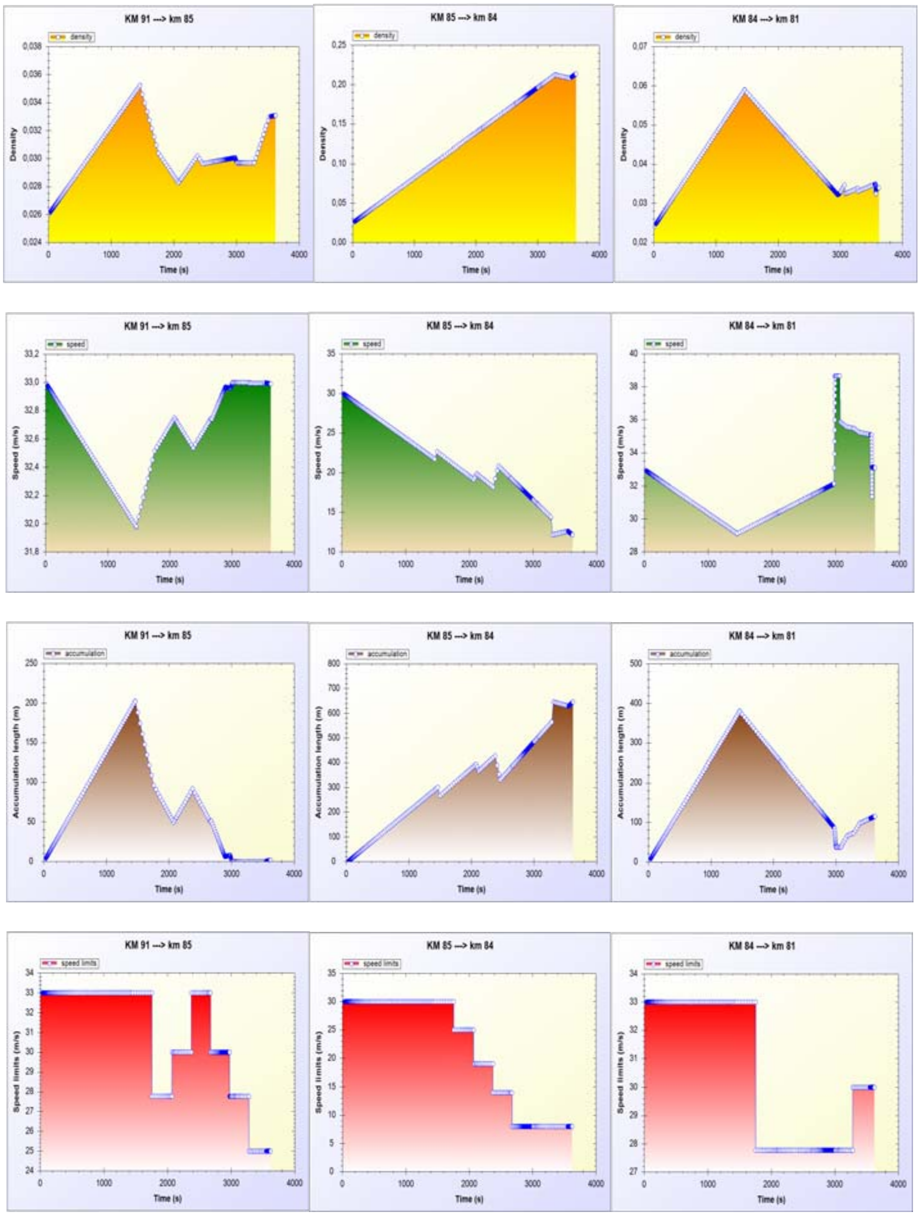
i) If the detector loops from km 85.1 or km 84 or km 81, detect that vehicles move with 2.7 m/s (defined accumulation speed) and the cameras placed at the same positions detect a 200 m length of vehicle accumulation, then:

a- If the accumulation length of a section is over 200 m, the speed limit on this section will be reduced by 5.5 m/s, but not under 8 m/s.

b- If the accumulation length of a section is still over 200 m and the speed limit on this section got under 14 m/s, the speed limit on the previous section will be reduced by 2.7 m/s, but not under 8 m/s.

ii) If 300 seconds elapsed from the last camera and detector loop accumulation detection (it means that during 300 seconds no accumulation over 200 m was detected on the road), the sensors of the highway will do another state check, the information received from the detector loops and the cameras from all three sections will be analyzed and depending on the variation of the accumulations lengths different speed limits will be shown on the information panels.

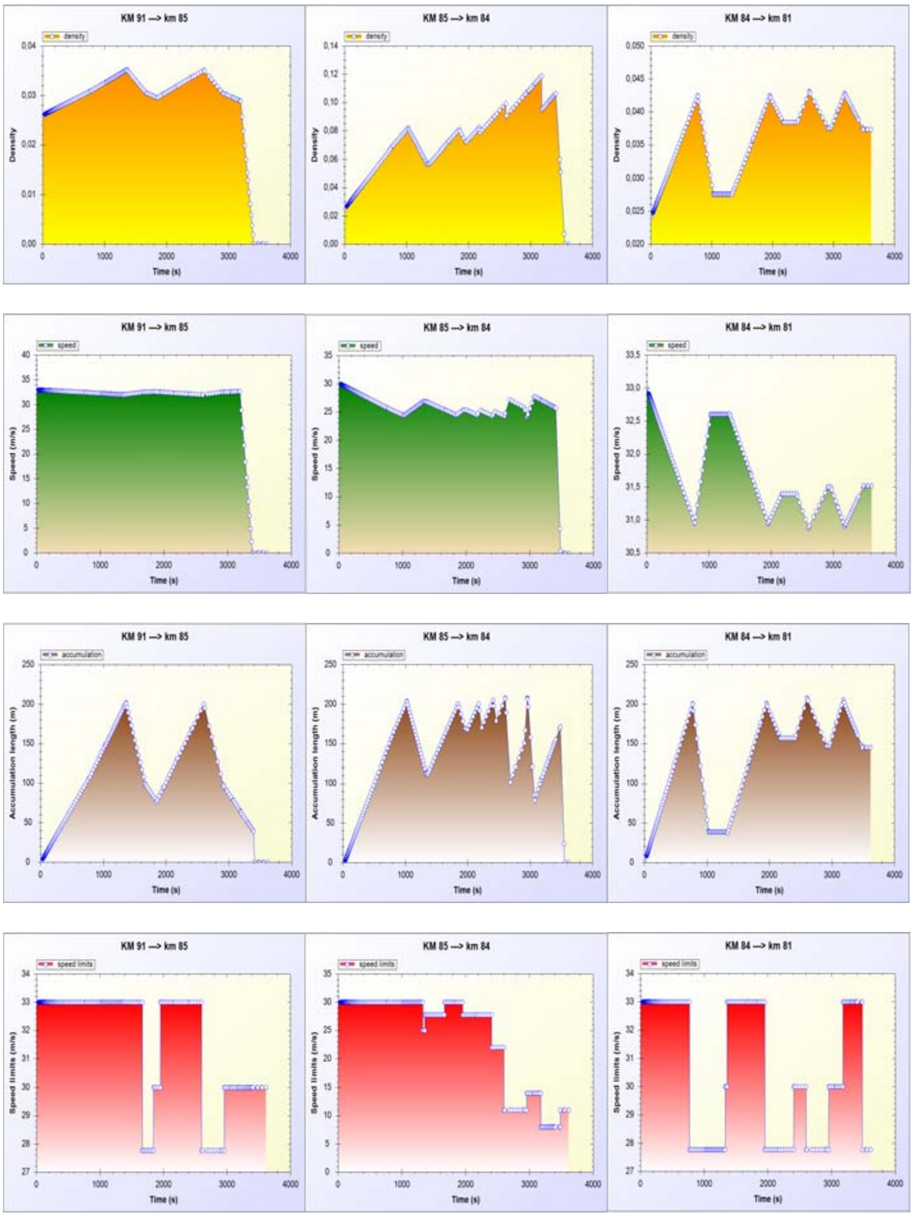
a- If the accumulation length of a section is below 200 m, the speed limit on the section will be increased by 2.7 m/s, unless it already reached the maximum speed imposed by the default speed limit of the section.



Section 1 Section 2 Section 3

**Fig. 11.** Parameter evolutions – decision algorithm 1

b- If the accumulation length of a section is below 200 m, and the speed limit on the section is over 14 m/s, the speed limit on the previous section will be increased by 2.7 m/s, unless it already reached the maximum speed imposed by the default speed limit of this other section.



Section 1 Section 2 Section 3

**Fig. 12.** Parameter evolutions – decision algorithm 2

From Fig. 12, it can be observed that when the vehicle accumulation on section 3 reaches 200 m at 768 seconds, the accumulation length on the three sections is very well controlled. The variation of the average density is very good on the first section, acceptable on the second, and good on the third one, while the average speed maintains high values in sections 1 and 2 and a slight decrease in section 3. The speed limits during the simulation confer a pleasant and relaxing trip on sections 1 and 3, but some small delays can occur on section 2 due to the lower speed limits applied. Finally, the improved decision algorithm allows the accumulation length on all three sections to be controlled to a maximum value of 207 m, a mean value of 150 m and a minimum value of 37 m. The vehicle accumulation is not totally eliminated, but it is successfully maintained under the value of 200 m, that is considered the critical length for a normal traffic state to be considered with accumulations.

A disadvantage of reducing the speed limits is, as expected, a longer time for the non-accumulated vehicles to transit the section, however for security reasons, the control of congestion is more important in traffic situations.

## 4 Conclusions

The main advantage of the batches Petri net class is the mixing of discrete-event model, continuous time model, and model integrating the set theory, through the concept of batches, which represents variable delays on continuous flows. Thanks to this formalism, several studies have dealt with performance evaluation and control evaluation of a real high throughput manufacturing system, i.e. a bottling line of Perrier spring company. In this paper, we use Generalized Batches Petri nets with controllable batch speed to model a portion of a highway and to analyze two speed control strategies. A further step could be to apply optimal control strategies by dealing with supervisory control and hybrid control in order to regulate congestion through the combined use of ramp metering and VSL control. Finally, connections between different tools dedicated to timed discrete PN, continuous PN, hybrid PN (as HYPENS or SIRPHYCO) and Batches Petri nets (SIMULEAU) should also be developed, to allow designers and users to analyze different systems from manufacturing , transportation or computer science domains.

## References

[1] Audry, N., Prunet, F.: Controlled Batches Petri nets. In: Int. Conference on Systems, Man and Cybernetics, IEEE/SMC, Vancouver, Canada, pp. 1849–1854 (1994)

[2] David, R., Alla, H.: Discrete, Continuous and Hybrid Petri nets. Springer, Heidelberg (2005)

[3] Demongodin, I.: Generalised Batches Petri Net: Hybrid model for high speed systems with variable delays. J. of Discrete Event Systems 11, 137–162 (2001)

[4] Demongodin, I., Hzami, Z.: Modélisation et analyse de la circulation de véhicules par réseaux de Petri lots. In: 7$^{\text{ième}}$ Conférence Francophone de Modélisation et Simulation , Paris, France (2008)

[5] Di Febbraro, A., Sacco, N.: On modelling urban transportation networks via hybrid Petri nets. Control Engineering Practice 12, 1225–1239 (2004)

[6] Hoogendoorn, S.P., Bovy, P.H.L.: State-of-the-art of vehicular traffic flow modeling. J. of Systems & Control Engineering 215, 283–303 (2001)

[7] Júlvez, J., Boel, R.: Modelling and controlling traffic behavior with continuous Petri Nets. In: 16th IFAC World Congress, Prague (2005)

[8] Kaakai, F., Hayat, S., El Moudni, A.: Quantitative assessment of travellers'connection times into a multimodal hub owing to batches Petri nets. In: 17th IMACS World Congress, Paris, France (2005)

[9] Papageorgiou, M., Diakaki, C., Vaya, D., Apostolos, K., Wang, Y.: Review of road traffic control strategies. Proc. of the IEEE 91(12), 2043–2067 (2003)

[10] Papamichail, I., Kampitaki, K., Papageorgiou, M., Messmer, A.: Integrated Ramp Metering and Variable Speed Limit Control of Motorway Traffic Flow. In: 17th IFAC World Congress, Seoul, Korea, pp. 14084–14089 (2008)

[11] Pinna, A.: Modeling, Calibration and Validation of Highway Traffic Networks. Master thesis, Delft University of Technology / University of Cagliari (2007)

[12] Terzi, L.: Modeling and analysis of urban and highway traffic systems by batches Petri nets. Master thesis, Politechnica univ. Bucarest / University of Aix-Marselle (2008)

[13] Tolba, C., Lefebvre, D., Thomas, P., El Moudni, A.: Continuous and timed Petri nets for the macroscopic and microscopic traffic flow control. Simulation Modelling Practice and Theory 13, 407–436 (2005)

[14] Wang, Y., Zhou, C.: Fluid based simulation approach for high volume conveyor transportation systems. J. of Systems Science and Systems Engineering 13(3), 297–317 (2004)

# Oclets – Scenario-Based Modeling with Petri Nets

Dirk Fahland

Humboldt-Universität zu Berlin, Institut für Informatik,
Unter den Linden 6, 10099 Berlin, Germany
fahland@informatik.hu-berlin.de

**Abstract.** We present a novel, operational, formal model for scenario-based modeling with Petri nets. A scenario-based model describes the system behavior in terms of partial runs, called *scenarios*. This paradigm has been formalized in message sequence charts (MSCs) and live sequence charts (LSCs) which are in industrial and academic use. A particular application for scenarios are process models in disaster management where system behavior has to be *adapted* frequently, occasionally at run-time. An *operational semantics* of scenarios would allow to execute and adapt such systems on a formal basis.

In this paper, we present a class of Petri nets for *specifying* and *modeling* systems with scenarios and anti-scenarios. We provide an operational semantics allowing to iteratively construct partially ordered runs that satisfy a given specification. We prove the correctness of our results.

**Keywords:** scenarios, operational semantics, partial order, Petri nets.

## 1 Introduction

A recurring application of formal methods is the design, validation, and verification of distributed systems which consist of several interacting processes or components. For this purpose, *scenario-based* methods like *message sequence charts* (MSCs) and *live sequence charts* (LSCs) [1] have become accepted *specification* techniques: The behavior of a system is specified as a set of *scenarios* being self-contained, partial executions. A scenario can be declared as possible, imperative, or forbidden. A formal semantics allows to validate a system's runs against the scenarios following their intuitively understandable meaning [2].

Following the scenario-based paradigm [1,2], we have shown in [3] that in some domains like disaster response system behavior can only faithfully be captured if the *complete* behavior is given by a set of scenarios and anti-scenarios. Assuming completeness and consistency turns a set of scenarios into a system *model*. This particular representation has advantages when *adapting* a given model by adding, removing, or modifying its scenarios without breaking the entire model. In [3], we sketched an approach for this kind of modeling, executing, and adapting systems with scenarios based on Petri nets.

In this paper, we present a complete and consistent formal model of our approach in [3] and explain how a scenario-based specification evolves into a system model within the same formalism.

A major difficulty when using scenarios is the step from a system specification to a system *model* with formal *operational semantics* that provides the set of enabled actions which extend a given run s.t. no scenario is violated. Existing operational models for MSCs and LSCs require a translation into another formalism like automata [4], process algebras [5], or state charts [6], or use formal techniques like graph grammars [7]. This makes operational semantics for scenarios surprisingly technical while scenarios and their composition appear to be very intuitive. In the worst case, the modeler cannot relate the operational model to its original scenarios by mere comparison.

A formal model for scenarios with operational semantics within the same formalism would give a more coherent view on the technique and on system models. A candidate formalism are *Petri nets*: They offer an intuitively understandable notation together with a rigorously defined, simple, and well-understood partial-order semantics [8]. The well-developed Petri net structure theory and available verification techniques could be used for analyzing and verifying behavioral properties of the system [9,10]. Established extensions for Petri nets, like colors or time, could easily be transferred to scenarios. Petri net synthesis techniques could help in translating a scenario-based model into a state-based model.

In this paper, we propose a novel formalization of scenarios based on Petri nets that takes existing results, specifically from LSCs into account. We define a new class of Petri nets called *oclets*. An oclet formalizes a scenario as an acyclic, labeled net, that can be read as a partial, partially ordered run. A prefix of the oclet is denoted as a *precondition* for the scenario which must be observed before the entire scenario can occur. We also define *anti-oclets* to denote partial runs which must not occur completely. A *specification* is a set of oclets and anti-oclets.

We provide a *declarative, formal semantics* of oclets to characterize the satisfying, partially ordered runs of an oclet specification. The semantics allows to check whether a given (Petri net) system satisfies the given scenarios. We complement this semantics with an *operational semantics* that turns a specification into an operational model and allows for directly constructing partially ordered runs from oclets. We show that the operational semantics are equivalent to the declarative semantics under a closed world assumption.

The remainder of this paper is organized as follows. In the next section, we informally introduce the concepts of our approach as we revisit the dining philosophers problem and sketch a solution of the problem with scenarios. We formally define the new Petri net class of oclets in Sect. 3, followed by their declarative semantics in Sect. 4. Section 5 is dedicated to the operational semantics of oclets. We wrap up our approach as we solve the problem of the dining philosophers with oclets in Sect. 6. We compare our approach with related work in Sect. 7 and conclude in Sect. 8.

## 2 Specifying with Scenarios – An Informal Introduction

Before we begin with formal definitions, we explain the concepts and the underlying intuition of our approach by the help of the well-known *dining philosophers*; see [9] for instance. We first illustrate the philosophers problem on a Petri net model of the system, and then informally sketch a solution with scenarios. We revisit and solve the problem with our model in Section 6. We assume the reader to be familiar with basic notions of Petri nets.

### 2.1 The Dining Philosophers Problem

The Petri net system $(N^3_{\mathsf{phil}}, m^3_{\mathsf{phil}})$ in Fig. 1 models three philosophers each taking his forks at once; this stricter variant will be sufficient to illustrate our ideas.

Each circle $\langle \mathsf{th}_i, \mathsf{take}_i, \mathsf{eat}_i, \mathsf{rel}_i \rangle, i = 1, 2, 3$ models the behavior of philosopher $i$ going from thinking to eating and back by taking and releasing his left and right fork $\mathsf{f}_i$ and $\mathsf{f}_{i \oplus 1}$; by $\oplus$ (and $\ominus$ later on), we denote addition (and subtraction) modulo $n$. The philosophers synchronize on their shared forks: no two neighboring philosophers may eat at the same time. The system exhibits linear runs like the following: (a) $\langle \mathsf{take}_1, \mathsf{rel}_1, \mathsf{take}_1, \mathsf{rel}_1, \ldots \rangle$, (b) $\langle \mathsf{take}_1, \mathsf{rel}_1, \mathsf{take}_2, \mathsf{rel}_2, \mathsf{take}_1, \ldots \rangle$, and (c) $\langle \mathsf{take}_1, \mathsf{rel}_1, \mathsf{take}_2, \mathsf{rel}_2, \mathsf{take}_3, \mathsf{rel}_3, \mathsf{take}_1, \ldots \rangle$.

In (a), phil. 1 always takes both of his forks, none of his neighbors eats. In (b), phil 1 and phil 2 alternatingly eat, alternatingly taking the left and the right fork of phil. 3 who never eats. In (c), each philosopher eats. Runs (a) and (b) are *unfair* as transition $\mathsf{take}_3$ gets enabled infinitely often but never fires. These unfair runs are undesired in the system, runs like (c) are desired.

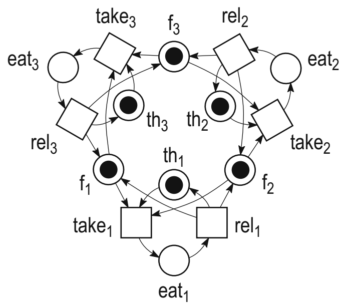


**Fig. 1.** Petri net model $N^3_{\mathsf{phil}}$ of three dining philosophers with its initial marking $m^3_{\mathsf{phil}}$

The *dining philosophers problem* is to specify a system that distributedly coordinates the execution of the philosophers s.t. the system contains no deadlocks and no unfair runs. Here, we seek for a stricter solution that has only *decent* runs where a philosopher, after having released his forks, refuses to take them again until each neighbor has taken and released the corresponding shared fork [9]. Runs of kind (c) are decent. Figure 2 depicts a decent, partially ordered run of $(N^3_{\mathsf{phil}}, m^3_{\mathsf{phil}})$, corresponding to run (c) above.

### 2.2 The Basic Idea

We now want to sketch a solution for the dining philosophers problem with scenarios. Our solution shall have the following properties: (i) A scenario is a well-understandable fragment of a partially ordered run. (ii) System behavior is composeable from scenarios in an intuitive way. (iii) Anti-scenarios allow specifying forbidden behavior. (iv) Behavioral preconditions of scenarios restrict the applicability of a scenario to certain situations. (v) The semantics of scenarios allows testing whether a set of runs satisfies all scenarios. (vi) Finally, satisfying runs can be constructed from the given scenarios in an operational manner. We follow this agenda on an informal level in this section and we provide a corresponding formal model from Sect. 3 onwards.

We begin with the notion of a scenario. In the run $\pi_3$ of Fig. 2 we not only find copies of the transitions and places of $N^3_{\mathsf{phil}}$, but even larger, overlapping patterns.

The possibly most obvious pattern, out of which the entire run is composed, is depicted in Fig. 3. It denotes one unrolled execution cycle of philosopher $i \in \{1, 2, 3\}$. The net itself is acyclic but its labels denote that at the end of the execution, the local state $[\mathsf{f}_i, \mathsf{th}_i, \mathsf{f}_{i\oplus1}]$ is visited again. It specifies a logically self-contained, partial execution of the philosophers system. Such a structure is a *scenario*.

By the symmetry of the philosophers system, *every* partially-ordered run of $(N^3_{\mathsf{phil}}, m^3_{\mathsf{phil}})$ consists of overlapping copies of $\mathsf{phil}(i), i \in \{1, \ldots, 3\}$ which denote the elementary scenarios of $(N^3_{\mathsf{phil}}, m^3_{\mathsf{phil}})$. As we wish to observe these scenarios in the system, we call them *qualified*.

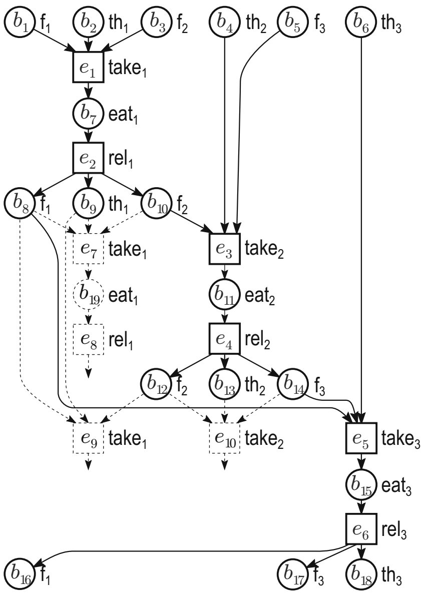


**Fig. 2.** A decent run $\pi_3$ (bold nodes) of the three dining philosophers of Fig. 1

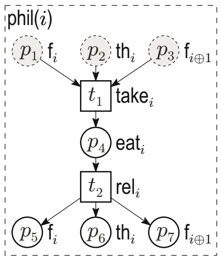


**Fig. 3.** The elementary behavioral fragment of the philosophers – oclet $\mathsf{phil}(i)$

From this observed decomposition, we can infer an appropriate and intuitive *composition* of qualified scenarios: Append a scenario $A$ to another scenario $B$ by merging places at the beginning of $A$ with equally labeled places at the end of $B$. Likewise, a scenario can be appended to a run: If an initial run $\pi_0$, consisting of the places $b_1, \ldots, b_6$ of Fig. 2, is given, then $\pi_3$ can be constructed by first appending $\mathsf{phil}(1)$ followed by $\mathsf{phil}(2)$ and $\mathsf{phil}(3)$.

This way, we can compose all partially ordered runs of $(N^3_{\mathsf{phil}}, m^3_{\mathsf{phil}})$, even the non-decent ones. For instance, appending $\mathsf{phil}(1)$ to $\pi_0$ followed by $\mathsf{phil}(1)$ again adds transition $e_7$ ($\mathsf{take}_1$) and subsequent nodes. A run that begins with firing $\mathsf{take}_2$ can be composed by first appending $\mathsf{phil}(2)$ to $\pi_0$. Intuitively, all these runs *satisfy* the scenarios $\{\mathsf{phil}(1), \ldots, \mathsf{phil}(3)\}$. In the course of this paper, we generalize this appending composition to overlapping scenarios. In either case, a set of scenarios is meaningful only, if the scenarios share some labels.

## 2.3 Anti-Scenarios Exclude Behavior

We just sketched how composing qualified scenarios yields partially ordered runs. Although each scenario $\mathsf{phil}(i), i = 1, 2, 3$, is qualified, we can construct undesired, non-decent runs as explained. *Anti-scenarios* are an expressive mean to exclude undesired behavior [6].

The non-decent behavior of the philosophers, as defined in Sect. 2.1, can be narrowed down to the anti-scenarios $\mathsf{decentL}(i)$ and $\mathsf{decentR}(i)$ of Fig. 4. Scenario $\mathsf{decentL}(i)$ denotes that after the left fork $\mathsf{f}_i$ was released by philosopher $i$, it is directly taken again by philosopher $i$; $\mathsf{decentR}(i)$ respectively for the right fork $\mathsf{f}_{i\oplus 1}$. A partially ordered run that completely contains an anti-scenario $\mathsf{decentL}(i)$ or $\mathsf{decentR}(i)$ is not decent; such a run *violates* the anti-scenario. The run consisting of the nodes $\{b_1, \ldots, b_{10}, e_1, e_2, e_7, b_{19}\}$ of Fig. 2 violates $\mathsf{decentL}(1)$.

## 2.4 Behavioral Preconditions

The previous sections introduced the basic concepts of scenarios and their relation to runs. So far, a scenario can be appended to a run as soon as its beginning can be merged with the run. We now introduce a behavioral *precondition* for scenarios.

The grey shaded (dashed) behavior in Fig. 3, 4, and 5 denotes each scenario's *precondition*. It can be a partial marking as in Fig. 3 or a finite, connected history as in Fig. 4 and 5. All other behavior is the *contribution* of the scenario. The interpretation is that a qualified scenario can extend a given run only, if its precondition is satisfied (has been observed) in the run. Conversely, the run must not continue with the contribution of an anti-scenario, if its precondition has been observed.

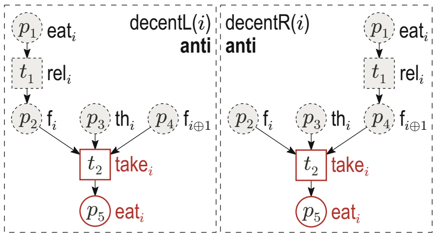


**Fig. 4.** Anti-oclets $\mathsf{decentL}(i)$ and $\mathsf{decentR}(i)$ specifying the decent use of forks $i$ and $i \oplus 1$

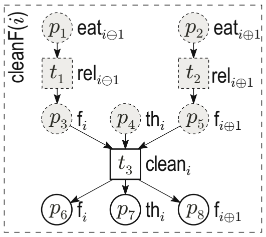


**Fig. 5.** A qualified scenario with a history-based precondition

Scenario $\mathsf{cleanF}(i)$ in Fig. 5 denotes that phil. $i$ may clean both forks after they have been used and released by his left neighbor $i \ominus 1$ and his right neighbor $i \oplus 1$. The precondition specifies that $\mathsf{clean}(1)$ can be appended to run $\pi_3$ of Fig. 2. Directly appending $\mathsf{cleanF}(1)$ in the initial state is not forbidden by qualified scenario $\mathsf{cleanF}(1)$, but such a run cannot be constructed. Thus a qualified scenario reads as “if the precondition holds, the contribution is executable.”

## 2.5 Scenarios Specify Systems, Scenarios Model Systems

Up to now, we only related scenarios to single runs and explained rather vaguely how a set of scenarios relates to the complete behavior of a system. We explain this relation subsequently.

A *specification* is a set of scenarios. In general, a system *satisfies* a specification, if for every prefix of a run $\pi$ of the system, which allows to append a qualified scenario according to its precondition, the system also has a run $\pi'$ where the scenario was appended to $\pi$. Additionally, no run of the system may violate an anti-scenario of the specification. This definition entails progress for all transitions.

This strict interpretation allows for contradicting specifications: Consider run $\pi_1$ that is created by appending $\mathsf{phil}(1)$ to $\pi_0$ (consisting of $b_1, \ldots, b_{10}, e_1, e_2$). Qualified scenario $\mathsf{phil}(1)$ requires the presence of a run $\pi'_1$ constructed by appending $\mathsf{phil}(1)$ again; thus run $\pi'_1$ contains transition $e_7$ which violates anti-scenario $\mathsf{decentL}(1)$.

For a more flexible style of scenario-based specifications, we weaken the semantics of qualified scenarios allowing that a qualified scenario is not executed completely if this would violate an anti-scenario. Simply said, we prioritize anti-scenarios over qualified scenarios to solve the contradiction (thus a qualified scenario corresponds to a universal LSC with cold cuts only). With this weaker semantics, a system that executes run $\pi_1$, but not $\pi'_1$, as denoted above, does satisfy the specification $\{\mathsf{phil}(1), \mathsf{decentL}(1)\}$. Apparently, every system satisfying $\bigcup_{i=1}^{3}\{\mathsf{phil}(i), \mathsf{decentL}(i), \mathsf{decentR}(i)\}$ has only decent runs.

In [1], Damm and Harel point out an important issue when interpreting a set of scenarios as there are two principle ways to do so: The *existential* interpretation requires only the possibility to execute the qualified scenarios and forbids anti-scenarios in a system. Any other behavior is allowed. The *universal* interpretation requires that the entire behavior of a system is composed only of the qualified scenarios while disallowing anti-scenarios. Any other behavior that cannot be constructed from the scenarios is forbidden.

A modeler usually begins shaping the system behavior with the existential interpretation in mind. Each new scenario adds a further requirement on the system behavior. Once the scenarios are sufficiently detailed, the modeler changes to the universal interpretation enforcing that the system behaves as specified in the scenarios, only. The universal interpretation turns a set of scenarios into a complete system *model*. It allows to define an operational semantics for scenarios which is not permissible for the existential interpretation.

This concludes the informal introduction of the key concepts for scenario-based specifications and models. In the subsequent sections, we revisit these concepts as we defined the notion of a scenario by generalizing the notion of a local step. This allows us to define an existential, and a universal semantics for scenarios based on Petri nets. The latter constructs runs by appending qualified scenarios while preventing the violation of anti-scenarios as sketched above. We will return to the philosophers example in Sect. 6.

## 3 Oclets – A Petri Net Model for Scenarios

The next three sections are dedicated to our formal model of scenario-based specifications and their semantics. We begin with structural definitions of the syntax. We assume the reader to be familiar with the basic formal notions of Petri net theory, we recall the most important ones that we need subsequently; for an introduction we refer to [9].

**Recalling some basic notions.** As usual, we denote a Petri net as $N = (P, T, F)$; we call each place $p \in P$ and each transition $t \in T$ a *node* of $N$. We will use *labeled* nets, $N = (P, T, F, \ell)$, with a labeling function $\ell$ assigning each *node* $x$ of $N$ a label $\ell(x)$ from some set $\mathcal{L}$; $\mathcal{L} = \mathcal{T} \uplus \mathcal{P}$ is partitioned into *action labels* $\mathcal{T}$ and resource labels $\mathcal{P}$ with $\ell(P) \subseteq \mathcal{P}$ and $\ell(T) \subseteq \mathcal{T}$. We canonically lift any notion on any object to sets and to tuples of these objects.

We write ${}^\bullet x$ for the *preset*, and $x^\bullet$ for the *postset* of a node $x$ of $N$. A net $N$ is *acyclic* if the flow-relation $F$ has no directed cycles, i.e. the transitive closure of $F$ contains no pair $(x, x)$; we write $\leq_N$ for the reflexive-transitive closure of $F$. The *minimal* nodes of a set $Y \subseteq P \cup T$ is the set $\min_N Y := \{y \in Y \mid {}^\bullet y \cap Y = \emptyset\}$; *maximal* nodes of $Y$ are $\max_N Y := \{y \in Y \mid y^\bullet \cap Y = \emptyset\}$. The set of transitively reachable predecessors of $Y$ is the set $\lfloor Y \rfloor_N := \{x \mid \exists y \in Y, x \leq_N y\}$; $Y$ is *causally closed* iff $\lfloor Y \rfloor_N \subseteq Y$. The transitively reachable successors are $\lceil Y \rceil_N := \{x \mid \exists y \in Y, y \leq_N x\}$.

A Petri net $\pi = (B, E, F)$ is a *causal net* iff (1) $\pi$ is acyclic, (2) for each node $x$ of $\pi$, $\lfloor \{x\} \rfloor$ is finite, and (3) each place $b \in B$ has at most one pretransition, $|{}^\bullet b| \leq 1$ and at most one posttransition $|b^\bullet| \leq 1$. A labeled causal net that formalizes a *partially ordered run* of a Petri net system as a Petri net again is called *process* (of the system) [8]. We use these three terms synonymously. The net $\pi_3$ of Fig. 2 (bold nodes) is a process of $(N^3_{\mathsf{phil}}, m^3_{\mathsf{phil}})$.

The elements of $B$ and $E$ are called *conditions* and *events*, respectively. For the systems considered in this paper, no event of a process has two equally labeled preconditions and no two equally labeled postconditions; further each event of a process has a non-empty preset. The following notions will help us to argue about the structure of processes:

**Definition 1 (Induced subnet).** *Let $N = (P, T, F, \ell)$ and $M = (P', T', F', \ell')$ be nets. $N$ is a* subnet *of $M$, $N \subseteq M$, iff $P \subseteq P'$, $T \subseteq T'$, $F \subseteq F'$, and $\ell(x) = \ell'(x)$ for all $x \in P \cup T$. Let $Y \subseteq (P \cup T)$. By $N[Y]$ we denote the $Y$-*induced *subnet* $(P \cap Y, T \cap Y, F|_{(Y \times Y)}, \ell|_Y)$ *of $N$.*

**Definition 2 (Complete prefix, ends-with).** *A causal net* $\pi = (B, E, F)$ *is a* prefix *of a causal net* $\rho = (B', E', F')$, $\pi \mathrel{[\!\!\rightarrow} \rho$, *iff* $\pi \subseteq \rho$, $\lfloor B \cup E \rfloor_\rho \subseteq B \cup E$, *and* $\lceil B \cup E \rceil_\rho = B' \cup E'$.

*Prefix* $\pi$ *of* $\rho$ *is* complete *(wrt. postconditions) iff* $(e, b) \in F'$ *implies* $(e, b) \in F$ *for each* $e \in E$. *A set* $R$ *of causal nets is* prefix-closed *iff each complete prefix of each net of* $R$ *is also in* $R$. *The net* $\rho$ ends with $\pi$, $\rho \mathrel{\rightarrow\!\!|} \pi$ *iff* $\pi \subseteq \rho$ *and* $\max_\pi(B \cup E) \subseteq \max_\rho(B' \cup E')$.

**The structure of scenarios.** The aim of our formal model is to describe and construct a system's processes from smaller processes, i.e. the system's scenarios. The simplest kind of scenario is a process given by a single event with its pre- and postconditions; it denotes an *occurrence* of a single transition. We formalize such a scenario as an *atomic oclet*: The event's set of preconditions forms the oclet's *precondition*, the remainder of the process is the oclet's *contribution*. Figure 6 depicts the atomic oclet that denotes the occurrence of transition take$_1$ of Fig. 1.

The theory that we present subsequently generalizes atomic oclets by extending precondition and contribution. In an oclet the occurrence of a transition $t$ can depend on more than its preplaces being marked. Instead, an oclet's precondition can denote a history of transition occurrences which finally produce the tokens on $^\bullet t$. We denote only those predecessors that are *necessary* to fire $t$, i.e. not all postconditions of $t$'s predecessors must be included. Figure 5 depicts such an oclet. In the same way, we allow more events and conditions for building larger contributions of an oclet, but here we require each event's pre- and postconditions to be complete; Fig. 3 depicts this case. Altogether this results in the following formal definition of scenarios constituting the class of *oclets*.

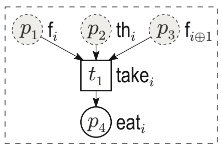


**Fig. 6.** An atomic oclet

**Definition 3 (Oclet).** *An* oclet $o = (P, T, F, \ell, pre)$ *is a labeled, finite causal net* $(P, T, F, \ell)$, *where each* $t \in T$ *has no equally labeled preplaces and no equally labeled postplaces, and a* precondition $pre \subseteq P \cup T$ *that induces a complete prefix* $o[pre]$ *of* $o$ *s.t. each* $x \in \max_o pre$ *has a successor in* $o$.

We call the set $(P \cup T) \setminus pre$ the *contribution* of $o$, which is non-empty by Def. 3. The nets of the Figures 3, 4, and 5 are oclets. We graphically denote the net structure of an oclet as usual, surrounded by a dashed box; a grey shading (and dashed lines) distinguishes the precondition from the contribution.

By definition, the precondition of every oclet is a complete prefix of the entire oclet, its maximal nodes are places, and each oclet ends with its contribution. Thus, the precondition can be evaluated in a state and the contribution begins with a transition. Further, the precondition is a *history* of the contribution. Otherwise, we would require the contribution to observe behavior on which it does not causally depend.

An *oclet specification* is a set of oclets partitioned into qualified oclets and anti-oclets.

**Definition 4 (Oclet specification).** *An oclet specification $O = (Q, A)$ consists of two finite, disjoint sets $Q$ and $A$ of oclets where for each $o \in A$ holds $|T_o \setminus pre_o| = 1$. We call $Q$* qualified *oclets and $A$* anti-oclets.

For instance, $\mathsf{Phil}_3^{\mathsf{dec}} := (\{\mathsf{phil}(1), \mathsf{phil}(2), \mathsf{phil}(3)\}, \bigcup_{i=1}^{3}\{\mathsf{decentL}(i), \mathsf{decentR}(i)\})$ is an oclet specification, see Fig. 3 and 4.

For the scope of this paper, we will impose a rather natural consistency condition on an oclet specification $O$: Let $t_1$ and $t_2$ be two distinct transitions from the contributions of two oclets of $O$. If $t_1$ and $t_2$ have the same labels, then for every preplace (postplace) $p_1$ of $t_1$ exists an equally labeled preplace (postplace) $p_2$ of $t_2$, and vice versa. If this property holds for any two transitions of any two oclets in $O$, then $O$ is *label-consistent*.

Label-consistency ensures that every two oclet transitions with equal labels specify (maybe different) occurrences of the same "system transition". The specification $\mathsf{Phil}_3^{\mathsf{dec}}$ is label-consistent. We do not impose consistency for transition of the precondition as these do not specify a contribution but an observation of behavior prior to a contribution. Here, partial correspondence is sufficient.

## 4 Formal Semantics of Oclets

We just defined the syntax of scenarios as oclets. In this section, we will define their semantics in terms of sets of satisfying runs, i.e., labeled causal nets. The semantics of a qualified oclet $o$ shall read as "if $pre_o$ holds, then the entire oclet $o$ can occur". The semantics of an anti-oclet is much simpler: the entire anti-oclet does not occur in any run.

**Some useful terminology.** The decisive concept to relate an oclet to a run, and hence, to define the semantics of oclets is what we call an *embedding*. An oclet can occur at several places in a run, that is, a run can have several subnets that are isomorphic to an oclet. For clearly distinguishing between an oclet and its occurrences, we say that an oclet is *embedded* in a run if the run contains a subnet that is isomorphic to the oclet; the corresponding subnet isomorphism is an embedding of the oclet into the run. For technical reasons, we formally define these terms for induced subnets.

**Definition 5 (Embedding).** *Let $N$ and $M$ bet two labeled Petri nets, let $X_N \subseteq P_N \cup T_N$ and $X_M \subseteq P_M \cup T_M$. A mapping $\alpha : X_N \to X_M$ is an* embedding *of $N[X_N]$ in $M[X_M]$ iff for each node $x \in X_N$ holds $\ell_N(x) = \ell_M(\alpha(x))$ and for each edge $(x_1, x_2) \in F_N$ exists an edge $(\alpha(x_1), \alpha(x_2)) \in F_M$.*

As an example, consider oclet $\mathsf{phil}(2)$ of Fig. 3 and the process $\pi_3$ of Fig. 2. The mapping $\alpha_1$ with $\alpha_1 = [p_1 \mapsto b_{10}, p_2 \mapsto b_4, p_3 \mapsto b_5]$ is an embedding of $pre_{\mathsf{phil}(2)}$ into $\pi_3$. To simplify notation, when referring to an induced subgraph, e.g. $\mathsf{phil}(2)[pre_{\mathsf{phil}(2)}]$, we only write its inducing nodes, e.g. $pre_{\mathsf{phil}(2)}$, if the net is obvious from the context and confusion is safely avoided.

Our next step will be to relate a run to a set of runs which we call its *continuations.* This effectively means that the run gets various new labeled events, conditions, and arcs. We want to distinguish all these continuations only up to *isomorphism*. That means, in the remainder of the paper, we will treat isomorphic Petri nets as equal, specifically regarding containment in sets, etc. This treatment comes natural if we bear the graphical interpretation of nets in mind. It has been shown earlier, e.g. in [10], how this treatment of isomorphic nets can be reduced to strict mathematical identity by choosing canonic identities for nodes of nets.

**Continuing a run with an oclet.** We now have all notions to formalize the semantics of an oclet. We first define how a prefix of one oclet relates to one run and how this run can be continued with the oclet. We then lift this notion to a set of runs that satisfies one oclet and finally define the semantics of an oclet specification, i.e. sets of oclets.

We introduce some notation for describing where (a prefix of) an oclet is embedded in a run. Let $o$ be an oclet and let $X \subseteq (P_o \cup T_o)$ be the nodes of a complete prefix $o[X]$ of $o$; let $\pi$ be a labeled causal net.

1. The prefix $o[X]$ *holds at the end* of $\pi$ by embedding $\alpha$, denoted $(\pi, \alpha) \models o[X]$, iff $\alpha$ embeds $o[X]$ at the end of $\pi$, formally $\pi \longrightarrow\!| \ \alpha(o[X])$, see Def. 2.
2. The prefix $o[X]$ *holds in* (the past of) $\pi$ by $\alpha$, denoted $(\pi, \alpha) \models \Diamond\, o[X]$ iff $\alpha$ embeds $o[X]$ in $\pi$.
3. We write $\pi \models \varphi$ for $\exists \alpha : (\pi, \alpha) \models \varphi$ and $\pi \models \neg\, \varphi$ for $\neg\, \exists \alpha : (\pi, \alpha) \models \varphi$.

For instance, $\pi \models \neg \Diamond\, o[X]$ expresses that $o[X]$ does not hold anywhere in $\pi$. Although our notation takes inspiration from temporal logic, we do not build such a logic here. Nevertheless, it is easy to prove that $(\pi, \alpha) \models \Diamond\, o[X]$ holds iff there exists a prefix $\pi'$ of $\pi$ with $(\pi', \alpha) \models o[X]$. In our example, the precondition of $\mathsf{phil}(1)$ of Fig. 3 holds at the end of run $\pi_3$ in Fig. 2 while the precondition of $\mathsf{decentL}(1)$ of Fig. 4 does not hold in $\pi_3$.

A process in which the precondition of a qualified oclet holds naturally suggests to *continue* this run by appending the complete oclet at its end. We are not only interested in this largest continuation but also in all intermediate continuations. Of course, a run is a complete prefix of each of its continuations.

**Definition 6 (Continuation).** *Let $\pi, \pi'$ be labeled causal nets, let $o$ be an oclet. $\pi'$ is a* continuation *of $\pi$ with $o$, $\pi \xrightarrow{o} \pi'$ iff $\pi$ is a prefix of $\pi'$ and there exists a complete prefix $o[X']$ of $o$ with $pre_o \subseteq X' \subseteq P_o \cup T_o$ and embeddings $\alpha$ and $\alpha'$ with $(\pi, \alpha) \models o[pre_o]$, $(\pi', \alpha') \models o[X']$ s.t. all new nodes come from $X'$ only: $\alpha'|_{pre_o} = \alpha$ and $\alpha'(X' \setminus pre_o) = (B' \cup E') \setminus (B \cup E)$.*

The set of prefixes of $o$ induces the set of continuations of $\pi$. Thus, a continuation of $\pi$ appends some nodes of $o$ s.t. a larger prefix of $o$ holds. In our example, consider the run $\pi_0$ (consisting of $b_1, \ldots, b_6$) and the run $\pi_1$ (consisting of $b_1, \ldots, b_{10}, e_1, e_2$) in Fig. 2. Run $\pi_1$ is the largest continuation of $\pi_0$ with oclet $\mathsf{phil}(1)$.

A set of runs $R$ is *closed under continuations* with $o$ iff for each $\pi \in R$, each continuation $\pi'$, $\pi \xrightarrow{o} \pi'$, is a run in $R$.

**Definition 7 (Semantics of an oclet).** *A set of labeled causal nets $R$* satisfies *an oclet $o$, $R \models o$ iff $R$ is prefix-closed and closed under continuations with $o$. $R$* satisfies *the* negation *of $o$, $R \models \neg\, o$ iff $R$ is prefix-closed and $o$ does not hold in any run in $R$: $\forall \pi \in R : \pi \models \neg \Diamond\, o$.*

A strict interpretation of an oclet specification $O = (Q, A)$ would be $R \models O$ iff $R \models o$ for all $o \in Q$, and $R \models \neg\, o$ for all $o \in A$. This would allow for contradicting specifications with no satisfying run as explained in Sect. 2.5.

**Existential semantics of an oclet specification.** We motivated in Sect. 2.5, that we are interested in a weaker semantics of an oclet specification that requires the satisfaction of a qualified oclet in a run only up to the point where an anti-oclet would be violated.

To achieve this, we cannot require that a set of runs is closed under all continuations; we have to exclude those continuations that would violate an anti-oclet. The formalization is straight forward: Let $\pi$ be a run, let $o$ be a qualified oclet, and let $o'$ be an anti-oclet. A continuation $\pi \xrightarrow{o} \pi'$ does not violate $o'$ iff $\pi' \models \neg \Diamond\, o'$ holds. For a second anti-oclet $o''$, the continuation $\pi \xrightarrow{o} \pi'$ also does not violate $o''$ iff $\pi' \models \neg \Diamond\, o'$ and $\pi' \models \neg \Diamond\, o''$ holds. Thus we can generalize this notion of non-violating continuations to a set of anti-oclets.

**Definition 8 (Non-violating continuation).** *Let $o$ be an oclet and let $A$ be a set of anti-oclets. Let $\pi, \pi'$ be labeled causal nets; $\pi'$ is a* non-violating *continuation of $\pi$ with $o$ wrt. $A$ iff $\pi \xrightarrow{o} \pi' \wedge \forall o' \in A : \pi' \models \neg \Diamond\, o'$. We write $\pi \xrightarrow{o \wedge \neg A} \pi'$ in this case.*

A non-violating continuation wrt. $A$ does not embed any anti-oclet $o \in A$. Because we exclude only those continuations that do violate an anti-oclet in $A$, the set of non-violating continuations is maximal. We may now close a set of processes in the right way by only considering the non-violating continuations.

A set of runs $R$ is *closed under non-violating continuations* with $o$ wrt. $A$ iff for each $\pi \in R$ each non-violating continuation of $\pi$ with $o$ wrt. $A$ is a run in $R$. With this notion, lifting semantics of an oclet to a set of oclets yields the formal *existential semantics* of oclet specifications.

**Definition 9 (Semantics of an oclet wrt. anti-oclets).** *A set of labeled causal nets $R$* satisfies *an oclet $o$ wrt. a set of anti-oclets $A$, $R \models (o \wedge \neg\, A)$, iff $R$ is prefix-closed and closed under non-violating continuations with $o$ wrt. $A$.*

**Definition 10 (Existential semantics).** *A set of labeled causal nets $R$* satisfies *an oclet specification $(Q, A)$, $R \models (Q, A)$ iff $R \models (o \wedge \neg\, A)$ for each $o \in Q$ and $R \models \neg\, o$ for each $o \in A$.*

With Def. 10, we can use qualified oclets and anti-oclets to formally specify the behavior of systems. A model, say a Petri net system, satisfies an oclet specification if the system's processes satisfy the intuitively understandable meaning of "if the precondition holds, the entire scenario must be executable as long as no anti-scenario is violated". In any other respect, the behavior of the system can be arbitrary.

## 5 Operational Semantics of Oclets

In the previous section, we established the existential, formal semantics of oclets for specifying systems. In this section, we turn an oclet specification into an *oclet system* having only the behavior that is specified by its oclets.

We define the *universal* semantics of oclets which is the behavior that can be constructed from a given set of oclets only. The universal semantics is operational as it defines exactly the actions that extend a given run. Such a semantics needs a specific point to begin with the construction; we define an oclet system.

**Definition 11 (Oclet system).** *Let $O$ be a label-consistent oclet specification, and $\pi_0$ be a process. Then $\Omega = (O, \pi_0)$ is an* oclet system.

Like for oclet specifications, we require that equally labeled transitions have equally label preplaces and equally labeled postplaces; see Sect. 3. For example, oclet specification $(\bigcup_{i=1}^{3}\{\mathsf{phil}(i)\}, \bigcup_{i=1}^{3}\{\mathsf{decentL}(i), \mathsf{decentR}(i)\}) =: \mathsf{Phil}_3^{\mathsf{dec}}$ yields the oclet system $\Omega_3^{\mathsf{dec}} = (\mathsf{Phil}_3^{\mathsf{dec}}, \pi_0)$ with $\pi_0$ having only the conditions $B_{\pi_0} := \{b_1, \ldots, b_6\}$ of Fig. 2; $\pi_0$ is called *initial* process.

Considering transition $t_1$ of $\mathsf{phil}(1)$, we see that the strict past of $t_1$, i.e. $\lfloor {}^\bullet t_1 \rfloor$, can be embedded in $\pi_0$ while $\lfloor t_1 \rfloor$ cannot be embedded. That is $\pi_0 \models \lfloor {}^\bullet t_1 \rfloor$. Simply said, $t_1$ is *enabled* in $\pi_0$. Transition $t_2$ of $\mathsf{decentL}(1)$ is not enabled in $\pi_0$ because there is no embedding of the entire $\lfloor {}^\bullet t_2 \rfloor$ at the end of $\pi_0$.

**Definition 12 (Enabled transition).** *Let $o$ be an oclet, let $t \in T_o \setminus pre_o$, and let $\pi$ be a labeled causal net. Transition $t$ is* enabled *in $\pi$ iff $\pi \models \lfloor {}^\bullet t \rfloor$.*

This definition of enabling a transition generalizes the definition for classical nets: In order to embed the preset of a transition in a process, all its predecessors must be embeddable. Thus in order to enable a transition of an oclet, the transition's history must have occurred. Transition $t_2$ of $\mathsf{decentL}(1)$ is enabled in $\pi_1$ (having nodes $\{b_1, \ldots, b_{10}, e_1, e_2\}$); the corresponding embedding $\alpha_2$ yields $\alpha_2({}^\bullet t_2) = \{b_8, b_9, b_{10}\}$.

Intuitively, *firing* an enabled transition $t$ means to continue a process $\pi$ with transition $t$ and its postset. We formalize this pattern as an *extension* of $\pi$: It consists of a new event $e_t$ that consumes from those conditions of $\pi$ that correspond to $t$'s preplaces and produces on new conditions that correspond to $t$'s postplaces.

**Definition 13 (Extension).** *Let $t$ be a transition of an oclet $o$, $t \in T_o \setminus pre_o$ that is enabled in $\pi$ by $\alpha$: $(\pi, \alpha) \models \lfloor {}^\bullet t \rfloor$. An* extension *of $\pi$ by $t$ at $\alpha$ is a net fragment $E_t^\alpha := (B_t, \{e_t\}, F_t^\alpha, \ell_t)$ with $B_t = \{b_p^* \mid p \in t^\bullet\}$, $B_t \cap B_\pi = \emptyset$, $e_t \notin E_\pi$,*

- *$F_t^\alpha = \{(b, e_t) \mid b \in \alpha({}^\bullet t)\} \cup \{(e_t, b_p^*) \mid b_p^* \in B_t\}$, and*
- *$\ell_t(e_t) = \ell_o(t)$ and $\ell_t(b_p^*) = \ell_o(p)$ for all $p \in t^\bullet$.*

In our example, the extension that corresponds to $t_1$ ($\mathsf{take}_1$) of $\mathsf{phil}(1)$ in $\pi_0$ consists of nodes $e_1$ and $b_7$ and all incoming arcs, especially $(b_1, e_1)$, etc. in Fig. 2. An extension is not a Petri net, because its arcs refer to nodes that are not part of the extension. We therefore call it a net fragment. To fire a transition in a process, append the corresponding extension to the process.

In our operational semantics, firing a transition must not violate an anti-oclet. Observe that transition $t_1$ of $\mathsf{phil}(1)$ is enabled in $\pi_1$ as well; the corresponding embedding $\alpha_3$ yields $\alpha_3({}^\bullet t_1) = \{b_8, b_9, b_{10}\} = \alpha_2({}^\bullet t_2)$ where $t_2$ is the contributing transition of anti-oclet $\mathsf{decentL}(1)$. Both transitions have the same label; firing $t_1$ of $\mathsf{phil}(1)$ in $\pi_1$ would violate $\mathsf{decentL}(1)$.

In general, a transition $t$ *would violate* an anti-oclet $o_a$ in a process $\pi$ iff $t$ is enabled in $\pi$ by $\alpha$ and $o_a$ has an equally-labeled transition $s \in (T_a \setminus pre_a)$ that is enabled in $\pi$ by $\alpha_a$ s.t. $t$ and $s$ denote the same occurrence: $\alpha({}^\bullet t) = \alpha_a({}^\bullet s)$. Firing a violating transition is forbidden. This interpretation yields the *processes* of an oclet system.

**Definition 14 (Processes of an oclet system).** *Let $\Omega = ((Q, A), \pi_0)$ be an oclet system. The set $Proc(\Omega)$ of all* processes *of $\Omega$ is the least set that satisfies:*

1. *$\pi_0$ is a* process *of $\Omega$ iff $\pi_0 \models \neg\, o_a$ for all $o_a \in A$.*
2. *Let $\pi \in Proc(\Omega)$. Let $o \in Q$ and let $t \in (T_o \setminus pre_o)$ be a transition that is enabled in $\pi$ and that would not violate any $o_a \in A$.*
   *The net $(\pi \oplus E^\alpha_t) := (B_\pi \cup B_t, E_\pi \cup \{e_t\}, F_\pi \cup F^\alpha_t, \ell_\pi \cup \ell_t)$ is a* process *of $\Omega$.*

Definition 14 completes the formal semantics of oclets. The entire process $\pi_3$ of Fig. 2 is a process of $\Omega_3^{\mathsf{dec}}$.

In the remainder of this section, we show that these definitions make sense. We prove that oclet systems are at least as expressive as elementary net systems. We finally show that existential and universal oclet semantics are consistent: the processes of an oclet system (universal semantics) satisfy its own specification (existential semantics). But first of all we show that all processes of an oclet system are labeled causal nets, i.e. that Def. 14 is formally sound.

**Lemma 1.** *Let $\Omega$ be an oclet system. If $\pi \in Proc(\Omega)$ is a process, and $E^\alpha_t$ is an extension of $\pi$, then $\pi \oplus E^\alpha_t \in Proc(\Omega)$ is a labeled causal net.*

*Proof.* Let $\Omega$, $\pi$, and $E^\alpha_t$ be as assumed. Let $o$ be the oclet of $\Omega$ with $t \in T_o$.

$\rho := \pi \oplus E^\alpha_t$ is a Petri net: $\pi$ is a net. $B_\rho \cap E_\rho = (B_\pi \cup B_t) \cap (E_\pi \cup \{e_t\}) = \emptyset$ by Def. 13 and 14. It is easy to see that extending $F_\pi$ with $F^\alpha_t$ preserves the bipartite structure of nets and that the union of $\ell_\pi$ and $\ell_t$ is well-defined.

$\rho$ is a causal net: (1) $\pi$ and $E^\alpha_t$ are acylic and disjoint. Further, $E^\alpha_t$ has no arc $(x, y)$ with $y$ in $\pi$. Thus there exists no arc from $E^\alpha_t$ into $\pi$ to close a cycle in $\rho$. (2) In $\pi \oplus E^\alpha_t$, each node has only finitely many predecessors because $\pi$ is a causal net and $E^\alpha_t$ is finite. (3) Transition $t$ is enabled in $\pi$ (by Def. 13) which implies $\alpha({}^\bullet t) \subseteq \max_\pi(B_\pi \cup E_\pi)$ (by Def. 12). Thus each $b \in \alpha({}^\bullet t)$ has no successor in $\pi$. In $\rho$, each $b \in \alpha({}^\bullet t)$ has only one successor: $e_t$ (by Def. 13 and 14). The new postconditions $B_t \subseteq \max_\rho(E_\rho \cup B_\rho)$ have no successor and only one predecessor: $e_t$ (by Def. 13). Thus $\pi$ is a labeled causal net as defined in Sect. 3. □

The operational semantics of oclets is complete wrt. the partial order semantics of Petri net systems.

**Theorem 1.** *Let $N = (P, T, F, m_0)$ be a Petri net system with initial marking $m_0$. Then there exists an oclet system $\Omega_N$ s.t. the set of processes of $N$ and the set of processes of $\Omega_N$ are equal.*

*Proof.* We show completeness by defining an algorithm that constructs for each $N$ an oclet system $\Omega_N$ that has exactly the same processes as $N$.

Let $t \in T$. We define an atomic oclet $o_t$ that specifies the firing of transition $t$: $o_t := (P_o, T_o, F_o, \ell_o, pre_o)$ with $P_o = {}^\bullet t \cup t^\bullet$, $T_o = \{t\}$, $F_o$ the restriction of $F_N$ to $(P_o \times T_o) \cup (T_o \times P_o)$, $\ell_o$ the identity on $P_o \cup T_o$, and $pre_o = {}^\bullet t$; c.f. Fig. 6. The structure $o_t$ is an oclet by Def. 3. Define $\Omega_N := ((\{o_t \mid t \in T\}, \emptyset), \pi_0)$ with initial process $\pi_0$ consisting only of the set of conditions $B_{\pi_0} := \{b_1, \ldots, b_k \mid \exists p \in P_N : m_0(p) = k, \ell_{\pi_0}(b_i) = p, i = 1, \ldots, k\}$; $\Omega_N$ is an oclet system by Def. 14.

We prove the equivalence of processes by induction on the number of events in them. Firstly, $\pi_0$ is the initial process of $N$ iff it is the initial process of $\Omega_N$, which holds by construction. Let $\pi$ be a labeled causal net containing $n$ events. By inductive assumption, $\pi$ is a process of $N$ iff it is a process of $\Omega_N$.

By definition of partial order semantics of nets, $\pi$ reaches the marking $m$ with $m(p) = |\{b \in \max \pi \mid \ell_\pi(x) = p\}|$ for each place $p$. Let $T_m$ be the set of transitions that are enabled in $m$. By construction of $\Omega_N$ holds $t \in T_m$ iff there exists oclet $o_t$ in $\Omega_N$ with transition $t$ that is enabled in $\pi$ according to Def. 12.

If $T_m = \emptyset$, $\pi$ cannot be extended by $N$ and by $\Omega_N$. Otherwise, let $t \in T_m$ with ${}^\bullet t = \{p_1, \ldots, p_k\}$ and $t^\bullet = \{q_1, \ldots, q_l\}$; firing $t$ according to the Petri net semantics constructs process $\rho$ by adding a new $t$-labeled event $e$ with preconditions ${}^\bullet e = \{b_1, \ldots, b_k\} \subseteq \max_\pi(B_\pi \cup E_\pi)$ with $\ell_\rho(b_i) = p_i, i = 1, \ldots, k$ and new postconditions $e^\bullet = \{b_1^*, \ldots, b_l^*\}$ with $\ell_\rho(b_i^*) = q_i, i = 1, \ldots, l$. Because $\Omega_N$ has no anti-oclets, there exists an embedding $\alpha$ with $\pi \oplus E_t^\alpha \in Proc(\Omega)$ (Def. 14). By definition of $o_t$ and by Def. 13 holds $\pi \oplus E_t^\alpha = \rho$. Thus $\rho$, with $n+1$ events, is a process of $N$ iff it is a process of $\Omega_N$. □

Theorem 1 relates oclet systems to classical Petri net systems. The following theorem relates the universal semantics of oclet systems to the existential semantics of oclet specifications.

**Theorem 2.** *Let $\Omega = (O, \pi_0)$ be an oclet system. $Proc(\Omega) \models O$.*

We prove Thm. 2 by the help of two lemmata. The first technical lemma states that every complete prefix of a continuation is a continuation as well. The second lemma proves that the processes of an oclet system are closed under non-violating continuations.

**Lemma 2.** *Let $\pi$ be a process, let $o$ be an oclet, and let $\pi \xrightarrow{o} \pi_2$. Let $\pi_1$ be a complete prefix of $\pi_2$, i.e. $\pi_1$ contains all postconditions of its events (Def. 2), s.t. $\pi$ is a prefix of $\pi_1$. Then $\pi \xrightarrow{o} \pi_1$.*

*Proof.* We construct the prefix $o[X_1]$ of $o$ that is embedded at the end of $\pi_1$ according to Def. 6. The nodes $X_\Delta := (B_2 \cup E_2) \setminus (B_1 \cup E_1)$ are not in $\pi_1$.

Because $\pi \xrightarrow{o} \pi_2$, there exists a prefix $o[X_2]$ of $o$ with $pre_o \subseteq X_2$ and embeddings $\alpha$ and $\alpha_2$ with $(\pi, \alpha) \models o[pre_o]$ and $(\pi_2, \alpha_2) \models o[X_2]$; see Def. 6.

By Def. 5, $\alpha_2$ is injective. The nodes $X_1 := X_2 \setminus \alpha_2^{-1}(X_\Delta)$ are all oclet nodes that are embedded into $\pi_1$: Because $\pi_1$ is a complete prefix of $\pi_2$ and by $\alpha_2$ being injective follows $o[X_1]$ is a complete prefix of $o[X_2]$. The restricted embedding $\alpha_1 := \alpha_2|_{X_1}$ embeds $o[X_1]$ at the end of $\pi_1$: $(\pi_1, \alpha_1) \models o[X_1]$.

From $\pi$ being prefix of $\pi_1$ follows $(B \cup E) \cap X_\Delta = \emptyset$. Thus $\alpha(pre_o) \cap X_\Delta = \emptyset$ holds, which implies $pre_o \cap \alpha_2^{-1}(X_\Delta) = \emptyset$. Hence $pre_o \subseteq X_2$ and $X_1 = X_2 \setminus \alpha_2^{-1}(X_\Delta)$ imply $pre_o \subseteq X_1$. Thus $\pi \xrightarrow{o} \pi_1$ by Def. 6. □

**Lemma 3.** *Let $\Omega = ((Q, A), \pi_0)$ be an oclet system. Let $o \in Q$ and let $\pi \in Proc(\Omega)$. Then every process $\pi_2$ with $\pi \xrightarrow{o \wedge \neg A} \pi_2$ is a process of $\Omega$.*

*Proof.* We prove the property by induction on the number $n$ of new events in $\pi_2$, $n = |E_{\pi_2} \setminus E_\pi|$. For $n = 0$ we have $\pi_2 = \pi$ by Def. 6; thus $\pi_2 \in Proc(\Omega)$.

Consider $n > 0$: Let $\pi_2$ be a non-violating continuation of $\pi$; $\pi_2$ contains a new event $e \in (E_{\pi_2} \setminus E_\pi)$ that has no successor event, i.e. there ex. no $e' \in E_{\pi_2}, e \neq e'$ with $e \leq_{\pi_2} e'$. Let $\pi_1$ be the prefix of $\pi_2$ which we obtain by removing $e$ and $e^\bullet$ from $\pi_2$. Effectively, we remove events $E^* = \{e\}$, conditions $B^* = e^\bullet$, and arcs $F^* = \{(b, e) \mid b \in {}^\bullet e\} \cup \{(e, b) \mid b \in B^*\}$.

Because $e \notin E_\pi$, $\pi$ is a prefix of $\pi_1$. From Lemma 2 follows that $\pi_1$ is a continuation of $\pi$. Trivially, $\pi_1$ does not violate any anti-oclet of $A$ because $\pi_2$ does not. Thus by inductive assumption, $\pi_1 \in Proc(\Omega)$.

We have to show that $\pi_2 \in Proc(\Omega)$. From the definition of continuation (Def. 6) follows that some prefix $o[X_2]$ of $o$ is embedded at the end of $\pi_2$ by some embedding $\alpha_2$. Thus there exists a transition $t \in X_2$ with $\alpha_2(t) = e$ which we removed from $\pi_2$ to obtain $\pi_1$. We fire $t$ in $\pi_1$ to construct $\pi_2$:

Because $e$ and $e^\bullet$ were removed, the prefix $o[X_1]$, $X_1 := X_2 \setminus (t \cup t^\bullet)$ of $o$ is embedded at the end of $\pi_1$ by $\alpha_1 := \alpha_2|_{X_1}$. Then $(\pi_1, \alpha_1) \models \lfloor {}^\bullet t \rfloor$, i.e. $t$ is enabled in $\pi_1$ (Def. 12). From Def. 13 follows that $E_t^{\alpha_1} = (B^*, E^*, F^*, \ell_{\pi_2}|_{B^* \cup E^*})$ as removed from $\pi_2$. Thus $\pi_1 \oplus E_t^{\alpha_1} = \pi_2$. By Def. 14, $\pi_2$ is a process of $\Omega$. □

With Lem. 3 we have proven that the processes an oclet system are closed under non-violating continuations. The proof also shows how declarative and operational semantics are related to each other by the notion of local steps. The proof of Thm. 2 is now straight forward.

*Proof (of Thm. 2).* Let $\Omega = (O, \pi_0)$ be an oclet system with $O = (Q, A)$. We have to show $Proc(\Omega) \models O$ according to Def. 10. The set $Proc(\Omega)$ is prefix-closed by construction. From Lemma 3 follows that $Proc(\Omega)$ is closed under non-violating continuations with any qualified oclet of $\Omega$ wrt. $A$. Thus $Proc(\Omega) \models (o \wedge \neg A)$ for each $o \in Q$.

It remains to show that no process of $\Omega$ violates any anti-oclet $o_v \in A$: Assume there is a run $\pi_v \in Proc(\Omega), \pi_v \neq \pi_0$ and an embedding $\alpha_v$ that violates $o_v$: $(\pi_v, \alpha_v) \models \Diamond\, o_v$. Because $Proc(\Omega)$ is prefix-closed, we may assume that $(\pi_v, \alpha_v) \models o_v$ holds. From Def. 14 follows that there exists $\pi \in Proc(\Omega)$, an oclet $o \in Q$, and a transition $t$ of $o$ that is enabled in $\pi$ by an embedding $\alpha$ with $\pi_v = \pi \oplus E_t^\alpha$. But then, $o_v$ contains transition $s$ with $\ell_{o_v}(s) = \ell_o(t)$ and $\alpha_v({}^\bullet s) = \alpha({}^\bullet t)$. Hence $t$ would violate $o_v$. This contradicts $\pi_v = \pi \oplus E_t^\alpha \in Proc(\Omega)$ by Def. 14. Thus $Proc(\Omega) \models \neg\, o$ for each $o \in A$. □

We just have shown that the universal semantics of oclet systems imply the existential semantics of oclet specifications. The semantics are not equivalent in general, as any set of processes, that contain labels that do not occur in the specification, cannot be constructed with the universal semantics.

The behavior that satisfies an oclet specification $(Q, A)$ but that cannot be constructed by an oclet system $\Omega = ((Q, A), \pi_0)$ violates the following *closed-world assumption* of the universal semantics. A set of runs $R$ is *closed* wrt. $\Omega$ iff for all $\pi \in R$, $\pi_0$ is a complete prefix of $\pi$ and for each node $x \in (B_\pi \cup E_\pi)$ exists a qualified oclet $o$ of $O$ that contributes this node to $\pi$, i.e. there exists $y \in P_o \cup T_o$ and embedding $\alpha : \lfloor y \rfloor \to (B_\pi \cup E_\pi)$ with $\alpha(y) = x$. From the inductive definition of the processes of an oclet system follows that if $Proc(\Omega) \subset R$, then $R$ is not closed wrt. $\Omega$.

## 6 Modeling with Oclets

In the previous three sections, we defined the formal semantics of oclets. With this semantics, the oclet system $\Omega_3^{\mathsf{dec}} = (\mathsf{Phil}_3^{\mathsf{dec}}, \pi_0)$ with $\mathsf{Phil}_3^{\mathsf{dec}} := (\bigcup_{i=1}^{3}\{\mathsf{phil}(i)\}, \bigcup_{i=1}^{3}\{\mathsf{decentL}(i), \mathsf{decentR}(i)\})$ of our introductory Sect. 2 has only decent runs by construction. The run $\pi_3$ of Fig. 2 is a process of $\Omega_3^{\mathsf{dec}}$.

To give a better understanding for the use of oclet systems, we explain two application scenarios for oclets in the section. First, we use oclets to specify the solution of $n$ dining philosophers with only $n - 1$ forks available. Secondly, we sketch how oclets allow to quickly *adapt* system models.

### 6.1 Scenario-Based System Design with Oclets

We consider a variant of the dining philosophers: The philosophers are still sitting around a table and alternate between thinking and eating by taking and releasing forks they share with their neighbors. Unfortunately, one fork $\mathsf{f}_i$ is missing; philosophers $i$ and $i \oplus 1$ cannot eat. The task: *Specify a system of $n$ philosophers with $n - 1$ forks s.t. each philosopher always eventually eats.*

In our solution, the philosophers pass the forks around the table: Every philosopher $i \oplus 1$, who is missing his right fork $i \oplus 2$ may request the left fork of his left neighbor $i$. Philosopher $i$ grants this request by passing his left fork $i$ to his right neighbor $i \oplus 1$, who put this fork to his right. Now, phil. $i \ominus 1$ is missing his right fork $i$; he may send a request to his left neighbor. To allow each phil. $i$ to eat at least once before passing his left fork, he may do so only after he has just returned from eating. Oclet $\mathsf{exchF}(i)$ of Figure 7 denotes exactly this specification.

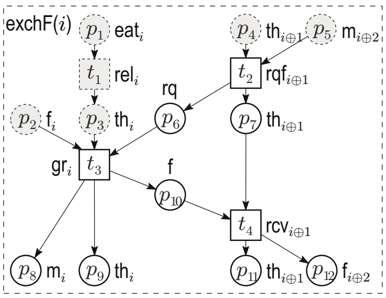


**Fig. 7.** Oclet $\mathsf{exchF}(i)$ specifying the fork-passing protocol

Now, consider the oclet system $\Omega_3^{\mathsf{m}} := (\mathsf{Phil}_3^{\mathsf{m}}, \pi_0^{\mathsf{m}})$ with oclet specification $\mathsf{Phil}_3^{\mathsf{m}} := (\bigcup_{i=1}^{3}\{\mathsf{phil}(i), \mathsf{exchF}(i)\}, \emptyset)$ and initial process $\pi_0^{\mathsf{m}}$ having initial conditions with labels $\mathsf{f}_1, \mathsf{th}_1, \mathsf{f}_2, \mathsf{th}_2, \mathsf{m}_3, \mathsf{th}_3$. Figure 8 depicts a process $\pi_4^{\mathsf{m}}$ of $\Omega_3^{\mathsf{m}}$: In $\pi_0^{\mathsf{m}}$, transition $\mathsf{take}_1$ of $\mathsf{phil}(1)$ and $\mathsf{rqf}_2$ of $\mathsf{exchF}(1)$ are enabled concurrently. Because we made the enabling of a transition dependent only its own history (instead of the entire precondition of its oclet), it is possible to start executing a scenario even if not the entire precondition was observed. Theorem 2 justifies this behavior. Firing $\mathsf{take}_1$ and $\mathsf{rqf}_2$ yields $\pi_1^{\mathsf{m}}$ (nodes $b_1, \ldots, b_9, e_1, e_2$).

In $\pi_1^{\mathsf{m}}$, only $\mathsf{rel}_1$ of $\mathsf{phil}(1)$ is enabled, yielding $\pi_2^{\mathsf{m}}$ (nodes $b_1, \ldots, b_{12}$, $e_1, \ldots, e_3$). Transitions $\mathsf{take}_1$ of $\mathsf{phil}(1)$, and $\mathsf{gr}_1$ of $\mathsf{exchF}(1)$ are enabled in conflict: their presets can only be overlappingly embedded. Firing $\mathsf{gr}_1$ and the subsequently enabled $\mathsf{rcv}_2$ of $\mathsf{exchF}(1)$ constructs $\pi_3^{\mathsf{m}}$ $(b_1, \ldots, b_{17}, e_1, \ldots, e_5)$ where philosopher 2 can now take both forks. Continuing with the construction, we reach $\pi_4^{\mathsf{m}}$ where now philosopher 2 has to choose whether to grant the request of philosopher 3 or whether to take the forks again.

The system $\Omega_3^{\mathsf{m}}$ solves the missing fork problem for 3 philosophers, but has non-decent runs. We can easily refine $\mathsf{Phil}_3^{\mathsf{m}}$ to $\mathsf{Phil}_3^{\mathsf{m,d}}$ by adding anti-oclets $\mathsf{decentL}(i)$ and $\mathsf{decentR}(i)$ of Fig. 4 for $i = 1, 2, 3$. Process $\pi_4^{\mathsf{m}}$ is also a process of the refined system $\Omega_3^{\mathsf{m,d}} := (\mathsf{Phil}_3^{\mathsf{m,d}}, \pi_0^{\mathsf{m}})$, and cannot be extended by $e_9$ ($\mathsf{take}_2$) because of $t_2$ of $\mathsf{decentL}(2)$. The system $\Omega_3^{\mathsf{m,d}}$ has only decent runs by construction.

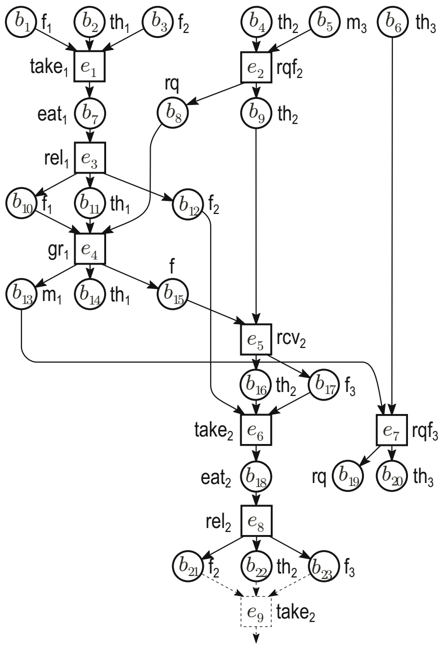


**Fig. 8.** A process $\pi_4^{\mathsf{m}}$ of the fork-passing philosophers

This solution has another unfair run in case of more than three philosophers: Assume phil. 2 requests and receives fork $\mathsf{f}_1$ from phil. 1 and puts it as fork $\mathsf{f}_3$ between phil. 2 and phil. 3. Fork $\mathsf{f}_3$ can equally be taken from phils. 2 and 3. Meanwhile, the other philosophers may have kept on passing forks until phil. 4 requests $\mathsf{f}_3$ from 3. If now phil. 3 takes and releases his forks, and then grants the request of 4, phil. 2 was not able to eat with the forks he just has requested. The specification can easily be extended with an anti-oclet to prevent this behavior.

## 6.2 Adapting System Models with Oclets

This rather flexible style of creating oclet specifications also helps when specifying systems that have to be adapted frequently. Processes in disaster response are such as case, where the system model must be adapted to incorporate changes of the real-world processes [3].

As it is fairly easy to add, remove, and modify single scenarios, the changes are well-conceivable and do not break the model. Because our operational semantics makes no assumptions regarding the initial process, adaptation can be done as follows: Construct a process $\pi$ of an oclet system $(O, \pi_0)$ until a problem is encountered. Change specification $O$ by adding, removing, or modifying oclets; $O \to O'$. Then continue in the system $(O', \pi)$. Iterate this procedure, possibly beginning again at $\pi_0$ or $\pi$, until the system is adapted. Our formal semantics guarantees well-defined behavior at any time during adaptation.

## 7 Related Work

In this section, we compare our approach for scenario-based modeling with Petri nets to existing works.

MSCs formalize scenarios as partial orders on events; several extensions are available. Hierarchical MSCs (HMSCs) and Message Sequence Graphs (MSGs) explicitly denote in a graph how scenarios may be concatenated, for specifying entire systems. Operational semantics of (H)MSCs and MSGs translate a specification into process algebraic expressions [5], automata [4], or employ graph grammars to construct runs [7] from MSCs. These, as well as existing Petri net semantics like [11] do not support anti-scenarios.

LSCs are an extension of MSCs with a formal semantics for overlapping scenarios, anti-scenarios, and modalities for scenarios and events. LSCs are more expressive than oclets; the original LSC semantics is declarative. Operational semantics for LSCs, i.e. LSC play-out, is defined by a translation to state charts [6], or by constructing an automaton from a specification [12]. Unfortunately, this linearizes the partial order explicitly specified in the charts.

Desel et al brought up the approach of scenario-based system design and validation with Petri nets [13,2]. The principle idea is to let the system designer denote desired and undesired behavior as complete (finite) partially ordered runs, i.e. complete scenarios and anti-scenarios. These mediate between a formal specification and a system model (a Petri net): Specification and model are validated against the scenarios, that is, whether each scenario satisfies the specification and whether the system model executes the desired scenarios while disallowing the undesired ones. The modeler iteratively reaches a valid system model; thereby refinement of the system model is a creative step involving human interaction. This step can be supported by folding desired scenarios into an overapproximating Petri net [2]. In [14], Bergenthum et al show how an equivalently implementing Petri net can be synthesized from a complete set of finite desired runs. The approach is extended in [15] where desired behavior is given as a regular expression over finite scenarios.

The oclet model follows this idea of scenario-based system design. Oclets contribute history-based preconditions to scenarios allowing that a scenario specifies behavior "in the middle" of an execution. Thereby the composition of scenarios to complete runs follows from the oclet's inherent precondition requiring no further notion like an expression for composition. Our operational semantics allows to execute a set of scenarios directly without the need for additional synthesis or transformation. In that respect, our semantics makes a set of scenarios a *complete* system model. Still, a synthesis into Petri nets as in [14,15] allows to use the entire Petri net theory for verification.

The concept of history-dependent firing of transitions has been proposed earlier by defined corresponding transiting guards [16]; oclets provide a graphical syntax for a subclass of these guards. The net composition techniques defined in [17] are a general case of the net composition employed in our model. In the context of adapting system models, existing works in the area of adaptive workflows, see [18] for a survey, use models with sequential semantics or require to denote adaptations in explicit model transformation rules. Graph transformations on nets also require explicit adaptations rules for adaptations, e.g. [19]. In comparison, adaptations of oclet systems can be done from the perspective of desired and undesired scenarios only.

## 8 Conclusion

We presented a novel formal model for specifying and modeling systems with Petri net scenarios. We defined a specification to be a set of oclets, labeled causal nets with a dedicated precondition; oclets are partitioned into qualified oclets and anti-oclet describing desired and forbidden behavior, respectively.

We defined a declarative semantics that characterizes sets of runs that satisfy a given specification. We then provided an operational semantics to construct a maximal set of satisfying runs; we have shown that any run, that cannot be constructed either violates the specification, or includes an action that is not defined in the specification. We solved the dining philosophers problem in two variants to illustrate how our model can be used for modeling distributed systems. Providing an operational, partial-order Petri net semantics for scenarios *and* anti-scenarios makes our work a contribution in the area of scenario-based techniques.

Our results hint to further research: We already have first results towards constructing the complete finite prefix of a branching process of an oclet system [10] which allows for the verification of oclet systems (the full branching process can already be constructed with the given semantics). These results also hint towards a synthesis of Petri nets from scenarios and anti-scenarios. We also research structural properties of oclet specifications to derive system properties directly from scenarios and intend to introduce modalities known from LSCs such as imperative scenarios and events.

Our approach is implemented in our graphical runtime environment Greta, that is available online at `http://www.service-technology.org/greta/` together with several example specifications.

**Acknowledgements.** This paper has greatly benefitted from discussions with and suggestions by Wolfgang Reisig, Karsten Wolf, Peter Massuthe, and all referees of this paper. Our tool Greta, which substantially helped developing the concepts would not have been possible without the work of Manja Wolf. Dirk Fahland is funded by the DFG-Graduiertenkolleg 1324 "METRIK".

## References

1. Damm, W., Harel, D.: LSCs: Breathing Life into Message Sequence Charts. Form. Methods Syst. Des. 19(1), 45–80 (2001)
2. Desel, J.: From human knowledge to process models. In: UNISCON, pp. 84–95 (2008)
3. Fahland, D., Woith, H.: Towards process models for disaster response. In: Ardagna, D., et al. (eds.) BPM 2008 Workshops. LNBIP, vol. 17, pp. 244–256. Springer, Heidelberg (2008)
4. Mukund, M., Kumar, K.N., Thiagarajan, P.S.: Netcharts: Bridging the gap between HMSCs and executable specifications. In: Amadio, R., Lugiez, D. (eds.) CONCUR 2003. LNCS, vol. 2761, pp. 293–307. Springer, Heidelberg (2003)
5. Mauw, S., Reniers, M.A.: An algebraic semantics of Basic Message Sequence Charts. The Computer Journal 37, 269–277 (1994)
6. Harel, D., Kugler, H.: Synthesizing State-Based Object Systems from LSC Specifications. In: Yu, S., Păun, A. (eds.) CIAA 2000. LNCS, vol. 2088, pp. 1–33. Springer, Heidelberg (2001)
7. Hélouët, L., Jard, C., Caillaud, B.: An event structure based semantics for high-level message sequence charts. Mathematical. Structures in Comp. Sci. 12(4), 377–402 (2002)
8. Engelfriet, J.: Branching processes of Petri nets. Acta Inf. 28(6), 575–591 (1991)
9. Reisig, W.: Elements Of Distributed Algorithms: Modeling and Analysis with Petri Nets. Springer, Heidelberg (1998)
10. Esparza, J., Heljanko, K.: Unfoldings - A Partial-Order Approach to Model Checking. Springer, Heidelberg (2008)
11. Kluge, O.: Petri nets as a semantic model for Message Sequence Chart specifications. In: INT 2002, Grenoble, France, pp. 138–147 (2002)
12. Harel, D., Kugler, H., Marelly, R., Pnueli, A.: Smart play-out of behavioral requirements. In: Aagaard, M.D., O'Leary, J.W. (eds.) FMCAD 2002. LNCS, vol. 2517, pp. 378–398. Springer, Heidelberg (2002)
13. Desel, J., Juhás, G., Lorenz, R., Neumair, C.: Modelling and validation with VipTool. In: van der Aalst, W.M.P., ter Hofstede, A.H.M., Weske, M. (eds.) BPM 2003. LNCS, vol. 2678, pp. 380–389. Springer, Heidelberg (2003)
14. Bergenthum, R., Desel, J., Lorenz, R., Mauser, S.: Synthesis of Petri Nets from Finite Partial Languages. Fundam. Inform. 88(4), 437–468 (2008)
15. Bergenthum, R., Mauser, S.: Synthesis of Petri Nets from Infinite Partial Languages with VipTool. In: AWPN 2008, Rostock, Germany, University of Rostock (September 2008)
16. Hee, K., Serebrenik, A., Sidorova, N., Voorhoeve, M., Werf, J.: Modelling with History-Dependent Petri Nets. In: Alonso, G., Dadam, P., Rosemann, M. (eds.) BPM 2007. LNCS, vol. 4714, pp. 320–327. Springer, Heidelberg (2007)
17. Barros, J.a.P., Gomes, L.: Net model composition and modification by net operations: a pragmatic approach. In: Proceedings of INDIN 2004, Berlin, Germany (June 2004)
18. Rinderle, S., Reichert, M., Dadam, P.: Evaluation of correctness criteria for dynamic workflow changes. In: van der Aalst, W.M.P., ter Hofstede, A.H.M., Weske, M. (eds.) BPM 2003. LNCS, vol. 2678, pp. 41–57. Springer, Heidelberg (2003)
19. Ehrig, H., Hoffmann, K., Padberg, J., Prange, U., Ermel, C.: Independence of net transformations and token firing in reconfigurable place/transition systems. In: Kleijn, J., Yakovlev, A. (eds.) ICATPN 2007. LNCS, vol. 4546, pp. 104–123. Springer, Heidelberg (2007)

# Hornets: Nets within Nets Combined with Net Algebra

Michael Köhler-Bußmeier

University of Hamburg, Department of Informatics
Vogt-Kölln-Str. 30, D-22527 Hamburg
koehler@informatik.uni-hamburg.de

**Abstract.** In this contribution we propose an algebraic extension of object nets. Object nets, also known as *nets within nets*, allow nets itself as tokens. The algebraic structure introduced here refers to the topology of these net-tokens, i.e. we have operators which compose nets. Object nets that use net operations in arc expression are called *Higher Order Recursive Nets*, or short: *Hornets*.

The operations on nets allow to modify the structure of net-tokens at run-time. We apply this construct to the workflow management domain. We propose a simple Hornet model of a *distributed workflow management system*. This system consists of a network of workflow management agents. The agents cooperatively transfer workflows over the network for distributed execution, monitor their processes, and reorganise the workflow repository to improve e.g. the system's performance.

## 1 Motivation: Distributed Workflow Management

In this contribution we apply the *nets within nets* [1] approach to the domain of *workflow nets* and their management. The combination of both research directions was started in [2] as a joint work of Wil van der Aalst and the Hamburg group: The paper considered workflows that are executed within one organisation, which is itself a net. This scenario has been modelled using object nets and the model was executed using the Renew tool.

Both groups undertook further research which brought both areas, object nets and workflow nets, closer together: The Hamburg group investigated properties of object nets [3,4,5,6] and used them to model workflows as well as agent interaction protocols [7,8,9]. The Eindhoven group extended workflow nets with composition and hierarchy, resulting in *adaptive workflow nets* [10,11]: Workflow nets [12] may be composed sequentially, parallel, or in alternation – denoted $(N_1 \cdot N_2)$, $(N_1 \| N_2)$, and $(N_1 + N_2)$ – as depicted in Figure 1. Additionally, adaptive workflow nets are considered as tokens of other nets which control their execution.

In this paper we want to narrow the gap a little bit more and will show how adaptive workflow nets could be expressed in terms of object nets. Hierarchy is already a feature of object nets while composition is not. Therefore we consider an extension of object nets, called Hornets, which allow arbitrary algebraic operations on nets.

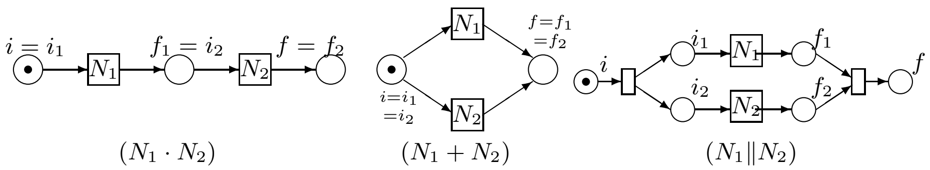


**Fig. 1.** Composition of Workflow Nets

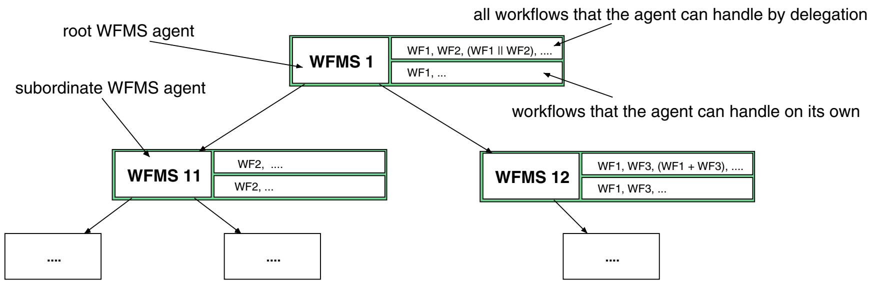

all workflows that the agent can handle by delegation
root WFMS agent
WFMS 1
WF1, WF2, (WF1 || WF2), ....
WF1, ...
subordinate WFMS agent
workflows that the agent can handle on its own
WFMS 11
WF2, ....
WF2, ...
WFMS 12
WF1, WF3, (WF1 + WF3), ....
WF1, WF3, ...
....
....
....

**Fig. 2.** The Workflow Management Network

We will illustrate the modelling power of Hornets by formalising a Workflow Management System (WFMS). We even allow to model a *distributed* WFMS, that consists of a network of WFMS-agents. Workflows are decomposed into smaller parts, distributed over the network and executed by several agents. Each WFMS-agent specifies those workflows that it can execute locally and those that it executes by delegating parts to other agents. In the following we consider a system with a tree-like structure as in Figure 2.

Following this aim the paper has the structure: Section 2 defines Hornets as the algebraic generalisation of object nets and Section 3 presents the model of the distributed WFMS including parts like analysis, repository management, delegation information management, and workflow execution. The paper closes with a conclusion.

## 2 Hornets: Higher Order Recursive Nets

In this paper we define the algebraic extension of object nets [1], i.e. Petri nets with nets as tokens. This nesting may be limited as in [13] or unlimited as in [3,4,5,6].

Among the wealth of research on defining nested systems, in recent years a variety of formalisms have been introduced: Calculi based approaches are e.g. the ambient calculus [14] (a relative of the $\pi$-calculus [15]) and Petri net based ones are e.g. mobile nets [16], recursive nets [17], nested nets [18], mobile pr/t nets [19], PN$^2$ [20], hypernets [21], mobile systems [22], AHO systems [23], adaptive workflow nets [10], and RN systems [24].

The algebraic extensions considered here are not introduced to have data types which replace the anonymous black tokens, like done by algebraic nets [25]. The algebra considered here is defined on the nets themself and allows to modify the structure of the net-tokens as a result of a firing transition. These object nets are called *Higher Order Recursive Nets*, or Hornets for short.

*Example 1.* We consider a Hornet with two workflow nets $N_1$ and $N_2$ as tokens – cf. Figure 3. To model a run-time adaption, we combine $N_1$ and $N_2$ resulting in the net $N_3 = (N_1 \| N_2)$. This modification is modelled by transition $t$ of the Hornets in Fig. 3. In a binding $a$ with $x \mapsto N_1$ and $y \mapsto N_2$ the transition $t$ is enabled. Assume that $(x\|y)$ evaluates to $N_3$ for $a$. If $t$ fires it removes the two net-tokens from $p$ and $q$ and generates one new net-token on place $r$. The net-token on $r$ has the structure of $N_3$ and its marking is obtained as a transfer from the token on $v$ in $N_1$ and the token on $s$ in $N_2$ into $N_3$. This transfer is possible since all the places of $N_1$ and $N_2$ are also places in $N_3$ and tokens can be transferred in the obvious way.

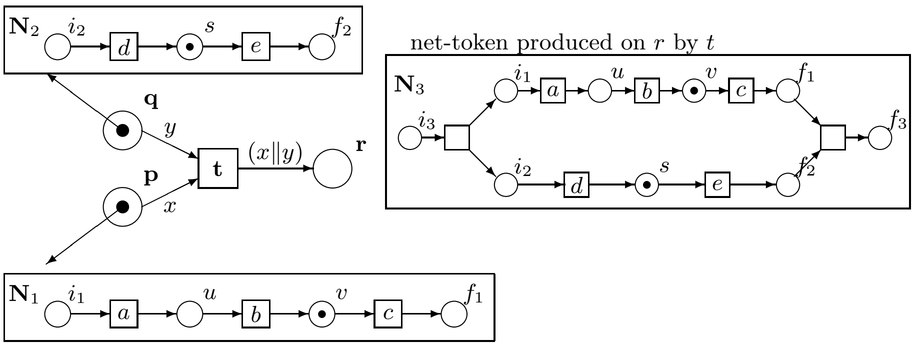


**Fig. 3.** Modification of the net-token's structure

The use of algebraic transformation in Hornets relates them to *algebraic higher-order (AHO) systems* [23], which are restricted to two-levelled systems but have a greater flexibility for the operations on net-tokens, since each net transformation is allowed.

## 2.1 Signatures, -Theories and -Logic

We define the algebraic structure of object nets. For a general introduction of algebraic specifications cf. [26].

Let $K$ be a set of net-types (kinds). We assume the type $\bullet \in K$ of anonymous, so called 'black' tokens (as in p/t nets).

A *signature* describes the set of all object net terms. A signature is a disjoint family $\Sigma = (\Sigma_{k_1 \cdots k_n, k})_{k_1, \cdots, k_n, k \in K}$ of operators. (For $K$-indexed families we use abbreviations like $\sigma \in \Sigma$ instead of $\sigma \in \bigcup_{k_1,\cdots,k_n,k\in K} \Sigma_{k_1\cdots k_n,k}$ if there is no danger of confusion.) The set $\Sigma_{\lambda,k}$ are the constants of type $k$.

Let $X = (X_k)_{k\in K}$ be a countable family of disjoint sets of variables, all disjoint to operators. The set of terms $\mathbb{T}^k_\Sigma(X)$ of type $k$ over a signature $\Sigma$ is defined as follows: Each variable $x \in X_k$ is a term of type $k$; for an operator $\sigma \in \Sigma_{k_1\cdots k_n,k}$ and terms $t_1 \in \mathbb{T}^{k_1}_\Sigma(X), \ldots, t_n \in \mathbb{T}^{k_n}_\Sigma(X)$ we have that $\sigma(t_1,\ldots,t_n) \in \mathbb{T}^k_\Sigma(X)$ is a term of type $k$. The set of all terms is $\mathbb{T}_\Sigma(X) := \bigcup_{k\in K} \mathbb{T}^k_\Sigma(X)$.

Let $\Sigma$ be a signature over $K$. For each type $k \in K$ there is a set of axioms $E_k$. An *axiom* is a pair of terms $(t_1,t_2) \in E_k$ with $t_1,t_2 \in \mathbb{T}^k_\Sigma(X)$, denoted as $\forall X : (t_1 \sim t_2)$.

A *specification* $(\Sigma, X, E)$ consists of a signature $\Sigma$, a family of variables $X = (X_k)_{k\in K}$, and a family of axioms $E = (E_k)_{k\in K}$.

Let $(\Sigma, X, E)$ be a specification. In the following we use (many-sorted) predicate logic, where the terms are generated by a signature $\Sigma$. In addition to the signature $\Sigma$ and the variables $X$ we define a family of typed predicates $\Psi = (\Psi_{k_1\cdots k_n})_{k_1,\ldots,k_n\in K}$. Together with the signature $\Sigma$ this defines the set of *formulae* in the standard way for predicate logic:
(i) If $\psi \in \Psi_{k_1\cdots k_n}$ is a predicate and $t_1 \in \mathbb{T}^{k_1}_\Sigma(X), \ldots t_n \in \mathbb{T}^{k_n}_\Sigma(X)$ are terms, then $\psi(t_1,\ldots,t_n)$ is a formulae.
(ii) If $F_1$ and $F_2$ are formulae and $x \in X$ is a variable, then $\neg F_1$, $(F_1 \wedge F_2)$, $(F_1 \vee F_2)$, $(F_1 \Longrightarrow F_2)$, $\forall x : F_1$, and $\exists x : F_1$ are formulae.

The set of formulae is denoted $PL_\Gamma$ where $\Gamma = (\Sigma, X, E, \Psi)$ defines the *logic structure*.

## 2.2 Object Nets

For the rest of the paper we fix a fixed logic structure $\Gamma = (\Sigma, X, E, \Psi)$ and the global set of places $\mathbf{P} := \bigcup_{k\in K} \mathbf{P}_k$, transitions $\mathbf{T} := \bigcup_{k\in K} \mathbf{T}_k$, and channels $\mathbf{C} := \bigcup_{k\in K} \mathbf{C}_k$ where all the $\mathbf{P}_k$, $\mathbf{T}_k$, and $\mathbf{C}_k$ are countable and pairwise disjoint. The intended meaning is that a place $p \in \mathbf{P}_k$ contains only net-tokens of object nets that are of type $k$.

This implicit typing is also expressed by the mapping $cd : \mathbf{P} \cup \mathbf{C} \to K$ with the meaning that the net-tokens on $p$ have the structure of nets in the set $\mathcal{N}_{cd(p)}$. Similarly for channels. The mapping is deduced from the the families $\mathbf{P}_k$ and $\mathbf{C}_k$ as:

$$cd(x) = k \quad :\Longleftrightarrow \quad x \in \mathbf{P}_k \cup \mathbf{C}_k \tag{1}$$

In the graphical representation the typing is denoted as a label $::cd(p)$ of the place $p$. This label is omitted for $cd(p) = \bullet$.

Note that recursive nesting is possible, i.e. it is allowed that for some $p \in P(N)$ we have $cd(p) = N$.

An object net $N$ is the common structure for all the net-tokens of $N$. Object nets are Petri nets with finite subsets $P \subseteq \mathbf{P}$, $T \subseteq \mathbf{T}$, and $C \subseteq \mathbf{C}$. (This already implies $P \cap T = \emptyset$.) Each arc is labelled with a term of the signature (which means that the net-token used in the firing phase of the transition has the net structure that is obtained by evaluating this term).[1] This arc labelling is defined by the mappings $\partial_P^-, \partial_P^+ : T \to (P \to \mathbb{T}_\Sigma(X))$ where $\partial_P^\pm$ is a total mapping, while $\partial_P^\pm(t)$ is only defined for those $p$ that are connected with $t$.

Channels are modelled very similarly to normal places. The main difference is that channels are unmarked in 'real' states. Tokens on channel places denote some intermediate state occurring only during a synchronisation very similar to the mechanism of zero-safe nets [27].

Channels are used to transfer tokens between the nesting levels of the marking. Channels connecting to the level above are called *up-links* and those connecting to the levels below are called *down-links*. Due to the tree-like nesting structure of markings there may be several down-links but at most one up-link: $|C^-(t)| \leq 1$. The labelling for channels is described by the mappings $\partial_C^-, \partial_C^+ : T \to (C \to \mathbb{T}_\Sigma(X))$ where $\partial_C^-$ describes the *up-links* and $\partial_C^+$ the *down-links*. In the graphical representation an up-link $c$ is denoted as the transition label $?c$ and a down-link $c$ as $!c$.

Since places $\mathbf{P}$ and channels $\mathbf{C}$ are disjoint, we combine the mappings $\partial_P^-$ and $\partial_C^-$ into one single mapping (analogously for $\partial^+$):

$$\partial^-, \partial^+ : T \to ((P \cup C) \to \mathbb{T}_\Sigma(X))$$

The set of all *up-links* $C^-(t)$ and the set of all *down-links* $C^+(t)$ of a transition $t$ are those channels $c \in C$ such that $\partial_C^\pm(t)(c)$ is defined:

$$C^\pm(t) := \mathrm{dom}(\partial_C^\pm(t)) \tag{2}$$

For uniformity of notations we use $P^-(t) := \mathrm{dom}(\partial_P^-(t))$ instead of ${}^\bullet t$ and $P^+(t) := \mathrm{dom}(\partial_P^+(t))$ instead of $t^\bullet$ to denote pre- and postsets.

**Definition 1.** *An* object net *is a tuple:*

$$N = (P, T, C, \partial^-, \partial^+, G)$$

1. *$P \subseteq \mathbf{P}$ is a finite set of places.*
2. *$T \subseteq \mathbf{T}$ is a finite set of transitions.*
3. *$C \subseteq \mathbf{C}$ is a finite set of channels.*
4. *$\partial^\pm : T \to ((P \cup C) \to \mathbb{T}_\Sigma(X))$ are the pre- and post-conditions allowing at most one up-link: $|C^-(t)| \leq 1$ for all $t \in T$.*
5. *$G : T \to PL_\Gamma$ is the guard predicate.*

Given the object net $N$ then $P(N)$ denotes its set of places. Analogously for $T(N)$, $\partial^\pm(N)$ etc.

[1] Therefore, a transition removes at most one token from each place. This assumption allows us to identify each token with the place it has been removed from. Note, that this restriction can easily be dropped for the price of more cumbersome notations.

## 2.3 Hornets: Net-Algebra and -Models

Let $\Sigma$ be a signature over $K$. A *net-algebra* assigns to each type $k \in K$ a set $\mathcal{N}_k$ of object nets.[2] We assume the family $\mathcal{N} = (\mathcal{N}_k)_{k \in K}$ to be disjoint. We identify $\mathcal{N}$ with $\bigcup_{k \in K} \mathcal{N}_k$ in the following. The nodes of the object nets do not have to be disjoint. The firing rule allows to transfer tokens between those places that are shared by object nets (cf. below).

The type $\bullet \in K$ of anonymous tokens is interpreted with $\mathcal{N}_\bullet = \{N_\bullet\}$, where $N_\bullet$ is a net with empty sets of places and transitions.

The family of object nets $\mathcal{N}$ is the universe of the algebra. A net-algebra $(\mathcal{N}, \mathcal{I})$ assigns to each constant $\sigma \in \Sigma_{\lambda,k}$ an object net $\sigma^{\mathcal{I}} \in \mathcal{N}_k$ and to each operator $\sigma \in \Sigma_{k_1 \cdots k_n,k}$ with $n > 0$ a mapping $\sigma^{\mathcal{I}} : \mathcal{N}_{k_1} \times \cdots \times \mathcal{N}^n_{k_n} \to \mathcal{N}_k$.

A variable assignment $\alpha = (\alpha_k : X_k \to \mathcal{N}_k)_{k \in K}$ maps each variable onto an element of the algebra. For a variable assignment $\alpha$ the evaluation of a term $t \in \mathbb{T}^k_\Sigma(X)$ is uniquely defined and will be denoted as $\alpha(t)$.

An axiom $\forall X : (t_1 \sim t_2)$ is valid in the net-algebra $(\mathcal{N}, \mathcal{I})$ under the assignment $\alpha$ (denoted $\mathcal{I}, \alpha \models e$) iff $\alpha(t_1) = \alpha(t_2)$ holds. An axiom $e$ is valid (denoted as $\mathcal{I} \models e$) iff $e$ is valid for all assignments $\alpha$. A set of axioms $E$ is valid (denoted as $\mathcal{I} \models E$) iff each axiom $e \in E$ is valid. A net-algebra, such that all axioms of $(\Sigma, X, E)$ are valid, is called *net-theory*.

A net-theory $(\mathcal{N}, \mathcal{I})$ extends to a *net-model* of $\Gamma = (\Sigma, X, E, \Psi)$, if we extend the interpretation $\mathcal{I}$ on predicates: Each predicate $\psi \in \Psi_{k_1 \cdots k_n}$ is interpreted by $n$-ary relation $\psi^{\mathcal{I}} \subseteq \mathcal{N}_{k_1} \times \cdots \times \mathcal{N}_{k_n}$.

For a fixed net-model each variable assignment $\alpha : X \to \mathcal{N}$ induces the truth evaluation for each formulae $F \in PL_\Sigma$ in the usual way for predicate logic. This truth value is denoted by $F^{\mathcal{I},\alpha}$.

Deducibility of a formulae $F$ from a set of formulae $\mathcal{F}$ is denoted by $\mathcal{F} \models^\alpha_{\mathcal{I}} F$.

*Nested Markings.* We assume a given net-algebra with universe $\mathcal{N}$. Net-tokens of the object net $N$ differ in their markings $\mu$, which are nested multisets.

A *multiset* $\boldsymbol{m}$ on the set $D$ is a mapping $\boldsymbol{m} : D \to \mathbb{N}$. The set of all multisets over the set $D$ is denoted $MS(D)$. We use common notations for multisets, like cardinality $|\boldsymbol{m}|$ or the sum $(\boldsymbol{m}_1 \oplus \boldsymbol{m}_2)$. The empty multiset is denoted by $\mathbf{0}$.

Multisets are the free commutative monoid over $D$ since every multiset has the unique representation in the form $\boldsymbol{m} = \bigoplus_{d \in D} \boldsymbol{m}(d) \cdot d$ where $\boldsymbol{m}(d)$ denotes the multiplicity of $d$.

A net-token is denoted in the form $[N, \mu]$, where $\mu$ is a multiset of net-tokens which must be consistent with the typing $cd$ (cf. below). A *net-token* on the place $p$ is denoted $p[N, \mu]$, where $N$ must be in $\mathcal{N}_k$ for $k = cd(p)$. Similarly for net-tokens on channels.

We define markings as nested multisets. Let $\mathcal{P} := \bigcup_{N \in \mathcal{N}} \mathcal{P}(N)$, $\mathcal{P}_n(N) := \bigcup_{N \in \mathcal{N}} \mathcal{P}_n(N)$, and $\mathcal{P}(N) := \bigcup_{n=0}^{\infty} \mathcal{P}_n(N)$, where:

$$\mathcal{P}_n(N) = \left\{ p[N', \mu'] \mid p \in P(N) \land N' \in \mathcal{N}_{cd(p)} \land \mu' \in MS\big(\bigcup\nolimits_{i<n} \mathcal{P}_i(N')\big) \right\} \quad (3)$$

[2] To be more precise and to avoid cyclic definitions: $\mathcal{N}_k$ are *names* of nets. Since there is no danger of confusion we use $N \in \mathcal{N}_k$ as the name and for the net $N$ itself.

Black tokens on $p$ are of the form $p[N_\bullet, \mathbf{0}]$, since whenever $cd(p) = \bullet$ then the marking must be empty, as $N_\bullet$ has no places. The token $p[N_\bullet, \mathbf{0}]$ is abbreviated as $p[]$.

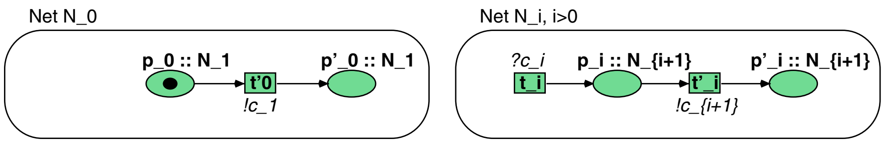


**Fig. 4.** Hornet with unbounded marking depth

There is no a-priori bound for the nesting of markings $\mu \in MS(\mathcal{P})$. E.g. we may have $K = \mathbb{N} \cup \{\bullet\}$ as the set of types and $\mathcal{N}_i = \{N_i\}$ for all $i \in \mathbb{N}$. The structure of the $N_i$ is given in Fig. 4. Here we have $cd(c_i) = \bullet$ and $cd(p_i) = cd(p'_i) = N_{i+1}$. Whenever $p_i$ is marked, then the transition $t'_i$ synchronises via the channel $c_{i+1}$ with $t_{i+1}$ and generates a net-token of type $i+2$ on $p_{i+1}$. Each firing step increases the number of used net-types as well as the nesting level by one:

$$p_0[N_1, \mathbf{0}] \to p'_0[N_1, p_1[N_2, \mathbf{0}]] \to p'_0[N_1, p'_1[N_2, p_2[N_3, \mathbf{0}]]] \to \cdots$$

*Hornets* Let $\Gamma = (\Sigma, X, E, \Psi)$ be a logic-structure over the types $k \in K$. A *Hornet* consists of a family of object nets $\mathcal{N} = (\mathcal{N}_k)_{k \in K}$ where $(\mathcal{N}, \mathcal{I})$ has to be a net-model for the given $\Gamma$.

All the arc expressions must be consistent with the typing $cd(p)$ of places:

$$\forall N \in \mathcal{N} : \forall t \in T(N) : \forall x \in (P^\pm(t) \cup C^\pm(t)) : \partial^\pm(t)(x) \in \mathbb{T}_\Sigma^{cd(x)}(X) \qquad (4)$$

Additionally, we distinguish the top-level net of the system $N_s \in \mathcal{N}$, called the system-net, and its initial marking $\mu_0 \in MS(\mathcal{P}(N_s))$.

**Definition 2.** *A* Higher Order Reflexive Net *(short:* Hornet*) is a tuple:*

$$OS = (\mathcal{N}, \mathcal{I}, N_s, \mu_0)$$

1. *$(\mathcal{N}, \mathcal{I})$ is a net-model for $\Gamma$, respecting (4).*
2. *$N_s \in \mathcal{N}$ is the system-net.*
3. *$\mu_0 \in MS(\mathcal{P}(N_s))$ is the initial marking of $N_s$.*

A Hornet is called *elementary typed*, whenever each type denotes one net: $\forall k \in K : |\mathcal{N}_k| = 1$. We can enforce this property by restricting the operators for each $k \in K$ to exactly one constant $c_k$: $|\Sigma_{\epsilon,k}| = 1$ and $|\Sigma_{w,k}| = 0$ for all $|w| > 0$. In this case we can assume that in each algebra the set $\mathcal{N}_k$ consists of exactly one object net, i.e. the interpretation of the constant $c_k$. In this case we have a one-to-one correspondence between types in $K$ and nets in $\mathcal{N}$. For elementary typed Hornets we may omit the arc labels in the graphical representation, since they can be derived from the typing $cd$. Similarly, the net $N$ in markings $[N, \mu]$ may be omitted.

An Hornet is called *elementary communicating*, when channels transport black tokens only: $\forall c \in \mathbf{C} : cd(c) = \bullet$. Note, that the systems considered in [4] are restricted to elementary typed and communicating ones.

## 2.4 Bindings, Events, and the Firing Rule

*Bindings* An *assignment* $a = (a_k : X_k \to \mathcal{N}_k)_{k \in K}$ maps each net variable $x \in X_k$ to its value $a_k(x)$ in the algebra. An assignment extends the usual way to terms. For the arc expressions $\partial_P^{\pm}(t)(p)$ the assignment evaluates to a concrete object net: $N_p^{\pm} := a(\partial_P^{\pm}(t)(p))$. Analogously for channels $c$.

Let $t \in T(N)$ be a transition of the object net $N$. Each firing activity modifies the markings of the involved net-tokens. This is modelled by the set $X_\mu(t)$ of firing variables:

$$X_\mu(t) := \{x_{p_1}^-, x_{p_2}^+, y_u^{\pm}, z_v^{\pm}, x_{N_1,N_2}^{\pm}, x_v^{\pm} \mid p_1 \in P^-(t), p_2 \in P^+(t), \\ u \in C^-(t), v \in C^+(t), N_1, N_2 \in \mathcal{N}\} \tag{5}$$

The variables in $\bigcup_{t \in T} X_\mu(t)$ are chosen disjoint to those in $\bigcup_{k \in K} X_k$. The pair $b = (a, a_\mu)$ is called a *binding* of $t$, whenever $a$ is an assignment that satisfies the guard $G(t)$, i.e. $E \models_{\mathcal{I}}^a G(t)$ holds. The set of all bindings of $t$ is $\mathcal{B}(t)$.

The firing rule defines for each net type $N$ a separate constraint. Therefore, we identify the set of places and channels that are connected to $t$ in a given binding $b$ via an arc, whose expression evaluates to $N$.

$$P_N^{\pm}(t, b) = \{p \in P^{\pm}(t) \mid a(\partial_P^{\pm}(t)(p)) = N\} \tag{6}$$
$$C_N^{\pm}(t, b) = \{c \in C^{\pm}(t) \mid a(\partial_C^{\pm}(t)(c)) = N\} \tag{7}$$

*Events.* Events are a 'tree-like bundle' of transition instances synchronised via channels. Like markings events are nested, too and form *synchronisation trees.* For each down-link $c \in C^+(t)$ there must be a synchronisation tree $\tau(c)$ that has to fire synchronously with $t$. For $\theta = (t, b)[\tau]$ we define $P^{\pm}(\theta) := P^{\pm}(t)$ and $C^{\pm}(\theta) := C^{\pm}(t)$.

The set of synchronisation trees is defined by $\mathcal{T} := \bigcup_{N \in \mathcal{N}} \mathcal{T}(N)$ where $\mathcal{T}(N) := \bigcup_{n=0}^{\infty} \mathcal{T}_n(N)$, $\mathcal{T}_n := \bigcup_{N \in \mathcal{N}} \mathcal{T}_n(N)$, and:

$$\mathcal{T}_n(N) := \Big\{ (t, b)[\tau] \mid t \in T(N), b \in \mathcal{B}(t) \wedge \tau : C^+(t) \to \big(\textstyle\bigcup_{k<n} \mathcal{T}_k\big) \\ \wedge \forall c \in C^+(t) : C^-(\tau(c)) = \{c\} \Big\} \tag{8}$$

A synchronisation tree $(t, b)[\tau] \in \mathcal{T}$ not having an up-link is closed and is called *event.* The set of all events is:

$$\Theta := \{(t, b)[\tau] \in \mathcal{T} \mid C^-(t) = \emptyset\} \tag{9}$$

*Firing Rule.* The firing rule synchronises all transitions occurring in an event $\theta = (t,b)[\tau]$. We define the auxiliary relation $\cdot \overset{\cdot}{\rightrightarrows} \cdot$ recursively over the structure of a given synchronisation tree. The firing rule is then obtained from this relation as a restriction to events.

For a given $\theta = (t,b)[\tau], b = (a, a_\mu)$ we define the following abbreviations for the firing variables in $X_\mu(t)$:

$$\mu_p^\pm := a_\mu(x_p^\pm), \quad \mu_u^\pm := a_\mu(y_u^\pm), \quad \mu_v^\pm := a_\mu(z_v^\pm),$$
$$\alpha_{N_1,N_2}^\pm := a_\mu(x_{N_1,N_2}^\pm), \text{ and } \beta_v^\pm := a_\mu(x_v^\pm)$$

For each $N \in \mathcal{N}$ the tree $\theta = (t,b)[\tau]$ removes net-tokens $[N,\mu]$ of the structure $N$ from all the places $p \in P_N^-(t)$ in the preset and from all up-links $u \in C_N^-(t)$ (there is at most one up-link). Analogously firing generates net-tokens in the postset, i.e. on all $p \in P_N^+(t)$. During the firing the event synchronises with the sub-synchronisation trees $\tau(v)$ for each down-link $v \in C^+(t)$.

Each channel allows a bi-directional flow of tokens, i.e. there is a marking $\mu_u^-$ provided by the channel and consumed by the event and another marking $\mu_u^+$ generated during the firing which is sent back over the up-link $u$.

The concrete constraints on the combination and distribution of the markings is restricted by the predicates $\psi_1^\pm(\theta)$, $\psi_2(\theta)$ and $\psi_3(\theta)$ – cf. (10), (11) and (12) below.

**Definition 3.** *Let $\theta = (t,b)[\tau] \in \mathcal{T}$ be a synchronisation-tree of the Hornet OS. We define:*

$$\mu^\pm(\theta) := \bigoplus_{N \in \mathcal{N}} \bigoplus_{u \in C_N^-(t,b)} u[N, \mu_u^\pm] \oplus \bigoplus_{p \in P_N^\pm(t,b)} p[N, \mu_p^\pm]$$

*Then $\mu^-(\theta) \overset{\theta}{\rightrightarrows} \mu^+(\theta)$ holds iff $\psi(\theta) := \psi_1^-(\theta) \wedge \psi_1^+(\theta) \wedge \psi_2(\theta) \wedge \psi_3(\theta)$ holds.*

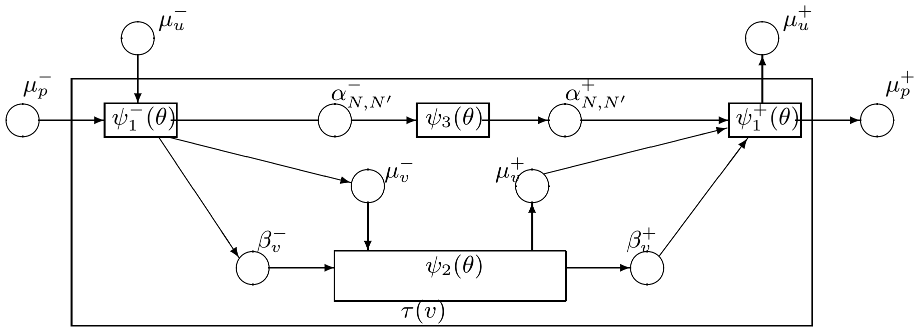


**Fig. 5.** The flow of tokens for nested events

In the following we define the conditions $\psi_1^\pm(\theta)$, $\psi_2(\theta)$ and $\psi_3(\theta)$ for $\theta = (t,b)[\tau]$. These conditions are illustrated in Figure 5 as a flow of net-tokens.

- For each $N$ the incoming markings $\mu_u^-$ and $\mu_p^-$ are added and distributed onto the markings $\alpha_{N,N'}^-$, $\beta_v^-$, and $\mu_v^-$. In a symmetric way, the markings $\alpha_{N,N'}^+$, $\beta_v^+$, and $\mu_v^+$ are added and distributed to the outgoing markings $\mu_u^+$ and $\mu_p^+$. The summation ranges over variables of the same net type $N$. The condition $\psi_1^\pm(\theta)$ holds iff:

$$\forall N \in \mathcal{N} : \bigoplus_{u \in C_N^-(t)} \mu_u^\pm \oplus \bigoplus_{p \in P_N^\pm(t)} \mu_p^\pm = \bigoplus_{N' \in \mathcal{N}} \alpha_{N,N'}^\pm \oplus \bigoplus_{v \in C_N^+(t)} \mu_v^\pm \oplus \bigoplus_{\substack{v \in C^+(t),\\ \tau(v) \in \mathcal{T}(N)}} \beta_v^\pm \tag{10}$$

Note that each channel, i.e. $u \in C_N^-(t)$ or $v \in C_N^+(t)$, is present in both equations, since channels transport net-tokens in both directions.
- For each down-link $v \in C^+(t)$ the marking $\beta_v^-$ enables the sub-synchronisation $\tau(v)$. Over the channel $v$ the net-token $[N_v, \mu_v^-]$ is moved downwards. On firing $\tau(v)$ generates the marking $\beta_v^+$ and from the channel $v$ the net-token $[N_v, \mu_v^+]$ is received:

$$\psi_2(\theta) \iff \forall v \in C^+(t) : \left(\beta_v^- \oplus v\big[N_v, \mu_v^-\big]\right) \overset{\tau(v)}{\rightrightarrows} \left(\beta_v^+ \oplus v\big[N_v, \mu_v^+\big]\right) \tag{11}$$
$$\text{with } N_v = a(\partial_C^+(t)(v))$$

- Whenever a place $p$ is a place of $N_1$ as well as one of $N_2$, it is possible to transfer tokens from $p$ in $N_1$ to this $p$ in $N_2$. The multiset $\alpha_{N_1,N_2}^- \in MS(\mathcal{P}(N_1))$ contains those tokens, that $N_1$ provides $N_2$ with for a transfer. Similarly, $\alpha_{N_2,N_1}^+ \in MS(\mathcal{P}(N_2))$ contains those tokens, that $N_2$ receives from $N_1$. Both must be equal (which implies that they are multisets over common places):

$$\psi_3(\theta) \iff \forall N_1, N_2 \in \mathcal{N} : \quad \alpha_{N_1,N_2}^- = \alpha_{N_2,N_1}^+ \tag{12}$$

An event $\theta$ is called *transfer-free* iff no token transfer between nets – except the trivial one – takes place:

$$\forall N_1, N_2 : (\alpha_{N_1,N_2}^- \neq \mathbf{0} \vee \alpha_{N_2,N_1}^+ \neq \mathbf{0}) \Longrightarrow N_1 = N_2$$

In this case we have $\alpha_{N,N}^- = \alpha_{N,N}^+$ which will be abbreviated as $\alpha_N$.

The constraint $\psi(\theta)$ guarantees that markings are only distributed when not specified differently by the sub-synchronisations. The constraint also guarantees that the typing of tokens remains correct.

The firing rule of Hornets is restricted to those synchronisation trees $\mu \overset{\theta}{\rightrightarrows} \mu'$ where $\theta \in \mathcal{T}$ such that we have a closed synchronisation, i.e. to events: $\theta \in \Theta$. In addition we have identities $\mu \xrightarrow{id_\mu} \mu$ and the firing rule is closed with respect to addition and context formation. This closure extends the nested multiset structure of markings onto the set of actions. It is generated very similarly to the "Petri nets are monoids" approach of [28].

**Definition 4.** *The firing rule* $\xrightarrow{\theta}$ *is the smallest relation closed with respect to the rules in Table 1.*

**Table 1.** Deduction rules for the firing rule

| | | |
|---|---|---|
| generator | $\dfrac{\mu \overset{\theta}{\rightrightarrows} \mu'}{\mu \xrightarrow{\theta} \mu'}$ | $\theta \in \Theta$ |
| identities | $\dfrac{}{\mu \xrightarrow{id_\mu} \mu}$ | $\mu \in MS(\mathcal{P})$ |
| multiset addition | $\dfrac{\mu_1 \xrightarrow{\theta_1} \mu_1' \qquad \mu_2 \xrightarrow{\theta_2} \mu_2'}{(\mu_1 \oplus \mu_2) \xrightarrow{(\theta_1 \oplus \theta_2)} (\mu_1' \oplus \mu_2')}$ | |
| sequence | $\dfrac{\mu \xrightarrow{\theta_1} \mu' \qquad \mu' \xrightarrow{\theta_2} \mu''}{\mu \xrightarrow{(\theta_1 \cdot \theta_2)} \mu''}$ | |
| context | $\dfrac{\mu \xrightarrow{\theta} \mu'}{p[N,\mu] \xrightarrow{p[N,\theta]} p[N,\mu']}$ | $\mu,\mu' \in MS(\mathcal{P}(N))$, $p \in P, N \in \mathcal{N}_{cd(p)}$ |

*Example: Events without Sub-Synchronisations* The firing rule given in Definition 3 simplifies a lot whenever an event has no sub-synchronisations, i.e. for all events of the form $\theta = (t,b)[\emptyset]$. Since $C_N^+(t,b)$ is empty, the constraint $\psi_2(\theta)$ is always true and then we obtain:

$$\frac{\forall N \in \mathcal{N} : \bigoplus_{p \in P_N^{\pm}(t,b)} \mu_p^{\pm} = \bigoplus_{N' \in \mathcal{N}} \alpha_{N,N'}^{\pm} \quad \wedge \quad \forall N_1, N_2 \in \mathcal{N} : \quad \alpha_{N_1,N_2}^{-} = \alpha_{N_1,N_2}^{+}}{\bigoplus_{N \in \mathcal{N}} \bigoplus_{p \in P_N^{-}(t,b)} p[N, \mu_p^{-}] \overset{(t,b)[\emptyset]}{\rightrightarrows} \bigoplus_{N \in \mathcal{N}} \bigoplus_{p \in P_N^{+}(t,b)} p[N, \mu_p^{+}]}$$

*Example 2.* Consider the Hornet of Figure 3 in the marking $\mu = p[N_1, \mu_p] \oplus q[N_2, \mu_q]$ where $\mu_p = v[]$ and $\mu_q = s[]$. Since the places of $N_1$ are disjoint with those in $N_2$ and both are subsets of places in $N_3$, firing may transfer tokens.

The event $(t,b)[\emptyset]$ with $a(x) = N_1$, $a(y) = N_2$ and $a(x\|y) = N_1 \|^{\mathcal{I}} N_2 = N_3$ is enabled:

$$p[N_1, \mu_p] + q[N_2, \mu_q] \xrightarrow{(t,b)[\emptyset]} r[N_3, \mu_r)]$$

The firing rule determines $\mu_r$, since it holds:

$$\begin{aligned} \mu_p^- &= \alpha_{N_1,N_1}^- \oplus \alpha_{N_1,N_2}^- \oplus \alpha_{N_1,N_3}^- & \mathbf{0} &= \alpha_{N_1,N_1}^+ \oplus \alpha_{N_1,N_2}^+ \oplus \alpha_{N_1,N_3}^+ \\ \mu_q^- &= \alpha_{N_2,N_1}^- \oplus \alpha_{N_2,N_2}^- \oplus \alpha_{N_2,N_3}^- & \mathbf{0} &= \alpha_{N_2,N_1}^+ \oplus \alpha_{N_2,N_2}^+ \oplus \alpha_{N_2,N_3}^+ \\ \mathbf{0} &= \alpha_{N_3,N_1}^- \oplus \alpha_{N_3,N_2}^- \oplus \alpha_{N_3,N_3}^- & \mu_r^+ &= \alpha_{N_3,N_1}^+ \oplus \alpha_{N_3,N_2}^+ \oplus \alpha_{N_3,N_3}^+ \end{aligned}$$

Since $N_1$ and $N_2$ are disjoint, we have $\alpha_{N_1,N_2}^- = \mathbf{0}$ and $\alpha_{N_2,N_1}^- = \mathbf{0}$. Simplifying further, we obtain:

$$\mu_p^- = \alpha_{N_1,N_3}^-, \quad \mu_q^- = \alpha_{N_2,N_3}^-, \quad \text{and} \quad \mu_r^+ = \alpha_{N_1,N_3}^- \oplus \alpha_{N_2,N_3}^- = \mu_p^- \oplus \mu_q^-$$

This generates the expected event:

$$\mu = p[N_1, \mu_p] \oplus q[N_2, \mu_q] \xrightarrow{(t,b)[\emptyset]} r[N_3, (\mu_p \oplus \mu_q)]$$

The resulting marking of the net-tokens on place $r$ equals the sum of the markings of the two net-tokens on $p$ and $q$.

Whenever all object nets are disjoint we have $\alpha^{\pm}_{N_1,N_2} \neq \mathbf{0}$ only if $N_1 = N_2$. In this case we can simplify the rule further:

$$\frac{\forall N \in \mathcal{N} : \quad \bigoplus_{p \in P^-_N(t,b)} \mu^-_p = \bigoplus_{p \in P^+_N(t,b)} \mu^+_p}{\bigoplus_{N \in \mathcal{N}} \bigoplus_{p \in P^-_N(t,b)} p[N, \mu^-_p] \overset{(t,b)[\emptyset]}{\Rightarrow} \bigoplus_{N \in \mathcal{N}} \bigoplus_{p \in P^+_N(t,b)} p[N, \mu^+_p]}$$

The constraint can be read as follows: An event is enabled whenever the sum of the markings of all the removed net-tokens of type $N$ equals the sum of the markings of all the generated ones.

Since a transfer of tokens is only possible between the same place in different nets, we obtain the following sufficient condition.

**Lemma 1.** *Whenever the places of all nets in $\mathcal{N}$ are disjoint, then all events $\theta \in \Theta$ are transfer-free.*

Since we are free to name the elements of object nets, we assume that object nets share places whenever a conversion is explicitly wanted.

*Example: Transfer of Net-Tokens over Channels* In Hornets net-tokens can be transferred over channels. We discuss the event $\theta = (t, b_t)[\tau]$ with $\tau : c \mapsto (v, b_v)[]$ for the example in Figure 6. The current marking is $\mu = p_1[N_1, \mathbf{0}] \oplus p_2[N_2, q_1[]]$.

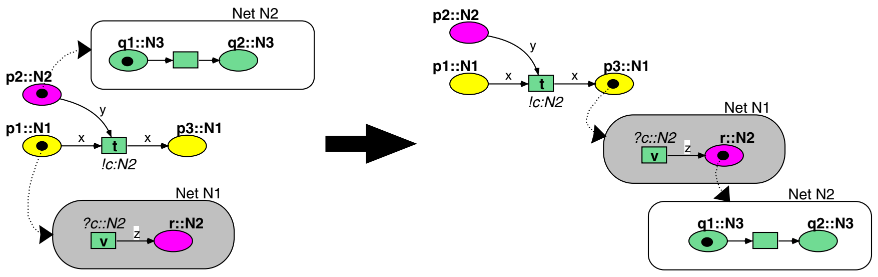


**Fig. 6.** Transfer of Net-Tokens over Channels

Since all the nets are disjoint, all events are transfer-free and only the multisets $\alpha^{\pm}_{N,N}$ are nonempty and $\alpha^-_{N,N} = \alpha^+_{N,N}$ due to $\psi_3$.

– The sub-synchronisation event $(v, b_v)[]$ where $b_v(z) = N_2$ is given as:

$$c[N_2, \mu^-_c] \overset{(v,b_v)[]}{\Rightarrow} c[N_2, \mu^+_c] \oplus r[N_2, \mu^+_r]$$

where $\psi^{\pm}_1$ is

$$\mu^-_c = \alpha_{N_2} = \mu^+_c \oplus \mu^+_r \qquad (*)$$

– For the event $\theta = (t, b_t)[\tau]$ with $b_t(x) = N_1$ and $b_t(y) = N_2$ we have:

$$p_1[N_1, \mu^-_{p_1}] \oplus p_2[N_2, \mu^-_{p_2}] \quad \overset{\theta}{\rightrightarrows} \quad p_3[N_1, \mu^+_{p_3}]$$

To match the actual marking $\mu$ we are interested in bindings $b_t$ where $\mu^-_{p_1} = \mathbf{0}$ and $\mu^-_{p_2} = q_1[]$. Here $\psi_1^\pm$ is given as:

$$\begin{array}{rl} \mu^-_{p_1} &= \alpha_{N_1} \oplus \beta^-_c \\ \mu^+_{p_3} &= \alpha_{N_1} \oplus \beta^+_c \\ \mu^-_{p_2} &= \alpha_{N_2} \oplus \mu^-_c \\ \mathbf{0} &= \alpha_{N_2} \oplus \mu^+_c \end{array} \qquad \begin{array}{l} \text{which simplifies,} \\ \text{using } \ \mu^-_{p_1} = \mathbf{0} \ \text{ and} \\ \mu^-_{p_2} = q_1[], \ \text{further to:} \end{array} \qquad \begin{array}{rl} \mathbf{0} &= \alpha_{N_1} = \beta^-_c \\ \mu^+_{p_3} &= \beta^+_c \\ q_1[] &= \mu^-_c \\ \mathbf{0} &= \alpha_{N_2} = \mu^+_c \end{array}$$

Analogously, $\psi_2$, which is $c[N_2, \mu^-_c] \oplus \beta^-_c \quad \overset{\tau(c)}{\rightrightarrows} \quad c[N_2, \mu^+_c] \oplus \beta^+_c$, simplifies to

$$c[N_2, q_1[]] \quad \overset{\tau(c)}{\rightrightarrows} \quad c[N_2, \mathbf{0}] \oplus \beta^+_c$$

Matching this with the sub-synchronisation $\overset{(v,b_v)[]}{\rightrightarrows}$ above and using $(*)$ we obtain $\mu^+_r = q_1[]$ and $\mu^+_{p_3} = \beta^+_c = r[N_2, q_1[]]$:

$$c[N_2, q_1[]] \quad \overset{(v,b_v)[]}{\rightrightarrows} \quad c[N_2, \mathbf{0}] \oplus r[N_2, q_1[]]$$

All in all we obtain the event $\theta$ as:

$$\mu = p_1[N_1, \mathbf{0}] \oplus p_2[N_2, q_1[]] \quad \overset{\theta}{\rightrightarrows} \quad p_3[N_1, r[N_2, q_1[]]] =: \mu'$$

Note, that the firing increases the nesting level of the marking by one. The resulting marking is shown on the right of Figure 6.

The effect of the transfer is very similar to the in/out primitives of the ambient calculus [14].

## 2.5 Expressiveness

Hornets can simulate counter programs and therefore have the power of Turing machines. This simulation can be given in two different ways: The first variant uses the algebraic structure to define the data type Nat with the constant zero and the unary operator suc. For the second variant we encode a counter value $n$ by an object nets that has a marking of depth $n$ (cf. [4]).

But even if one does not use the algebraic structure and limits the nesting depth, we obtain that the reachability problem is undecidable [29].

## 2.6 Reversibility of the Firing Rule

A basic property of Petri nets is that their firing rule is symmetric in time, i.e. whenever all arcs are reversed then we can fire backwards: This is expressed by the reversed net $N^{rev} = (P, T, \partial^-, \partial^+)$ which is obtained from $N = (P, T, \partial^+, \partial^-)$ by dualising the effect. Symmetry in firing is expressed as:

$$m_1 \xrightarrow[N]{t} m_2 \iff m_2 \xrightarrow[N^{rev}]{t} m_1$$

This property of reversibility holds also for object nets. For each object net $N = (P, T, C, \partial^-, \partial^+, G)$ the reverse object net is $N^{rev} = (P, T, C, \partial^+, \partial^-, G)$.

We have defined the firing rule carefully in such a way that reversibility holds also for object net systems:

**Theorem 1.** *Let OS be a Hornet and $\theta \in \Theta$ an event, then we have the reversibility:*

$$\mu_1 \xrightarrow[OS]{\theta} \mu_2 \iff \mu_2 \xrightarrow[OS^{rev}]{\theta} \mu_1$$

*Proof.* Observe that reverting all arcs is equivalent to the inversion of the flow in Figure 5. Formally this is obtained if we consistently change $\pm$ into $\mp$. An easy induction over the depth of the synchronisation tree shows that the firing constraints are reversible, too: If we revert the flow in Figure 5 then $\psi_1^-(\theta)$ becomes $\psi_1^+(\theta)$ and vice versa. The constraint $\psi_3(\theta)$ is self-dual. For synchronisation trees of depth zero we are done. Whenever we have a tree of depth $n+1$, we use that $\psi_2(\theta)$ and $\psi_3(\theta)$ are reversible by induction assumption and we are done, too.

*Example 3.* Cf. the Hornet in Fig. 3. The reverse of event $(t, b)[\emptyset]$ may fire:

$$r[N_3, (\mu_p \oplus \mu_q)] \xrightarrow[OS^{rev}]{(t,b)[\emptyset]} p[N_1, \mu_p] \oplus q[N_2, \mu_q]$$

Note, that this does not mean that each possible marking $\mu_r$ of the net-token $[N_3, \mu_r]$ on place $r$ may fire in $OS^{rev}$, since this is possible only if we can decompose $\mu_r$ into two markings: $\mu_r = (\mu_p \oplus \mu_q)$. This cannot be done e.g. for $\mu_r = i_3$ since $i_3$ is neither a place in $N_1$ nor in $N_2$.

## 2.7 Relationships to Other Formalisms

Hornets are expressive enough, so that many other formalisms can be embedded into them. The most obvious ones are algebraic nets and object nets.

Algebraic Nets [25] can be seen as Hornets that do not use the feature of nesting and all the net-tokens' markings are empty. Only the system net itself is used and the signature of the coloured tokens are directly used for the object nets. Of course, the algebra $A$ of the algebraic net is not a net-algebra in general, but at least we always have an net algebra that is isomorphic to $A$.

It is quite obvious that Hornets extend the object net systems presented in [4]. We like to point out the main differences:

(a) The object net systems do not use signatures and algebras. In fact there are no arc expressions at all since for each place the structure of all net-tokens is equal. Therefore these object nets can be seen as elementary typed Hornets (i.e. each type denotes only one net).

(b) The second difference is that the channels of object nets are used only for synchronisation; net-tokens are not transferred over them. Therefore they are elementary communicating Hornets.

(c) All the object nets are assumed to be disjoint. By Lemma 1 we have no transfer between net-tokens derived from different object nets. Therefore they are transfer-free Hornets. To summarise: Object net systems are Hornets that are elementary typed, *elementary communicating*, and transfer-free.

We like to mention that the formalism of *elementary object nets* (Eos), which we studied in [29], can be considered as a special object net system where only the system net is allowed to have non-anonymous tokens. Therefore, the marking structure is limited to depth of two. Note, transfers between different nets has been studied for Eos already in [30].

## 3 The Workflow Management System

For the distributed workflow management systems we have two sorts, one for the workflows: WFN and one for the the workflow management system agents: WFMS. Thus $K = \{\mathsf{WFN}, \mathsf{WFMS}\}$.

For WFN we have the binary operators $\cdot$, $+$ and $\|$ in $\Sigma_{\mathsf{WFN}\cdot\mathsf{WFN},\mathsf{WFN}}$. Their axioms (associativity, commutativity etc.) are the same as in the box calculus [31]. We assume a set of names for workflows and for each name $n$ we have the constant $\sigma_n \in \Sigma_{\lambda,\mathsf{WFN}}$.

The kind $k = \mathsf{WFN}$ is interpreted by $\mathcal{N}_{\mathsf{WFN}}$ which is the set of all workflow nets. More precisely, the variant of workflow nets where the in-place $i$ is preceded by a start transition with an up-link channel start() and the out-place $o$ is followed by a stop transition with an up-link channel stop(). With these additional transitions the environment of the workflow (here: the WFMS net) can control the start/stop-activation of each workflow. Each WFN constant $\sigma_n$ is interpreted as one workflow net. The operators $\cdot$, $+$ and $\|$ are interpreted as sequential, alternative, and parallel composition, respectively.

For WFMS we have only one constant $\sigma_{wfms} \in \Sigma_{\lambda,\mathsf{WFMS}}$ and no operators. The kind WFMS is interpreted by $\mathcal{N}_{\mathsf{WFMS}} = \{N_{\mathsf{WFMS}}\}$ where $N_{\mathsf{WFMS}}$ is the net given in Figure 7.

We have the variables $A, A_1, A_2 \in X_{\mathsf{WFMS}}$ for WFMS and $N, N_1, N_2 \in X_{\mathsf{WFN}}$ for WFN.

Additionally we use the kinds pair and boolean (using net kinds here as abstract data types) which are interpreted as pairs and booleans with the usual operators and axioms. We have $b$ as a boolean variable.

The complete WFMS is a tree-structure obtained by nesting net-tokens of the net $N_{\mathsf{WFMS}}$ like in Fig. 2. Each parent WFMS stores its children as tokens on the place subordinate WF agents. The resulting Hornet provides channels towards the environment: inWF$(N)$, secure_in$(N)$, and enactWF$(N)$. The channel inWF$(N)$ is used to store a new workflow net $N$ in the WFMS repository while the channel secure_in$(N)$ is used when $N$ is known already to be well-formed. The channel enactWF$(N)$ is used to activate an new instance of the workflow $N$.

The tasks of each node in the WFMS-tree are carried out by WFMS agents (cf. Fig. 7). Their tasks cover the following aspects:

– Workflow Analysis: ensures well-formed workflows
– Delegation Information Management: responsible for the management of the delegation information

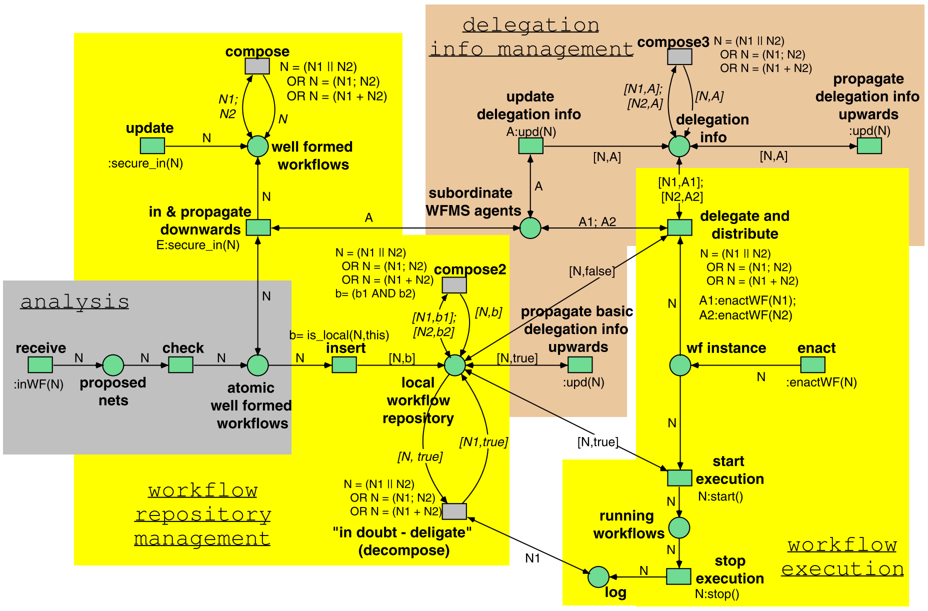


**Fig. 7.** The Workflow Management Agent

- Workflow Repository Management: responsible for the repository of known and executable workflows
- Workflow Execution: responsible for the execution and the distribution of sub-workflows in the WFMS

*Workflow Analysis.* This part receives over the channel inWF$(N)$ a workflow $N$ as a proposed net. This net is checked whether it is a well-formed workflow net, i.e. whether its short-cut is bounded and live. These tests are performed as part of WFMS. Figure 8 shows an appropriate refinement of the transition check. Each workflow net that passes the check successfully is put on the place atomic well formed workflows.

*Workflow Repository Management.* The atomic well formed workflows are handled in two ways: (a) Each well formed workflow $N$ is propagated downwards via the channel secure_in$(N)$ to a subordinate $A$ and stored in the place well formed workflows. The transition compose takes two nets $N_1$ and $N_2$ and stores their composition, i.e. either $(N_1 \cdot N_2)$ or $(N_1 + N_2)$, or $(N_1 \| N_2)$ – which is correct since well-formedness is preserved by composition. (b) Then insert checks whether $N$ is locally executable (computed by the function $b = \text{is\_local}(N)$) and stores the pair $[N, b]$ in the local workflow repository. The transition compose2 stores the composition of $N_1$ and $N_2$ and additionally evaluates whether they are locally executable, which is true if both are.

If $N$ is locally executable, i.e. we have a token $[N, \mathit{true}]$ in the local workflow repository, then this information is propagated upwards along the WMFS-tree

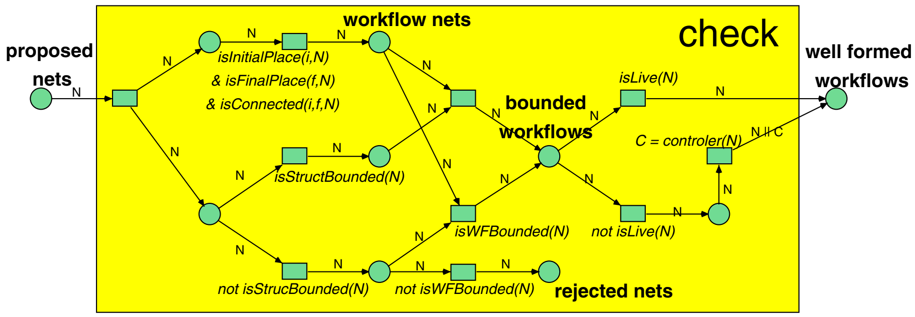


**Fig. 8.** Refinement: Analysis of WF-Nets

via the up-link upd$(N)$ of transition propagate basic delegation info upwards. The corresponding down-link is in the *Delegation Information Management* part.

Transition "in doubt: delegate" is an example for the reorganisation of the local workflow repository. If $[N, b]$ is stored there and $N$ is decomposable into $N_1$ and $N_2$ such that $N_1$ is in the log then $N$ is replaced by $N_1$. (This transition is just for illustrational purposes. Of course the log file has to be analysed in more detail to conclude which reorganisations might be useful.).

*Delegation Information Management.* The transition update delegation info receives from one of its subordinates $A$ the information, that $A$ is capable of enacting $N$ (either directly or indirectly by delegation) over the down-link $A$:upd$(N)$. This fact is stored as a pair $[N, A]$ on delegation info. If $N_1$ is delegatable to $A$ and $N_2$ to $A$, so is the composition of $N_1$ and $N_2$ – which is expressed by compose3. The delegation info $[N, A]$ is also propagated upwards via the up-link upd$(N)$ and eventually reaches the root of the WFMS. This corresponds to the fact that the root is capable of enacting $N$ if some of its descendant is.

*Workflow Execution.* The *Workflow Execution* part is responsible for the workflow enactment. Each WFMS receives the request to enact $N$ via the channel enactWF$(N)$, called either from the environment at the root or from the father at its children. The request is stored on wf instance.

There are two possibilities to enact $N$: either (a) directly via start execution or (b) via transition delegate and distribute. (a) A workflow is locally executable if we have the token $[N, \mathit{true}]$ in the local workflow repository. The transition start execution activates the execution via the channel $N$:start() and puts $N$ on running workflows. When $N$ terminates, the channel $N$:stop() becomes activated and stop execution may fire generating some log information.

(b) For a workflow that is in the local workflow repository but is not locally executable we have the token $[N, \mathit{false}]$. In this case we try to decompose $N$ into $N_1$ and $N_2$ and check whether we know by our delegation info if $A_1$ is capable of enacting $N_1$ (and $A_2$ for $N_2$) and then transition delegate and distribute enacts both sub-workflows via the down-links $A_1$:enact$(N_1)$ and $A_2$:enact$(N_2)$. This distribution continues recursively downwards. Due to the mechanism that manages the delegation info we can be sure that some descendant will finally execute the sub-workflows.

## 4 Conclusion

In this contribution we have introduced an algebraic extension of object nets. These nets are called *Higher Order Recursive Nets*, or short: *Hornets*. Hornets are a combination of algebraic nets and object nets. The algebraic structure introduced here refers to the topology of net-tokens. This is formalised by a signature and many sorted logic that is interpreted by net-algebras.

Compared to object net systems studied in previous works Hornets do not only provide signatures and algebras, but also channels that can transfer net-tokens over the hierarchy; and it is possible that an event can transfer tokens between net-tokens derived from different object nets.

Our main intention for the algebraic extension was to modify the structure of net-tokens at run-time, e.g. by sequential, parallel, or alternative composition (or decomposition) through pattern matching.

If we tailor this construct to the domain of workflow management, our approach reveals an interesting connection to the formalism of *adaptive workflow nets*. Compared to Hornets the origin of adaptive workflow nets has been quite the other way round: It started with WF-nets and extended them with composition and hierarchy, while Hornets started as a *nets within nets* formalism, extended by composition and specialised to workflows.

To demonstrate the elegance of the Hornet formalism we presented a simple Hornet model of a *distributed workflow management system*. This system formalised a network of workflow management agents. These agents cooperatively execute workflow nets (part: workflow execution).

If one agent is not able to execute a workflow itself it decomposes it into simpler parts (exploiting the algebraic net structure) and delegates these parts over the network to other agents. The agents take care that they delegate only if it is guaranteed that their delegation partners are able to execute their parts – either themselves or by further delegation. The necessary data is managed by the *delegation information management* part of the model. The agent network also monitors the execution processes, i.e. the firing of their net-tokens, in order to reorganise the workflow nets structure. This is handled by the *workflow repository management* part. Another specialty is that the analysis whether a workflow net is well-formed can be performed inside the formalism itself, as done in the *workflow analysis* part.

The resulting prototype model of a WFMS is very lean which demonstrates the modelling power of Hornets.

## References

1. Valk, R.: Object Petri nets: Using the nets-within-nets paradigm. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) Advanced Course on Petri Nets 2003. LNCS, vol. 3098, pp. 819–848. Springer, Heidelberg (2004)

2. Aalst, W.v.d., Moldt, D., Valk, R., Wienberg, F.: Enacting interorganizational workflows using nets in nets. In: Working Paper Series of the Department of Information systems: Proceedings of the 1999 Workflow Management Conference, vol. 70, pp. 117–136. University of Münster (1999)
3. Köhler, M., Rölke, H.: Concurrency for mobile object-net systems. Fundamenta Informaticae 54(2-3) (2003)
4. Köhler, M., Rölke, H.: Properties of Object Petri Nets. In: Cortadella, J., Reisig, W. (eds.) ICATPN 2004. LNCS, vol. 3099, pp. 278–297. Springer, Heidelberg (2004)
5. Köhler, M., Rölke, H.: Reference and value semantics are equivalent for ordinary Object Petri Nets. In: Ciardo, G., Darondeau, P. (eds.) ICATPN 2005. LNCS, vol. 3536, pp. 309–328. Springer, Heidelberg (2005)
6. Köhler, M., Farwer, B.: Modelling global and local name spaces for mobile agents using object nets. Fundamenta Informaticae 72(1-3), 109–122 (2006)
7. Köhler, M., Moldt, D., Rölke, H.: Modeling the behaviour of Petri net agents. In: Colom, J.M., Koutny, M. (eds.) ICATPN 2001. LNCS, vol. 2075, pp. 224–241. Springer, Heidelberg (2001)
8. Köhler, M., Moldt, D., Rölke, H.: Modelling mobility and mobile agents using nets within nets. In: van der Aalst, W.M.P., Best, E. (eds.) ICATPN 2003. LNCS, vol. 2679, pp. 121–140. Springer, Heidelberg (2003)
9. Rölke, H., Moldt, D.: Pattern based workflow design using reference nets. In: van der Aalst, W.M.P., ter Hofstede, A.H.M., Weske, M. (eds.) BPM 2003. LNCS, vol. 2678, pp. 246–260. Springer, Heidelberg (2003)
10. Lomazova, I.A., van Hee, K.M., Oanea, O., Serebrenik, A., Sidorova, N., Voorhoeve, M.: Nested nets for adaptive systems. In: Donatelli, S., Thiagarajan, P.S. (eds.) ICATPN 2006. LNCS, vol. 4024, pp. 241–260. Springer, Heidelberg (2006)
11. van Hee, K., Oanea, O., Serebrenik, A., Sidorova, N., Voorhoeve, M., Lomazova, I.: Checking properties of adaptive workflow nets. Fundamenta Informaticae 79(3-4), 347–362 (2007)
12. Aalst, W.v.d.: Verification of workflow nets. In: Azeme, P., Balbo, G. (eds.) ICATPN 1997. LNCS, vol. 1248, pp. 407–426. Springer, Heidelberg (1997)
13. Valk, R.: Petri nets as token objects: An introduction to elementary object nets. In: Desel, J., Silva, M. (eds.) ICATPN 1998. LNCS, vol. 1420, pp. 1–25. Springer, Heidelberg (1998)
14. Cardelli, L., Gordon, A.D., Ghelli, G.: Mobility types for mobile ambients. In: Wiedermann, J., Van Emde Boas, P., Nielsen, M. (eds.) ICALP 1999. LNCS, vol. 1644, pp. 230–239. Springer, Heidelberg (1999)
15. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, parts 1-2. Information and computation 100(1), 1–77 (1992)
16. Busi, N.: Mobile nets. In: Ciancarini, P., Fantechi, A., Gorrieri, R. (eds.) Formal Methods for Open Object-Based Distributed Systems, vol. 139, pp. 51–66. Kluwer, Dordrecht (1999)
17. Haddad, S., Poitrenaud, D.: Theoretical aspects of recursive Petri nets. In: Donatelli, S., Kleijn, J. (eds.) ICATPN 1999. LNCS, vol. 1639, pp. 228–247. Springer, Heidelberg (1999)
18. Lomazova, I.A.: Nested Petri nets – a formalism for specification of multi-agent distributed systems. Fundamenta Informaticae 43(1-4), 195–214 (2000)
19. Xu, D., Deng, Y.: Modeling mobile agent systems with high level Petri nets. In: IEEE International Conference on Systems, Man, and Cybernetics 2000 (2000)
20. Hiraishi, K.: $PN^2$: An elementary model for design and analysis of multi-agent systems. In: Arbab, F., Talcott, C.L. (eds.) COORDINATION 2002. LNCS, vol. 2315, pp. 220–235. Springer, Heidelberg (2002)

21. Bednarczyk, M.A., Bernardinello, L., Pawlowski, W., Pomello, L.: Modelling mobility with Petri hypernets. In: Fiadeiro, J.L., Mosses, P.D., Orejas, F. (eds.) WADT 2004. LNCS, vol. 3423, pp. 28–44. Springer, Heidelberg (2005)
22. Lakos, C.: A Petri net view of mobility. In: Wang, F. (ed.) FORTE 2005. LNCS, vol. 3731, pp. 174–188. Springer, Heidelberg (2005)
23. Hoffmann, K., Ehrig, H., Mossakowski, T.: High-level nets with nets and rules as tokens. In: Ciardo, G., Darondeau, P. (eds.) ICATPN 2005. LNCS, vol. 3536, pp. 268–288. Springer, Heidelberg (2005)
24. Velardo, F.R., de Frutos-Escrig, D.: Name creation vs. replication in petri net systems. Fundam. Inform. 88(3), 329–356 (2008)
25. Reisig, W.: Petri nets and algebraic specifications. Theoretical Computer Science 80, 1–34 (1991)
26. Ehrig, H., Mahr, B.: Fundamentals of algebraic Specification. EATCS Monographs on TCS. Springer, Heidelberg (1985)
27. Bruni, R., Montanari, U.: Zero-safe nets: Comparing the collective and individual token approaches. Information and Computation 156(1-2), 46–89 (2000)
28. Meseguer, J., Montanari, U.: Petri nets are monoids. Information and Computation 88(2), 105–155 (1990)
29. Köhler, M.: Reachable markings of object Petri nets. Fundamenta Informaticae 79(3-4), 401–413 (2007)
30. Köhler, M., Farwer, B.: Object nets for mobility. In: Kleijn, J., Yakovlev, A. (eds.) ICATPN 2007. LNCS, vol. 4546, pp. 244–262. Springer, Heidelberg (2007)
31. Best, E., Devillers, R., Koutny, M.: Petri Net Algebra. EATCS Monographs on Theoretical Computer Science Series. Springer, Heidelberg (2001)

# Monotonicity in Service Orchestrations*

Anne Bouillard[1], Sidney Rosario[2], Albert Benveniste[2], and Stefan Haar[3]

[1] ENS Cachan, IRISA, Université Européenne de Bretagne, Bruz France
[2] IRISA/INRIA, Campus de Beaulieu, Rennes France
[3] INRIA Saclay, ENS Cachan, France

**Abstract.** Web Service orchestrations are compositions of different Web Services to form a new service. The services called during the orchestration guarantee a given Quality of Service (QoS) to the orchestrator, usually in the form of contracts. These contracts can then be used by the orchestrator to deduce the contract it can offer to its own clients, by performing contract composition. An implicit monotonicity assumption in contract based QoS management is: "the better the component services perform, the better the orchestration's performance will be".

In some orchestrations, however, monotonicity can be violated, i.e., the performance of the orchestration improves when the performance of a component service degrades. This is highly undesirable since it can render the process of contract composition inconsistent.

In this paper we formally define monotonicity for orchestrations modelled by Colored Occurrence Nets (CO-nets) and we characterize the classes of monotonic orchestrations. Contracts can be formulated as hard, possibly nondeterministic, guarantees, or alternatively as probabilistic guarantees. Our work covers both cases. We show that few orchestrations are indeed monotonic, mostly because of complex interactions between control, data, and timing. We also provide user guidelines to get rid of non-monotonicity when designing orchestrations.

## 1 Introduction

Web Services and their compositions are being widely used to build distributed applications over the web. Web Service orchestrations are compositions of Web Services to form an aggregate, and usually more complex, Web Service. Different formalisms have been proposed for orchestrating Web Services, the most popular amongst these is the Business Process Execution Language (BPEL) [3]. Another such formalism is Orc [7], a small and elegant language equipped with extensive semantics work [6,12]. Various other models have been used either to directly model orchestrations, or as a semantic domain for some formalisms; see for example the Petri Nets based WorkFlow Nets [13].

Though the main focus of the existing models is to capture the functional aspects of service and their compositions, the non-functional - also called Quality

* This work was partially funded by the ANR national research program DOTS (ANR-06-SETI-003), DocFlow (ANR-06-MDCA-005) and the project CREATE ActivDoc.

of Service (QoS) - aspects also need to be considered. The QoS of a service is characterised by different metrics - called QoS parameters - , *e.g.,* latency, availability, throughput, security, etc. QoS management is usually based on the notion of a Service Level Agreement (SLA) or *contract*, which specifies constraints on the QoS parameters of the service. A typical service contract could be : *for 95% of the requests, the response time will be less than 5ms.* The WSLA Standard [5] is one such proposition for specifying QoS through SLAs.

In service orchestrations, contracts are agreements made between the orchestrator and the different services called by the orchestrator (also called *sub-contractors*) which formalise the duties and responsibilities for each of them. The orchestrator can then compose all the contracts with its sub-contractors, to help it propose a contract to its own clients. This process is called *contract composition.* In [10] we introduced the notion of *probabilistic contracts* to formalise the QoS behaviour of services — the work of [10] focused on latency. We showed how these contracts can be composed to get the orchestration's contract. We also showed that there is room for *overbooking* the orchestrator's resources.

Contract based QoS management in orchestrations relies on the implicit assumption that if each of the sub-contractor meets its contract's objectives, then so does the orchestrator. Vice-versa, a sub-contractor breaching its contract can cause the orchestrator to breach the contract with its clients. Thus the whole philosophy behind contracts is that the better the sub-contractors behave, the better the overall orchestration will meet its contract. In fact, the authors themselves have developed their past work [10] based on this credo . . . until they discovered that this implicit assumption could easily be falsified. Why so?

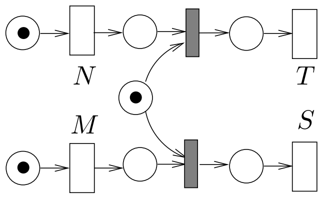


**Fig. 1.** A non-monotonic orchestration

As an example, consider the orchestration modeled by the Petri net in Figure 1. Services $M$ and $N$ are first called in parallel. If $M$ responds first, service $S$ is next called and the response of $N$ is ignored. If $N$ responds first, $T$ is called and not $S$. Let $\delta_i$ denote the response time of site $i$. Assume the following delay behaviour: $\delta_M < \delta_N$ and $\delta_S \gg \delta_T$. Since $M$ responds faster, the end-to-end orchestration delay is $d_0 = \delta_M + \delta_S$. Now let service $M$ behaves slightly 'badly', *i.e* delay $\delta_M$ increases and becomes slightly greater than $\delta_N$. Now service $T$ is called and the new orchestration delay is $d_1 = \delta_N + \delta_T$. But since $\delta_S \gg \delta_T$, $d_1$ is in fact lower than $d_0$. This orchestration is *non-monotonic* since increasing the latency of one of its components can decrease the end-to-end latency of the orchestration. So, what is the nature of the difficulty?

"Simple" composed Web services are such that QoS aspects do not interfere with functional aspects and do not interfere with each other. Their flow of control is typically rigid and does not involve if-then-else branches. For such cases, latencies will compose gently and will not cause pathologies as shown above. However, as evidenced by the rich constructions offered by BPEL, orchestrations and choreographies can have branching based on data and QoS values, various kinds of exceptions, and timers. With such flexibility, non-monotonicity such as that exhibited by the example of Figure 1 can very easily occur.

*Lack of monotonicity impairs using contracts for the compositional management of QoS.* Surprisingly enough, this fact does not seem to have been noticed in the literature.

In this paper we classify orchestrations based on their monotonic characteristics. We focus on latency, although other aspects of QoS are discussed as well. Section 2 informally introduces the notion of monotonicity with examples. In Section 3 we recall the definition of Petri nets and introduce our model, OrchNet. A formal definition of monotonicity and a characterisation of monotonic orchestrations is then given in Section 4. Section 5 extends the notion of monotonicity to nets whose transitions' delays are probability distributions. Section 6 gives a few ideas to avoid the problem of non-monotonicity and Section 7 concludes. Proofs of non-trivial results are deferred to the appendix.

## 2 Examples for Non-monotonic Orchestrations

In this section we look at sample orchestrations and illustrate the concept of (non) monotonicity using them.

*The Travel Planner orchestration:* The orchestration to the left in Figure 2 is inspired by [14]. A client calls the Travel Planner orchestration with a city he

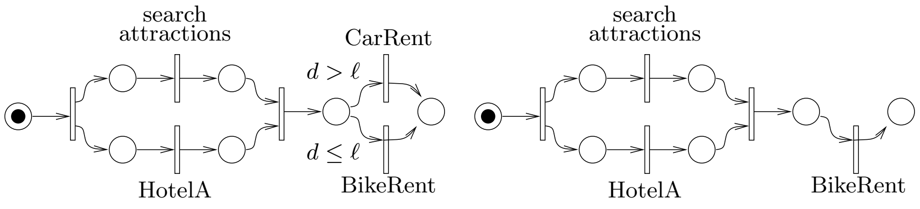


**Fig. 2.** The Travel Planner orchestration (left); a simplified version (right)

plans to visit along with the dates of his visit. The orchestration looks for a hotel in that city (service *HotelA*) for those dates and parallelly looks for sites of attractions (service *Search Attractions*) in the city. Once both these tasks are completed, it calculates the maximal distance '$d$' between the hotel found and the attraction sites. If this distance is less than a certain threshold $\ell$, a bike rental service is called to get quotes for a rental bike. If distance $d$ exceeds $\ell$, then *Car Rent* is called to get quotes for a rental car instead. The orchestration to the right in Figure 2 is a simplified version of travel planner, in which it is assumed that all returns from HotelA are closer than $\ell$ to the attraction site.

This Travel Planner orchestration is monotonic: Increasing (or decreasing) the response time of any of its component services does result in a corresponding increase (or decrease) in the end to end latency. Monotonicity holds in this case because increasing (or decreasing) the response time of the services called first does not affect the value returned by these services.

*The Travel Planner orchestration – A Modified Version.* The presence of timeouts and data dependant choices in orchestrations can however complicate things. Figure 3 (left) is a modified version of the Travel planner example where quotes for hotels are obtained from two services, *HotelA* and *HotelB*. Such an extension is quite natural in orchestrations, where a pool of services with similar functionality are queried with the same request. The orchestration selects the best response obtained from the pool, or combines their responses. In this modified Travel Planner example, of the two hotel offers received, the cheaper one is taken. Calls to the hotels are guarded by timers: if only one hotel has replied before a timeout, the response of the other is ignored. The rest of the example is unchanged.

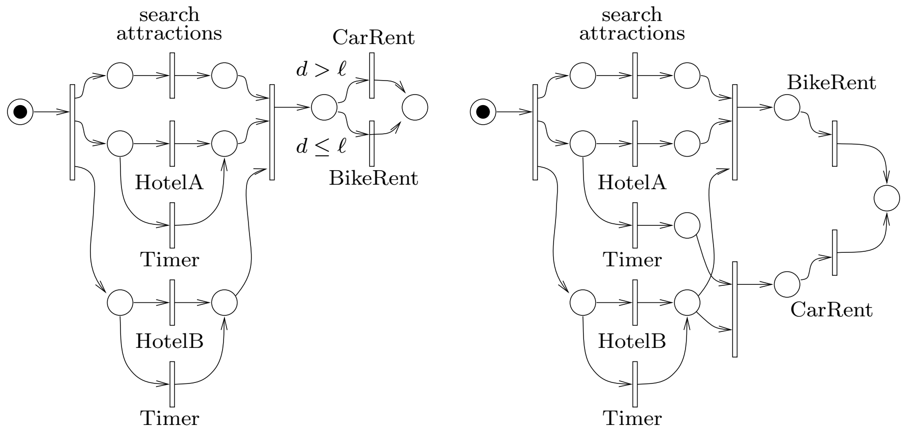


**Fig. 3.** The Modified Travel Planner orchestration. By convention, each *Timer* has priority over the *HotelX* service it is in conflict with. Left (a), right (b).

Now look at the following scenario: *HotelA* returns propositions that are usually cheaper than those of *HotelB* and so *HotelA*'s propositions are chosen. Let the distance $d$ in this case be greater than $\ell$ and so service *Car Rent* is called. If the performance of *HotelA* now degrades such that it doesn't reply before a timeout, only *HotelB*'s response is taken. Say that the maximum distance $d$ in this case is less than $\ell$ and so service *Bike Rent* is called. Now if *Car Rent* takes a significantly greater time to respond compared to *Bike Rent*, it is possible that the overall latency is shorter in the second case. That is, a degradation in the performance of a service (*HotelA* here) leads to an improvement in the overall performance of the orchestration.

A solution to this is to make the choice in the Travel Planner orchestration dependent on the orchestration's client. For e.g, if we alter this orchestration such that the client specifies in the start of the orchestration whether he wants to rent a car or a bike, the choice is resolved by the client. The exact execution path of the orchestration is known at the start, on receiving the client's request. This execution path is a partial order, which is monotonic. We could then have input-dependent contracts, e.g., promising a certain response time for a given set of input parameters and promising another response behaviour for a different set of inputs.

The orchestration to the right in Figure 3 assumes that HotelA's propositions are all close to the attraction sites, whereas those of HotelB are all far from them. The net on the left can thus be simplified to the guard-free net of the right.

The examples in figure 3 are non-monotonic due to the presence of choice followed by paths with different performances. In the sequel, we formally characterize the classes of orchestrations that are monotonic, giving both necessary and sufficient conditions for it. The formal material for this is introduced next.

## 3 The Orchestration Model: OrchNets

In this section we present the high level Petri Nets model for orchestrations that we use for our studies, which we call *OrchNets*. OrchNets are a special form of *colored occurrence nets (CO-nets).*

We have chosen this mathematical model for the following reasons. From the semantic studies performed for BPEL [9,2] and Orc [6,12], we know that we need to support in an elegant and succinct way the following features: concurrency, rich control patterns including preemption, representing data values, and for some cases even recursion. The first three requirements suggest using colored Petri nets. The last requirement suggests considering extensions of Petri nets with dynamicity. However, in our study we will not be interested in the specification of orchestrations, but rather in their executions. Occurrence nets are concurrent models of executions of Petri nets. As such, they encompass orchestrations involving recursion at no additional cost. The executions of Workflow Nets [13] are also CO-nets.

### 3.1 Background on Petri Nets and Occurrence Nets

A *Petri net* is a tuple $N = (\mathcal{P}, \mathcal{T}, \mathcal{F}, M_0)$, where: $\mathcal{P}$ is a set of *places*, $\mathcal{T}$ is a set of *transitions* such that $\mathcal{P} \cap \mathcal{T} = \emptyset$, $\mathcal{F} \subseteq (\mathcal{P} \times \mathcal{T}) \cup (\mathcal{T} \times \mathcal{P})$ is the *flow relation*, $M_0 : \mathcal{P} \to \mathbf{N}$ is the *initial marking*.

The elements in $\mathcal{P} \cup \mathcal{T}$ are called the *nodes* of $N$ and will be denoted by variables for *e.g*, $x$. For a node $x \in \mathcal{P} \cup \mathcal{T}$, we call ${}^\bullet x = \{y \mid (y, x) \in \mathcal{F}\}$ the *preset* of $x$, and $x^\bullet = \{y \mid (x,y) \in \mathcal{F}\}$ the *postset* of $x$. A *marking* of the net is a multiset $M$ of places, *i.e* a map from $\mathcal{P}$ to $\mathbf{N}$. A transition $t$ is *enabled* in marking $M$ if $\forall p \in {}^\bullet t, M(p) > 0$. This enabled transition can *fire* resulting in a new marking $M - {}^\bullet t + t^\bullet$ denoted by $M[t\rangle M'$. A marking $M$ is *reachable* if there exists a sequence of transitions $t_0, t_1 \dots t_n$ such that $M_0[t_0\rangle M_1[t_1\rangle \dots [t_n\rangle M$. A net is *safe* if for all reachable markings $M$, $M(\mathcal{P}) \subseteq \{0,1\}$.

For a net $N = (\mathcal{P}, \mathcal{T}, \mathcal{F}, M_0)$ the *causality relation* $<$ is the transitive closure of the flow relation $\mathcal{F}$. The reflexive closure of $<$ is denoted by $\leq$. For a node $x \in \mathcal{P} \cup \mathcal{T}$, the set of *causes* of $x$ is $\lceil x \rceil = \{y \in \mathcal{P} \cup \mathcal{T} \mid y \leq x\}$. Two nodes $x$ and $y$ are in *conflict* - denoted by $x\#y$ - if there exist distinct transitions $t, t' \in T$, such that $t \leq x, t' \leq y$ and ${}^\bullet t \cap {}^\bullet t' \neq \emptyset$. Nodes $x$ and $y$ are said to be *concurrent* - written as $x \| y$ - if neither $(x \leq y)$ nor $(y \leq x)$ nor $(x\#y)$. A set of concurrent places $P \subseteq \mathcal{P}$ is called a *co-set*. A *cut* is a maximal (for set inclusion) co-set.

A *configuration* of $N$ is a subnet $\kappa$ of nodes of $N$ such that:

1. $\kappa$ is *causally closed*, *i.e*, if $x < x'$ and $x' \in \kappa$ then $x \in \kappa$
2. $\kappa$ is *conflict-free*, *i.e*, for all nodes $x, x' \in \kappa, \neg(x\#x')$

For convenience, we will assume that the maximal nodes (w.r.t the $<$ relation) in a configuration are places.

A safe net $N = (\mathcal{P}, \mathcal{T}, \mathcal{F}, M_0)$ is called an *occurrence net (O-net)* iff

1. $\neg(x\#x)$ for every $x \in \mathcal{P} \cup \mathcal{T}$.
2. $\leq$ is a partial order and $\lceil t \rceil$ is finite for any $t \in \mathcal{T}$.
3. For each place $p \in \mathcal{P}$, $|{}^\bullet p| \leq 1$.
4. $M_0 = \{p \in \mathcal{P} | {}^\bullet p = \emptyset\}$, *i.e* the initial marking is the set of minimal places with respect to $\leq$.

Occurrence nets are a good model for representing the possible executions of a concurrent system. *Unfoldings* of a safe Petri net, which collect all the possible executions of the net, are occurrence nets. Unfoldings are defined as follows. For $N$ and $N'$ two safe nets, a map $\varphi : \mathcal{P} \cup \mathcal{T} \mapsto \mathcal{P}' \cup \mathcal{T}'$ is called a *morphism* of $N$ to $N'$ if: 1/ $\varphi(\mathcal{P}) \subseteq \mathcal{P}'$ and $\varphi(\mathcal{T}) \subseteq \mathcal{T}'$, and 2/ for every $t \in \mathcal{T}$ and $t' = \varphi(t) \in \mathcal{T}'$, ${}^\bullet t \cup \{t\} \cup t^\bullet$ is in bijection with ${}^\bullet t' \cup \{t'\} \cup t'^\bullet$ through $\varphi$. A *branching process* of a safe net $N$ is a pair $(U, \varphi)$ where $U$ is an occurrence net and $\varphi : U \mapsto N$ is a morphism such that 1/ $\varphi$ establishes a bijection between $M_0$ and the minimal places of $U$, and 2/ ${}^\bullet t = {}^\bullet t'$ and $\varphi(t) = \varphi(t')$ together imply $t = t'$. Branching processes are partially ordered (up to isomorphism) by the prefix order and there exists a unique maximal branching process called the *unfolding* of $N$ and denoted by $U_N$. The configurations of $U_N$ capture the executions of $N$, seen as partial orders of events. For a configuration $\kappa$ of an occurrence net $N$, the *future* of $\kappa$ in $N$, denoted by $N^\kappa$ is a sub-net of $N$ with the nodes:

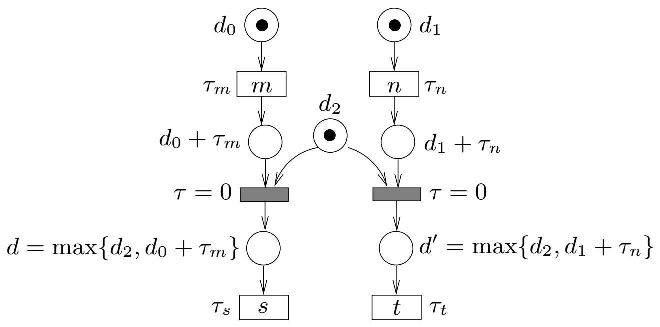
$$N^\kappa = \{x \in N \setminus \kappa \mid \forall x' \in \kappa, \neg(x\#x')\} \cup max(\kappa)$$

where $max(\kappa)$ is the set of maximal nodes of $\kappa$ (which are all places by our restriction on configurations).

## 3.2 Orchestration Model: OrchNets

We now present the orchestration model that we use for our studies, which we call *OrchNets*. OrchNets are occurrence nets in which

$$\text{tokens are equipped with a special attribute, referred to as a } \textit{color}, \text{ and consisting of a pair (value, date).} \qquad (1)$$



$$E = \begin{cases} \text{case } d < d' \text{ then } d + \tau_s \\ \text{case } d > d' \text{ then } d' + \tau_t \\ \text{otherwise nondeterministic} \end{cases}$$

**Fig. 4.** An OrchNet showing the dates of its tokens. The delay of a transition is shown next to it.

Figure 4 shows an OrchNet with its dates. Each place is labeled with a date which is the date of the token on reaching that place. Transitions are labeled with latencies. The tokens in the three minimal places are given *initial dates* (here, $d_0, d_1, d_2$). The four named transitions $m, n, s$ and $t$ are labeled with *latencies* $\tau_m, \tau_n, \tau_s$ and $\tau_t$ respectively, and the two shaded transitions have zero latency.

The presence of dates in tokens alters the firing semantics. A transition $t$ is enabled at a date when all places in its preset have tokens. and if its guard evaluates to *true* (absence of a guard is interpreted as the guard *true*). Once enabled, transition $t$ takes $\tau_t$ additional time to fire. For example, the shaded transition in the left has all its input tokens at $\max\{d_2, d_0 + \tau_m\}$ and so it fires at $\max\{d_2, d_0 + \tau_m\} + 0$ since it has zero latency. If a transition fires at date $d$, then the tokens in its postset have the date $d$. This is shown in the figure, e.g., on the place following the left shaded transition, which has date $\max\{d_2, d_0 + \tau_m\}$.

When transitions are in conflict, (e.g., the two shaded transitions in Figure 4), the transition that actually occurs is governed by a *race policy* [4,8]. If a set of enabled transitions are in conflict, the one with smallest date of occurrence will fire, preempting the other transitions in conflict with it. In Figure 4, the left or the right shaded transition will fire depending on whether $d < d'$ or $d > d'$ respectively, with a nondeterministic choice if $d = d'$. This results in selecting the left most or right most continuation (firing $s$ or $t$) accordingly. The resulting overall latency $E$ of the orchestration is shown at the bottom of the figure.

In addition to dates, tokens in OrchNets can have data attributes, which we call *values*. We have not shown this in Figure 4, in order to keep it simple. Values of tokens in the preset of a transition $t$ can be combined by a value function $\phi_t$ attached to $t$. The resulting value is taken by the token in the postset of $t$. At this point we are ready to provide the formal definition of *OrchNets*:

**Definition 1 (OrchNet).** *An* OrchNet *is a tuple* $\mathcal{N} = (N, \Phi, T, T_{\text{init}})$ *consisting of*

- *An occurrence net* $N$ *with token attributes* $c = (value, date)$*.*
- *A family* $\Phi = (\phi_t)_{t \in \mathcal{T}}$ *of* value functions, *whose inputs are the values of the transition's input tokens.*
- *A family* $T = (\tau_t)_{t \in \mathcal{T}}$ *of* latency functions, *whose inputs are the values of the transition's input tokens.*
- *A family* $T_{\text{init}} = (\tau_p)_{p \in \min(\mathcal{P})}$ *of* initial date functions *for the minimal places of* $N$*.*

In general, value, latency, and initial date functions can be nondeterministic. We introduce a global, invisible, daemon variable $\omega$ that resolves this nondeterminism and we denote by $\Omega$ its domain. That is, for a given value $\omega$ of this daemon, $\phi_t(\omega)$, $\tau_t(\omega)$, and $\tau_p(\omega)$ are all deterministic functions of their respective inputs.

## 3.3 The Semantics of OrchNets

We now explain how the presence of dates attached to tokens affects the semantics of OrchNets by adopting the so-called *race policy*. We first describe how a transition $t$ modifies the attributes of tokens. Let the preset of $t$ have $n$ places whose tokens have $(value, date)$ attributes $(v_1, d_1) \dots (v_n, d_n)$. Then all the tokens in the postset of $t$ have the pair $(v_t, d_t)$ of value and date, where:

$$\begin{aligned} v_t &= \phi_t(v_1 \dots v_n) \\ d_t &= \max\{d_1 \dots d_n\} + \tau_t(v_1 \dots v_n) \end{aligned} \tag{2}$$

The *race policy* for firing transitions is as follows. In any given marking $M$, let $T$ be the set of transitions that are *possibly enabled*, i.e. $\forall t \in T$, ${}^\bullet t$ is marked in $M$ and the guard of $t$ (if any) is true. Then the transition $t$ that is *actually enabled*, (which really fires) is given by:

$$t = \arg\min_{t \in T} d_t,$$
$$\text{where: } \arg\min_{x \in X} f(x) = x^* \in X \text{ s.t. } \forall x' \in X, f(x^*) \leq f(x').$$

If two possibly enabled transitions have the same $d_t$, then the choice of the transition that actually fires is non-deterministic. The race policy has the effect of filtering out configurations of OrchNets as explained now. Let $\mathcal{N} = (N, \Phi, T, T_{\text{init}})$ be a finite OrchNet. For a value $\omega \in \Omega$ for the daemon we can calculate the following *dates* for every transition $t$ and place $p$ of $\mathcal{N}$:

$$\begin{aligned} d_p(\omega) &= \tau_p(\omega) \textit{ if } p \textit{ is minimal, } d_s(\omega) \text{ where } s = {}^\bullet p \textit{ otherwise} \\ d_t(\omega) &= \max\{d_x(\omega) \mid x \in {}^\bullet t\} + \tau_t(\omega)(v_1, \dots v_n) \end{aligned} \tag{3}$$

where $v_1, \ldots v_n$ are the value components of the tokens in ${}^\bullet t$ as in equation (2). If $\kappa$ is a configuration of $N$, the future $\mathcal{N}^\kappa$ is the OrchNet $(N^\kappa, \Phi_{N^\kappa}, T_{N^\kappa}, T'_{\text{init}})$ where $\Phi_{N^\kappa}$ and $T_{N^\kappa}$ are the restrictions of $\Phi$ and $T$ respectively, to the transitions of $N^\kappa$. $T'_{\text{init}}$ is the family derived from $\mathcal{N}$ according to (3): for any minimal place $p$ of $N^\kappa$, the initialisation function is given by $\tau'_p(\omega) = d_p(\omega)$. For a net $N$ with the set of transitions $\mathcal{T}_N$, set $\mathcal{T}_{\min}(N) = \{t \in \mathcal{T}_N \mid {}^{\bullet\bullet}t \cap \mathcal{T}_N = \emptyset\}$. Let $min(\mathcal{P}_N)$ denote the minimal places of $N$. Now define $\kappa_0(\omega) = min(\mathcal{P}_N)$ and inductively,

$$\text{for } m > 0 : \kappa_m(\omega) = \kappa_{m-1}(\omega) \cup \{t_m\} \cup {}^\bullet t_m \cup t_m{}^\bullet \qquad (4)$$
$$\text{where } t_m = \arg \min_{t \in \mathcal{T}_{\min}(N^{\kappa_{m-1}(\omega)})} d_t(\omega)$$

Since net $N$ is finite, the above inductive definition terminates in finitely many steps when $N^{\kappa_m(\omega)} = \emptyset$. Let $M(\omega)$ be this number of steps. We thus have

$$\emptyset = \kappa_0 \subset \kappa_1(\omega) \cdots \subset \kappa_{M(\omega)}(\omega)$$

$\kappa_{M(\omega)}(\omega)$ is a maximal configuration of $\mathcal{N}$ that can actually occur according to the race policy, for a given $\omega \in \Omega$; such actually occurring configurations are generically denoted by

$$\overline{\kappa}(\mathcal{N}, \omega)$$

For $B$, a prefix-closed subset of the nodes of $N$ define

$$E_\omega(B, \mathcal{N}) = \max\{d_x(\omega) \mid x \in B\} \qquad (5)$$

If $B$ is a configuration, then $E_\omega(B, \mathcal{N})$ is the time taken for $B$ to execute (latency of $B$). The latency of the OrchNet $\mathcal{N} = (N, \Phi, T, T_{\text{init}})$ for a given $\omega$ is

$$E_\omega(\mathcal{N}) = E_\omega(\overline{\kappa}(\mathcal{N}, \omega), \mathcal{N}) \qquad (6)$$

Our design choices for the semantics of OrchNets were inspired by the application domain, i.e. compositions of web services. They reflect the following facts:

– Since we focus on latency, $\{\text{value}, \text{date}\}$ is the only color needed.
– Orchestrations rarely involve decisions on actions based on absolute dates. Timeouts are an exception, but these can be modelled explicitly, without using dates in guards of transitions. This justifies the fact that guards only have token values as inputs, and not their dates.
– The time needed to perform transitions does not depend on the tuple of dates $(d_1 \ldots d_n)$ when input tokens were created, but it can depend on the data $(v_1 \ldots v_n)$ and computation $\phi$ performed on these. This justifies our restriction for output arc expressions.

If it is still wished that control explicitly depends on dates, then dates must be measured and can then be stored as part of the value $v$.

## 4 Characterizing Monotonicity

In this article, we are interested in the total time taken to execute a web-service orchestration. As a consequence, we will consider only orchestrations that terminate in a finite time, *i.e*, only a finite number of values can be returned.

### 4.1 Defining and Characterizing Monotonicity

To formalize monotonicity we must specify how latencies and initial dates can vary. As an example, we may want to constrain some pair of transitions to have identical latencies. This can be stated by specifying a legal set of families of latency functions. For example, this legal set may accept any family $T = (\tau_t)_{t \in \mathcal{T}}$ such that two given transitions $t$ and $t'$ possess equal latencies: $\forall \omega \Rightarrow \tau_t(\omega) = \tau_{t'}(\omega)$. The same technique can be used for initial dates. Thus, the flexibility in setting latencies or initial dates can be formalized under the notion of pre-OrchNet we introduce next.

**Definition 2 (pre-OrchNet).** *Call* pre-OrchNet *a tuple* $\mathbb{N} = (N, \varPhi, \mathbb{T}, \mathbb{T}_{\text{init}})$*, where* $N$ *and* $\varPhi$ *are as before, and* $\mathbb{T}$ *and* $\mathbb{T}_{\text{init}}$ *are sets of families* $T$ *of latency functions and of families* $T_{\text{init}}$ *of initial date functions. Write* $\mathcal{N} \in \mathbb{N}$ *if* $\mathcal{N} = (N, \varPhi, T, T_{\text{init}})$ *for some* $T \in \mathbb{T}$ *and* $T_{\text{init}} \in \mathbb{T}_{\text{init}}$*.*

For two families $T$ and $T'$ of latency functions, write

$$T \geq T'$$

to mean that $\forall \omega \in \varOmega, \forall t \in \mathcal{T} \implies \tau_t(\omega) \geq \tau'_t(\omega)$, and similarly for $T_{\text{init}} \geq T'_{\text{init}}$. For $\mathcal{N}, \mathcal{N}' \in \mathbb{N}$, write

$$\mathcal{N} \geq \mathcal{N}' \quad \text{and} \quad E(\mathcal{N}) \geq E(\mathcal{N}')$$

to mean that $T \geq T'$ and $T_{\text{init}} \geq T'_{\text{init}}$ both hold, and $E_\omega(\mathcal{N}) \geq E_\omega(\mathcal{N}')$ holds for every $\omega$, respectively.

**Definition 3 (monotonicity).** *pre-OrchNet* $\mathbb{N} = (N, \varPhi, \mathbb{T}, \mathbb{T}_{\text{init}})$ *is called* monotonic *if, for any two* $\mathcal{N}, \mathcal{N}' \in \mathbb{N}$*, such that* $\mathcal{N} \geq \mathcal{N}'$*, we have* $E(\mathcal{N}) \geq E(\mathcal{N}')$*.*

**Theorem 1 (a global necessary and sufficient condition)**

1. *The following implies the monotonicity of pre-OrchNet* $\mathbb{N} = (N, \varPhi, \mathbb{T}, \mathbb{T}_{\text{init}})$*:*

$$\forall \mathcal{N} \in \mathbb{N}, \forall \omega \in \varOmega, \forall \overline{\kappa} \in \overline{\mathcal{V}}(N) \implies E_\omega(\overline{\kappa}, \mathcal{N}) \geq E_\omega(\overline{\kappa}(\mathcal{N}, \omega), \mathcal{N}) \quad (7)$$

   *where* $\overline{\mathcal{V}}(N)$ *denotes the set of all maximal configurations of net* $N$ *and* $\overline{\kappa}(\mathcal{N}, \omega)$ *is the maximal configuration of* $\mathcal{N}$ *that actually occurs under the daemon value* $\omega$*.*
2. *Conversely, assume that:*
   *(a) Condition (7) is violated, and*
   *(b) for any two OrchNets* $\mathcal{N}$ *and* $\mathcal{N}'$ *s.t.* $\mathcal{N} \in \mathbb{N}$*, then* $\mathcal{N}' \geq \mathcal{N} \Rightarrow \mathcal{N}' \in \mathbb{N}$*.*
   *Then* $\mathbb{N} = (N, \varPhi, \mathbb{T}, \mathbb{T}_{\text{init}})$ *is not monotonic.*

Statement 2 expresses that Condition (7) is also necessary provided that it is legal to increase at will latencies or initial dates. Observe that violating Condition (7) does not by itself cause non-monotonicity; as a counterexample, consider a case where $\mathbb{T}$ is a singleton for which (7) is violated—it is nevertheless monotonic.

The orchestration in the left of Figure 2 satisfies Theorem 1 trivially, since for any given $\omega$, there is only one possible maximal configuration. This is because the value of $d$ is fixed for a $\omega$ and only one branch of the two rental services is enabled. The orchestration in the left of Figure 3 does not fulfill Theorem 1. Consider an $\omega$ for which the actually occurring configuration $\kappa$ has both the responses of *HotelA* and *HotelB*. Say that $d > \ell$ for $\kappa$ and *Car Rent* is called. Now consider another configuration $\kappa'$ (under the same $\omega$), got by replacing *HotelA* by *Timer*. In this case, the response of *Hotel B* is used to calculate $d$, which may be different from that in configuration $\kappa$. This $d$ could be less than $\ell$ causing *Bike Rent* to be called. In this case, the latencies of *Car Rent* and *Bike Rent* can be set such that $E_\omega(\kappa, \mathcal{N}) > E_\omega(\kappa', \mathcal{N})$, violating Theorem 1.

## 4.2 A Structural Condition for the Monotonicity of Workflow Nets

*Workflow nets* [13] were proposed as a simple model for workflows. These are Petri nets, with a special minimal place $i$ and a special maximal place $o$. We consider the class of workflow nets that are 1-safe and which have no loops. Further, we require them to be *sound* [13]. A Workflow net $W$ is *sound* iff:

1. For every marking $M$ reachable from the initial place $i$, there is a firing sequence leading to the final place $o$.
2. If a marking $M$ marks the final place $o$, then no other place can in $W$ can be marked in $M$
3. There are no *dead* transitions in $W$. Starting from the initial place, it is always possible to fire any transition of $W$.

Workflow nets will be generically denoted by $W$. We can equip workflow nets with the same attributes as occurrence nets, this yields *pre-WFnets* $\mathbb{W} = (W, \Phi, \mathbb{T}, \mathbb{T}_{\text{init}})$. Referring to the end of Section 3.1, unfolding $W$ yields an occurrence net that we denote by $N_W$ with associated morphism $\varphi_W : N_W \mapsto W$. Here the morphism $\varphi_W$ maps the two $c$ transitions (and the place in its preset and postset) in the net on the right to the single $c$ transition (and its preset and postset) in the net on the left. Observe that $W$ and $N_W$ possess identical sets of minimal places. Morphism $\varphi_W$ induces a pre-OrchNet

$$\mathbb{N}_W = (N_W, \Phi_W, \mathbb{T}_W, \mathbb{T}_{\text{init}})$$

by attaching to each transition $t$ of $N_W$ the value and latency functions attached to $\varphi_W(t)$ in $\mathbb{W}$.

We shall use the results of the previous section in order to characterize those pre-WFnets whose unfoldings give monotonic pre-OrchNets. Our characterization will be essentially structural in that it does not involve any constraint on latency functions. Under this restricted discipline, the simple structural conditions we shall formulate will also be almost necessary. For this, we recall a notion of *cluster* [8] on nets. For a net $N$, a *cluster* is a (non-empty) minimal set $\mathbf{c}$ of places and transitions of $N$ such that $\forall t \in \mathbf{c},\ {}^\bullet t \subseteq \mathbf{c}$ and $\forall p \in \mathbf{c},\ p^\bullet \subseteq \mathbf{c}$.

**Theorem 2 (Sufficient Condition).** *Let $W$ be a WFnet and $N_W$ be its unfolding. A sufficient condition for the pre-OrchNet $\mathbb{N}_W = (N_W, \Phi_W, \mathbb{T}_W, \mathbb{T}_{\text{init}})$ to be monotonic is that every cluster* $\mathbf{c}$ *satisfies the following condition:*

$$\forall t_1, t_2 \in \mathbf{c},\ t_1 \neq t_2 \implies {t_1}^\bullet = {t_2}^\bullet \tag{8}$$

Recall that the sufficient condition for monotonicity stated in Theorem 1 is "almost necessary" in that, if enough flexibility exist in setting latencies and initial dates, then it is actually necessary. The same holds for the sufficient condition stated in Theorem 2 if the workflow net is assumed to be live.

**Theorem 3 (Necessary Condition).** *Suppose that the workflow net $W$ is sound. Assume that $\mathcal{W} \in \mathbb{W}$ and $\mathcal{W}' \geq \mathcal{W}$ implies $\mathcal{W}' \in \mathbb{W}$, meaning that there is enough flexibility in setting latencies and initial dates. In addition, assume that there is at least one $\mathcal{W}^* \in \mathbb{W}$ such that there is an daemon value $\omega^*$ for which the latencies of all the transitions are finite. Then the sufficient condition of Theorem 2 is also necessary for monotonicity.*

Observe that the orchestration in the right of figure 2 satisfies Theorem 2, whereas the orchestration in the right of figure 3 does not.

## 5 Probabilistic Monotonicity

So far we have considered the case where latencies of transitions are nondeterministic. In a previous work [10,11], on the basis of experiments performed on real Web services, we have advocated the use of probability distributions when modeling the response time of a Web service. Can we adapt our theory to encompass probabilistic latencies?

### 5.1 Probabilistic Setting, First Attempt

In Definitions 1 and 2, latency and initial date functions were considered nondeterministic. The first idea is to let them become random instead. This leads to the following straightforward modification of definitions 1 and 2:

**Definition 4 (probabilistic OrchNet and pre-OrchNet, 1).** *Call* probabilistic OrchNet *a tuple $\mathcal{N} = (N, \Phi, T, T_{\text{init}})$ where $\Phi = (\phi_t)_{t\in\mathcal{T}}$, $T = (\tau_t)_{t\in\mathcal{T}}$, and $T_{\text{init}} = (\tau_p)_{p\in\min(\mathcal{P})}$, are independent families of* random *value functions, latency functions, and initial date functions, respectively.*

*Call* probabilistic pre-OrchNet *a tuple $\mathbb{N} = (N, \Phi, \mathbb{T}, \mathbb{T}_{\text{init}})$, where $N$ and $\Phi$ are as before, and $\mathbb{T}$ and $\mathbb{T}_{\text{init}}$ are sets of families $T$ of* random *latency functions and of families $T_{\text{init}}$ of* random *initial date functions. Write $\mathcal{N} \in \mathbb{N}$ if $\mathcal{N} = (N, \Phi, T, T_{\text{init}})$ for some $T \in \mathbb{T}$ and $T_{\text{init}} \in \mathbb{T}_{\text{init}}$.*

We now equip random latencies and initial dates with a probabilistic ordering. If $\tau$ is a random latency function, its *distribution function* is defined by

$$F(x) = \mathbf{P}(\tau \leq x)$$

where $x \in \mathbb{R}_+$. Consider the following ordering: random latencies $\tau$ and $\tau'$ satisfy

$$\tau \geq_s \tau' \text{ if } F(x) \leq F'(x) \text{ holds } \forall x \in \mathbb{R}_+, \tag{9}$$

where $F$ and $F'$ are the distribution functions of $\tau$ and $\tau'$, respectively—with corresponding definition for the probabilistic ordering on initial date functions. Order (9) is classical in probability theory, where it is referred to as *stochastic dominance* or *stochastic ordering* among random variables [1].

Using order (9), for two families $T$ and $T'$ of random latency functions, write

$$T \geq_s T'$$

to mean that $\forall t \in \mathcal{T} \implies \tau_t \geq_s \tau'_t$, and similarly for $T_{\text{init}} \geq_s T'_{\text{init}}$. For $\mathcal{N}, \mathcal{N}' \in \mathbb{N}$, write

$$\mathcal{N} \geq_s \mathcal{N}'$$

if $T \geq_s T'$ and $T_{\text{init}} \geq_s T'_{\text{init}}$ both hold. Finally, the latency $E_\omega(\mathcal{N})$ of OrchNet $\mathcal{N}$ is itself seen as a random variable that we denote by $E(\mathcal{N})$, by removing symbol $\omega$. This allows us to define, for any two $\mathcal{N}, \mathcal{N}' \in \mathbb{N}$,

$$E(\mathcal{N}) \geq_s E(\mathcal{N}')$$

by requiring that random variables $E(\mathcal{N})$ and $E(\mathcal{N}')$ are stochastically ordered.

**Definition 5 (probabilistic monotonicity, 1).** *Probabilistic pre-OrchNet* $\mathbb{N}$ *is called* probabilistically monotonic *if, for any two* $\mathcal{N}, \mathcal{N}' \in \mathbb{N}$*, such that* $\mathcal{N} \geq_s \mathcal{N}'$*, we have* $E(\mathcal{N}) \geq_s E(\mathcal{N}')$*.*

It is a classical result on stochastic ordering that, if $(X_1, \ldots, X_n)$ and $(Y_1, \ldots, Y_n)$ are independent families of real-valued random variables such that $X_i \geq_s Y_i$ for every $1 \leq i \leq n$, then, for any increasing function $f : \mathbb{R}^n \to \mathbb{R}$, then $f(X_1, \ldots, X_n) \geq_s f(Y_1, \ldots, Y_n)$. Applying this yields that nondeterministic monotonicity in the sense of definition 3 implies probabilistic monotonicity in the sense of to definition 5. Nothing can be said, however, regarding the converse.

In order to derive results in the opposite direction, we shall establish a tighter link between this probabilistic framework and the nondeterministic framework of sections 3 and 4.

## 5.2 Probabilistic Setting: Second Attempt

Let us restart from the nondeterministic setting of sections 3 and 4. Focus on definition 1 of OrchNets. Equipping the set $\Omega$ of all possible values for the daemon with a probability $\mathbf{P}$ yields an alternative way to make the latencies and initial dates random. This suggests the following alternative setting for probabilistic monotonicity.

**Definition 6 (probabilistic OrchNet and pre-OrchNet, 2).** *Call* probabilistic OrchNet *a pair* $(\mathcal{N}, \mathbf{P})$*, where* $\mathcal{N}$ *is an OrchNet according to definition 1 and* $\mathbf{P}$ *is a probability over the domain* $\Omega$ *of all values for the daemon.*

*Call* probabilistic pre-OrchNet *a pair* $(\mathbb{N}, \mathbf{P})$, *where* $\mathbb{N}$ *is a pre-OrchNet according to definition* 2 *and* $\mathbf{P}$ *is a probability over the domain* $\Omega$ *of all values for the daemon.*

How can we relate the two definitions 4 and 6? Consider the following assumption, which will be in force in the sequel:

**Assumption 1.** *For any* $\mathcal{N} \in \mathbb{N}$, $\tau_t$ *and* $\tau_p$ *form an independent family of random variables, for* $t$ *ranging over the set of all transitions and* $p$ *ranging over the set of all minimal places of the underlying net.*

Let us now start from definition 4. For $t$ a generic transition, let $(\Omega_t, \mathbf{P}_t)$ be the set of possible experiments together with associated probability, for random latency $\tau_t$; and similarly for $(\Omega_p, \mathbf{P}_p)$ and $\tau_p$. Thanks to assumption 1, setting

$$\Omega = \Big(\prod_t \Omega_t\Big) \times \Big(\prod_p \Omega_p\Big) \quad \text{and} \quad \mathbf{P} = \Big(\prod_t \mathbf{P}_t\Big) \times \Big(\prod_p \mathbf{P}_p\Big), \tag{10}$$

yields the entities of definition 6. Can we use this correspondence to further relate probabilistic monotonicity to the notion of monotonicity of sections 3 and 4? In the nondeterministic framework of section 4, definition 2, we said that

$$\tau \geq \tau' \text{ if } \tau(\omega) \geq \tau'(\omega) \text{ holds } \forall \omega \in \Omega, \tag{11}$$

Clearly, if two random latencies $\tau$ and $\tau'$ satisfy condition (11), then they also satisfy condition (9). That is, *ordering* (11) *is stronger than stochastic ordering* (9). Unfortunately, the converse is *not* true in general. For example, condition (9) may hold while $\tau$ and $\tau'$ are two independent random variables, which prevents (11) from being satisfied. Nevertheless, the following routine result holds:

**Theorem 4.** *If condition* (9) *holds for the two distribution functions* $F$ *and* $F'$, *then there exists a probability space* $\Omega$, *a probability* $\mathbf{P}$ *over* $\Omega$, *and two real valued random variables* $\hat{\tau}$ *and* $\hat{\tau}'$ *over* $\Omega$, *such that:*

1. $\hat{\tau}$ *and* $\hat{\tau}'$ *possess* $F$ *and* $F'$ *as respective distribution functions, and*
2. *condition* (11) *is satisfied by the pair* $(\hat{\tau}, \hat{\tau}')$ *with probability* 1.

*Proof.* Take $\Omega = [0, 1]$ and $\mathbf{P}$ the Lebesgue measure. Then, taking, $\hat{\tau}(\omega) = \inf\{x \in \mathbb{R}_+ | F(x) \geq \omega\}$ and $\hat{\tau}'(\omega) = \inf\{x \in \mathbb{R}_+ | F'(x) \geq \omega\}$ yields the claim.

Theorem 4 allows reducing the stochastic comparison of real valued random variables to their ordinary comparison as functions defined over the same set of experiments endowed with a same probability. This applies in particular to each random latency function and each random initial date function, when considered in isolation. Thus, when performing construction (10) for two OrchNets $\mathcal{N}$ and $\mathcal{N}'$, we can take the *same* pair $(\Omega_t, \mathbf{P}_t)$ to represent both $\tau_t$ and $\tau'_t$, and similarly for $\tau_p$ and $\tau'_p$. Applying (10) implies that both $\mathcal{N}$ and $\mathcal{N}'$ are represented using the *same* pair $(\Omega, \mathbf{P})$. This leads naturally to definition 6.

In addition, applying theorem 4 to each transition $t$ and each minimal place $p$ yields that stochastic ordering $\mathcal{N} \geq_s \mathcal{N}'$ reduces to ordinary ordering $\mathcal{N} \geq \mathcal{N}'$. Observe that this trick does not apply to the overall latencies $E(\mathcal{N})$ and $E(\mathcal{N}')$ of the two OrchNets; the reason for this is that the space of experiments for these two random variables is already fixed (it is $\Omega$) and cannot further be played with as theorem 4 requires. Thus we can reformulate probabilistic monotonicity as follows—compare with definition 5:

**Definition 7 (probabilistic monotonicity, 2).** *Probabilistic pre-OrchNet* $(\mathbb{N}, \mathbf{P})$ *is called* probabilistically monotonic *if, for any two* $\mathcal{N}, \mathcal{N}' \in \mathbb{N}$*, such that* $\mathcal{N} \geq \mathcal{N}'$*, we have* $E(\mathcal{N}) \geq_s E(\mathcal{N}')$*.*

Note the careful use of $\geq$ and $\geq_s$. The following two results establish a relation between probabilistic monotonicity and monotonicity:

**Theorem 5.** *If pre-OrchNet* $\mathbb{N}$ *is monotonic, then, probabilistic pre-OrchNet* $(\mathbb{N}, \mathbf{P})$ *is probabilistically monotonic for any probability* $\mathbf{P}$ *over the set* $\Omega$*.*

This result was already obtained in the first probabilistic setting; it is here a direct consequence of the fact that $\tau \geq \tau'$ implies $\tau \geq_s \tau'$ if $\tau$ and $\tau'$ are two random variables defined over the same probability space. The following converse result completes the landscape and is much less straightforward. It assumes that it is legal to increase at will latencies or initial dates, see theorem 1:

**Theorem 6.** *Assume condition 2b of theorem 1 is satisfied. Then, if probabilistic pre-OrchNet* $(\mathbb{N}, \mathbf{P})$ *is probabilistically monotonic, then it is also monotonic with* $\mathbf{P}$*-probability* 1*.*

## 6 Getting Rid of Non-monotonicity

*Avoiding Non-Monotonicity.* We suggest a few ways in which non-monotonic orchestrations can be made monotonic. These might serve as guidelines to the designer of an orchestration, to avoid building non-monotonic orchestrations.

1. *Eliminate Choices.* We saw that choices in the execution flow can create non-monotonicity. So if possible, choices in the execution flow should be avoided while designing orchestrations. This seems very restrictive but is not totally unrealistic. For example, in the Travel Planner orchestration of figure 3, if the designer can find a rental service for both, cars and bikes, then the two mutually exclusive rental calls can be replaced by a call to that single rental service. This makes the execution flow an event graph and the Travel Planner orchestration monotonic.
2. *Balancing out performance of mutually exclusive branches.* One way to make an orchestration "more monotonic" is to ensure that all its mutually exclusive branches have similar response times. For e.g., in the Travel Planner example of figure 3, if the two exclusive services *Bike Rent* and *Car Rent* have similar response times, the orchestration is nearly monotonic.

3. *Externalising Choices.* Choices are of course integral to many execution flows and sometimes simply cannot be removed. A possible way out in this case is to *externalise* the choice and make them client dependent. This solution has already been discussed in the modified Travel Planner example of Section 2.
4. *If none of the above works,* then a brute force alternative consists in performing the following. Replace the orchestration latency $E_\omega(\mathcal{N})$ defined in (6) by the following pessimistic bound for it (see Theorem 1 for the notations):

$$F_\omega(\mathcal{N}) = \max \left\{ E_\omega(\kappa, \mathcal{N}) \mid \overline{\kappa} \in \overline{\mathcal{V}}(N) \right\} \tag{12}$$

   Then for any net $N$, and any two OrchNets $\mathcal{N}$ and $\mathcal{N}'$ over $N$, $\forall\omega$

$$F_\omega(\mathcal{N}) \geq E_\omega(\mathcal{N}) \tag{13}$$

$$\mathcal{N} \geq \mathcal{N}' \Rightarrow F_\omega(\mathcal{N}) \geq F_\omega(\mathcal{N}') \tag{14}$$

   holds. Therefore, using the pessimistic bound $F_\omega(\mathcal{N})$ instead of tight estimate $E_\omega(\mathcal{N})$ when building the orchestration's contract with its customer, is safe in that: 1) by (14), monotonicity of $F_\omega(\mathcal{N})$ with respect to the underlying OrchNet is guaranteed, and 2) by (13), the orchestration will meet its contract if its sub-contractors do so. In turn, this way of composing contracts is pessimistic and should therefore be avoided whenever possible.

*Where does monotonicity play a role in the orchestration's life cycle?* We use contracts to abstract the behaviour of the services involved in an orchestration. The orchestration, trusting these contracts, composes them to derive an estimate of its own performance, from which a contract between the orchestration and its customers can be established. Since this relies on trust between the orchestration and its sub-contractors, these contracts will have to be monitored at run-time to make sure that the sub-contractors deliver the promised performance. In case of violation, counter-measures like reconfiguring the orchestration might be taken. The orchestration's life cycle thus consists of the following phases [11]:

1. At *design time*, establish QoS contracts with the customer by composing QoS contracts from the called services; tune the monitoring algorithms accordingly; design reconfiguration strategy.
2. At *run time*, run the orchestration; in parallel, monitor the called services for possible QoS contract violation; whenever needed, perform reconfiguration.

Monotonicity plays a critical role at design time. The above pessimistic approach can be used as a backup solution if monotonicity is not satisfied. Monotonicity is however, not an issue at run time and the orchestration can be taken as such, with no modification. Monitoring of the called services remains unchanged too.

## 7 Conclusion

This paper is a contribution to the fundamentals of contract based QoS management of Web services orchestrations. QoS contracts implicitly assume monotonicity w.r.t. QoS parameters. We focus on one representative QoS parameter, namely response time. We have shown that monotonicity is easily violated in realistic cases. We have formalized monotonicity and have provided necessary and sufficient conditions for it. As we have seen, QoS can be very often traded for Quality of Data: poor quality responses to queries (including exceptions or invalid responses) can often be got much faster. This reveals that QoS parameters should not be considered separately, in isolation. We have provided guidelines for getting rid of non-monotonicity.

We see one relevant extension of this work: Advanced orchestration languages like Orc [7] offer a sophisticated form of preemption that are modelled by contextual nets (with read arcs). Our mathematical results do not consider nets with read arcs. Extending our results to this case would be interesting and useful.

## References

1. Anderson, G.: Nonparametric tests of stochastic dominance in income distributions. Econometrica 64(5), 1183–1193 (1996)
2. Arias-Fisteus, J., Fernández, L.S., Kloos, C.D.: Applying model checking to BPEL4WS business collaborations. In: SAC, pp. 826–830 (2005)
3. OASIS WSBPEL Technical Committee. Web Services Business Process Execution Language Version 2.0. OASIS Standard (April 2007)
4. Kartson, D., Balbo, G., Donatelli, S., Franceschinis, G., Conte, G.: Modelling with Generalized Stochastic Petri Nets. John Wiley & Sons, Inc., NY (1994)
5. Keller, A., Ludwig, H.: The wsla framework: Specifying and monitoring service level agreements for web services. J. Network Syst. Manage. 11(1) (2003)
6. Kitchin, D., Cook, W.R., Misra, J.: A language for task orchestration and its semantic properties. In: Baier, C., Hermanns, H. (eds.) CONCUR 2006. LNCS, vol. 4137, pp. 477–491. Springer, Heidelberg (2006)
7. Misra, J., Cook, W.R.: Computation orchestration: A basis for wide-area computing. Journal of Software and Systems Modeling (May 2006), http://dx.doi.org/10.1007/s10270-006-0012-1
8. Murata, T.: Petri nets: Properties, analysis and applications. Proceedings of the IEEE 77(4), 541–580 (1989)
9. Ouyang, C., Verbeek, E., van der Aalst, W.M.P., Breutel, S., Dumas, M., ter Hofstede, A.H.M.: Formal semantics and analysis of control flow in ws-bpel. Sci. Comput. Program. 67(2-3), 162–198 (2007)
10. Rosario, S., Benveniste, A., Haar, S., Jard, C.: Probabilistic QoS and soft contracts for transaction based web services. In: ICWS, pp. 126–133 (2007)
11. Rosario, S., Benveniste, A., Haar, S., Jard, C.: Probabilistic QoS and Soft Contracts for Transaction based Web Services. IEEE Transactions on Services Computing 1(4), 187–200 (2008)
12. Rosario, S., Kitchin, D., Benveniste, A., Cook, W.R., Haar, S., Jard, C.: Event structure semantics of orc. In: Dumas, M., Heckel, R. (eds.) WS-FM 2007. LNCS, vol. 4937, pp. 154–168. Springer, Heidelberg (2008)
13. van der Aalst, W.M.P.: Verification of workflow nets. In: Azéma, P., Balbo, G. (eds.) ICATPN 1997. LNCS, vol. 1248, pp. 407–426. Springer, Heidelberg (1997)
14. Zeng, L., Benatallah, B., Ngu, A.H.H., Dumas, M., Kalagnanam, J., Chang, H.: QoS-Aware Middleware for Web Services Composition. IEEE Trans. Software Eng. 30(5), 311–327 (2004)

## A Collecting Proofs

### A.1 Proof of Theorem 1

*Proof.* We first prove Statement 1. Let $\mathcal{N}' \in \mathbb{N}$ be such that $\mathcal{N}' \geq \mathcal{N}$. We have:

$$E_\omega(\overline{\kappa}(\mathcal{N}',\omega),\mathcal{N}') \geq E_\omega(\overline{\kappa}(\mathcal{N}',\omega),\mathcal{N}) \geq E_\omega(\overline{\kappa}(\mathcal{N},\omega),\mathcal{N})$$

where the first inequality follows from the fact that $\overline{\kappa}(\mathcal{N}',\omega)$ is a conflict free partial order and $\mathcal{N}' \geq \mathcal{N}$, and the second inequality follows from (7) applied with $\overline{\kappa} = \overline{\kappa}(\mathcal{N}',\omega)$. This proves Statement 1.

We prove statement 2 by contradiction. Let $(\mathcal{N},\omega,\overline{\kappa}^\dagger)$ be a triple violating Condition (7), in that

$$\overline{\kappa}^\dagger \text{ cannot occur, but } E_\omega(\overline{\kappa}^\dagger,\mathcal{N}) < E_\omega(\overline{\kappa}(\mathcal{N},\omega),\mathcal{N}) \text{ nevertheless holds.}$$

Now consider the OrchNet net $\mathcal{N}' = (N,\Phi,T',T_{\text{init}})$ where the family $T'$ is the same as $T$ except that in $\omega$, $\forall t \notin \overline{\kappa}^\dagger$, $\tau'_t(\omega) > E_\omega(\overline{\kappa}^\dagger,\mathcal{N})$. Clearly $\mathcal{N}' \geq \mathcal{N}$. But using construction (4), it is easy to verify that $\overline{\kappa}(\mathcal{N}',\omega) = \overline{\kappa}^\dagger$ and thus

$$E_\omega(\overline{\kappa}(\mathcal{N}',\omega),\mathcal{N}') = E_\omega(\overline{\kappa}^\dagger,\mathcal{N}') = E_\omega(\overline{\kappa}^\dagger,\mathcal{N}) < E_\omega(\overline{\kappa}(\mathcal{N},\omega),\mathcal{N}),$$

which violates monotonicity.

### A.2 Proof of Theorem 2

*Proof.* Let $\varphi_W$ be the net morphism mapping $N_W$ onto $W$ and let $\mathcal{N} \in \mathbb{N}$ be any OrchNet. We prove that condition 1 of Theorem 1 holds for $\mathcal{N}$ by induction on the number of transitions in the maximal configuration $\overline{\kappa}(\mathcal{N},\omega)$ that actually occurs. The base case is when it has only one transition. Clearly this transition has the least latency and any other maximal configuration has a greater execution time.

*Induction Hypothesis.* Condition 1 of Theorem 1 holds for any maximal occurring configuration with $m-1$ transitions ($m > 1$). Formally, for a pre-OrchNet $\mathbb{N} = (N,\Phi,\mathbb{T},\mathbb{T}_{\text{init}})$: $\forall \mathcal{N} \in \mathbb{N}, \forall \omega \in \Omega, \forall \overline{\kappa} \in \overline{\mathcal{V}}(N)$,

$$E_\omega(\overline{\kappa},\mathcal{N}) \geq E_\omega(\overline{\kappa}(\mathcal{N},\omega),\mathcal{N}) \tag{15}$$

holds if $|\{t \in \overline{\kappa}(\mathcal{N},\omega)\}| \leq m-1$.

*Induction Argument.* Consider the OrchNet $\mathcal{N}$, where the actually occurring configuration $\overline{\kappa}(\mathcal{N},\omega)$ has $m$ transitions. $\kappa'$ is any other maximal configuration of $\mathcal{N}$. If the transition $t$ in $\overline{\kappa}(\mathcal{N},\omega)$ with minimal date $d_t$ also occurs in $\kappa'$ then comparing execution times of $\overline{\kappa}(\mathcal{N},\omega)$ and $\kappa'$ reduces to comparing $E_\omega(\overline{\kappa}(\mathcal{N},\omega)\setminus\{t\},\mathcal{N}^t)$ and $E_\omega(\kappa' \setminus \{t\},\mathcal{N}^t)$. Since $\overline{\kappa}(\mathcal{N},\omega) \setminus \{t\}$ is the actually occurring configuration in the future $\mathcal{N}^t$ of transition $t$, using our induction hypothesis, we have

$$E_\omega(\overline{\kappa}(\mathcal{N},\omega) \setminus \{t\},\mathcal{N}^t) \leq E_\omega(\kappa' \setminus \{t\},\mathcal{N}^t)$$

and so

$$E_\omega(\overline{\kappa}(\mathcal{N},\omega),\mathcal{N}) \ \leq E_\omega(\kappa',\mathcal{N})$$

If $t \notin \kappa'$ for some $\kappa'$, then there must exist another transition $t'$ such that ${}^\bullet t \cap {}^\bullet t' \neq \emptyset$. By the definition of clusters, $\varphi_W(t)$ and $\varphi_W(t')$ must belong to the same cluster $\mathbf{c}$. Hence, $t^\bullet = t'^\bullet$ follows from condition 8 of Theorem 2. The futures $\mathcal{N}^t$ and $\mathcal{N}^{t'}$ thus have identical sets of transitions: they only differ in the initial marking of their places. If $T_{init}$ and $T'_{init}$ are the initial marking of these places, $T_{init} \leq T'_{init}$ (since $d_t \leq d_{t'}$, $t^\bullet$ has dates lesser than $t'^\bullet$). Hence

$$E_\omega(\overline{\kappa}(\mathcal{N},\omega),\mathcal{N}) \ = E_\omega(\overline{\kappa}(\mathcal{N},\omega) \setminus \{t\},\mathcal{N}^t) \tag{16}$$

and

$$E_\omega(\kappa',\mathcal{N}) \ = E_\omega(\kappa' \setminus \{t'\},\mathcal{N}^{t'}) \geq E_\omega(\kappa' \setminus \{t'\},\mathcal{N}^t) \tag{17}$$

The inequality holds since $\mathcal{N}^{t'} \geq \mathcal{N}^t$. The induction hypothesis on (16) and (17) gives $E_\omega(\overline{\kappa}(\mathcal{N},\omega),\mathcal{N}) \ \leq E_\omega(\kappa',\mathcal{N})$. This proves the theorem.

## A.3 Proof of Theorem 3

*Proof.* We will show that when condition (8) of Theorem 2 is not satisfied by $W$, the Orchnets in its induced preOrchNet $\mathbb{N}_W$ can violate condition (7) of Theorem 1, the necessary condition for monotonicity.

Let $c_W$ be any cluster in $W$ that violates the condition 8 of Theorem 2. Consider the unfolding of $W$, $N_W$ and the associated morphism $\varphi : N_W \mapsto W$ as introduced before. Since $W$ is sound, all transitions in $c_W$ are reachable from the initial place $i$ and so there is a cluster $c$ in $N_W$ such that $\varphi(c) = c_W$. There are transitions $t_1, t_2 \in c$ such that ${}^\bullet t_1 \cap {}^\bullet t_2 \neq \emptyset$, ${}^\bullet\varphi(t_1) \cap {}^\bullet\varphi(t_2) \neq \emptyset$ and $\varphi(t_1)^\bullet \neq \varphi(t_2)^\bullet$. Call $[t] = \lceil t \rceil \setminus \{t\}$ and define $K = [t_1] \cup [t_2]$. We consider the following two cases:

*K is a configuration.* If so, consider the OrchNet $\mathcal{N}^* \in \mathbb{N}_W$ obtained when transitions of $N_W$ (and so $W$) have latencies as that in $\mathcal{W}^*$. So for the daemon value $\omega^*$, the quantity $E_{\omega^*}(K,\mathcal{N}^*)$ is some finite value $\mathbf{n}^*$. Now, configuration $K$ can actually occur in a OrchNet $\mathcal{N}$, such that $\mathcal{N} > \mathcal{N}^*$, where $\mathcal{N}$ is obtained as follows ($\tau$ and $\tau^*$ denote the latencies of transitions in $\mathcal{N}$ and $\mathcal{N}^*$ respectively): $\forall t \in K, t' \in N_W$ s.t. ${}^\bullet t \cap {}^\bullet t' \neq \emptyset$, set $\tau_{t'}(\omega^*) = \mathbf{n}^* + 1$ and keep the other latencies unchanged. In this case, for the daemon value $\omega^*$, the latencies of all transitions of $\mathcal{N}$ (and so its overall execution time) is finite. Denote by $\mathcal{N}^K$ the future of $\mathcal{N}$ once configuration $K$ has actually occurred. Both $t_1$ and $t_2$ are minimal and enabled in $\mathcal{N}^K$.

Since $\varphi(t_1)^\bullet \neq \varphi(t_2)^\bullet$, without loss of generality, we assume that there is a place $p \in t_1{}^\bullet$ such that $\varphi(p) \in \varphi(t_1)^\bullet$ but $\varphi(p) \notin \varphi(t_2)^\bullet$. Let $t^*$ be a transition in $\mathcal{N}^K$ such that $t^* \in p^\bullet$. Such a transition must exist since $p$ can not be a maximal place: $\varphi(p)$ can not be a maximal place in $W$ which has a unique maximal place. Now consider the Orchnet $\mathcal{N}' > \mathcal{N}$ obtained as follows: $\tau'_{t_1}(\omega^*) =$

$\tau_{t_1}(\omega^*), \tau'_{t_2}(\omega^*) = \tau_{t_1}(\omega^*) + 1$ and for all other $t \in c, \tau'_t(\omega^*) = \tau'_{t_2}(\omega^*) + 1$. Set $\tau'_{t^*}(\omega^*) = \infty$ and for all other transitions of $\mathcal{N}'$, the delays are the same as that in $\mathcal{N}$ and thus are finite for $\omega^*$.

$t_1$ has the minimal delay among all transitions in $c$, and $t^*$ is in the future of $t_1$. So the actually occurring configuration $E_{\omega^*}(\overline{\kappa}(\mathcal{N}', \omega^*), \mathcal{N}')$ has an infinite delay. However any maximal configuration $\overline{\kappa}$ which does not include $t_1$ (for eg, when $t_2$ fires instead of $t_1$) will have a finite delay. For such $\overline{\kappa}$ we thus have $E_{\omega^*}(\overline{\kappa}(\mathcal{N}', \omega^*), \mathcal{N}') > E_{\omega^*}(\overline{\kappa}, \mathcal{N}')$ and so $\mathcal{N}'$ violates the condition (7) of Theorem 1.

*K is not a configuration.* If so, there exist transitions $t \in [t_1] \setminus [t_2]$, $t' \in [t_2] \setminus [t_1]$ such that ${}^\bullet t \cap {}^\bullet t' \neq \emptyset$, ${}^\bullet\varphi(t) \cap {}^\bullet\varphi(t') \neq \emptyset$ and $\varphi(t)^\bullet \neq \varphi(t')^\bullet$. The final condition holds since $t_2$ and $t_1$ are not in the causal future of $t$ and $t'$ respectively. Thus $t$ and $t'$ belong to the same cluster, which violates condition 8 of Theorem 2 and we can apply the same reasoning as in the beginning of the proof. Since $[t]$ is finite for any transition $t$, we will eventually end up with $K$ being a configuration.

### A.4 Proof of Theorem 6

*Proof.* The proof is by contradiction. Assume that $\mathbb{N}$ is *not* monotonic with positive $\mathbf{P}$-probability, i.e., :

$$\begin{array}{l}\text{there exists a pair } (\mathcal{N}, \mathcal{N}') \text{ of OrchNets such that} \\ \mathcal{N} \geq \mathcal{N}' \text{ and } \mathbf{P}\{\omega \in \Omega \mid E_\omega(\mathcal{N}) < E_\omega(\mathcal{N}')\} > 0.\end{array} \tag{18}$$

To prove the theorem it is enough to prove that (18) implies:

$$\begin{array}{l}\text{there exists } \mathcal{N}_o, \mathcal{N}'_o \in \mathbb{N} \text{ such that } \mathcal{N}_o \geq \mathcal{N}'_o, \\ \text{but } E(\mathcal{N}_o) \geq_s E(\mathcal{N}'_o) \text{ does not hold}\end{array} \tag{19}$$

To this end, set $\mathcal{N}_o = \mathcal{N}$ and define $\mathcal{N}'_o$ as follows, where $\Omega_o$ denotes the set $\{\omega \in \Omega \mid E_\omega(\mathcal{N}) < E_\omega(\mathcal{N}')\}$:

$$\mathcal{N}'_o(\omega) = \text{ if } \omega \in \Omega_o \text{ then } \mathcal{N}'(\omega) \text{ else } \mathcal{N}(\omega)$$

Note that $\mathcal{N}_o \geq \mathcal{N}'_o$ by construction. Also, $\mathcal{N}'_o \geq \mathcal{N}'$, whence $\mathcal{N}'_o \in \mathbb{N}$ since condition 2b of theorem 1 is satisfied. On the other hand, we have $E_\omega(\mathcal{N}_o) < E_\omega(\mathcal{N}'_o)$ for $\omega \in \Omega_o$, and $E_\omega(\mathcal{N}_o) = E_\omega(\mathcal{N}'_o)$ for $\omega \notin \Omega_o$. By (18), we have $\mathbf{P}(\Omega_o) > 0$. Consequently, we get:

$$[\forall \omega \in \Omega \ \Rightarrow \ E_\omega(\mathcal{N}_o) \leq E_\omega(\mathcal{N}'_o)] \quad \text{and} \quad [\mathbf{P}\{\omega \in \Omega \mid E_\omega(\mathcal{N}_o) < E_\omega(\mathcal{N}'_o)\} > 0]$$

which implies that $E(\mathcal{N}_o) \geq_s E(\mathcal{N}'_o)$ does not hold.

# Compositional Service Trees

Wil M.P. van der Aalst[1], Kees M. van Hee[1], Peter Massuthe[2], Natalia Sidorova[1], and Jan Martijn van der Werf[1]

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

[2] Humboldt-Universität zu Berlin, Institut für Informatik, Unter den Linden 6, 10099 Berlin, Germany
massuthe@informatik.hu-berlin.de

**Abstract.** In the world of Service Oriented Architectures, one deals with networks of cooperating components. A component offers services; to deliver a service it possibly needs services of other components, thus forming a *service tree.* This tree is built dynamically and not known beforehand. It is hard to verify the behavior of a service tree by using standard verification techniques, because these techniques typically assume a static flattened model. In this paper we model a component by an open Petri net. We give a sufficient condition for proper completion (called soundness) that requires only pairwise checks of the service compositions. We also provide a correctness-by-construction approach for building services trees.

## 1 Introduction

According to the paradigm of Service Oriented Architectures (SOA) a system can be seen as a (possibly open) network of cooperating *components* based on asynchronous communication [3, 8, 18]. A component offers *services* to a *client* which may be a component itself. Each component may play two roles: service provider and service client. One of the interesting features of SOA is the *dynamic binding* of services: in order to provide a service $S$ for its client, a component may invoke a service $S'$ of another component during the execution, while it might be not known at the beginning of this execution that the service $S'$ was needed and which component would be selected to deliver it. In this way, the components form a tree, called a *service tree*, to *deliver* a certain service. However, this tree is not known to any party and for privacy and security reasons we often do not want the tree to be known to anybody. This makes the verification of behavioral correctness very hard.

Correctness requirements concern both dataflow and control flow. In this paper, we focus on the control flow aspect, i.e., on the *orchestration.* There are several approaches to define the orchestration process. BPEL (Business process Execution Language) is one of the main languages to specify the orchestration

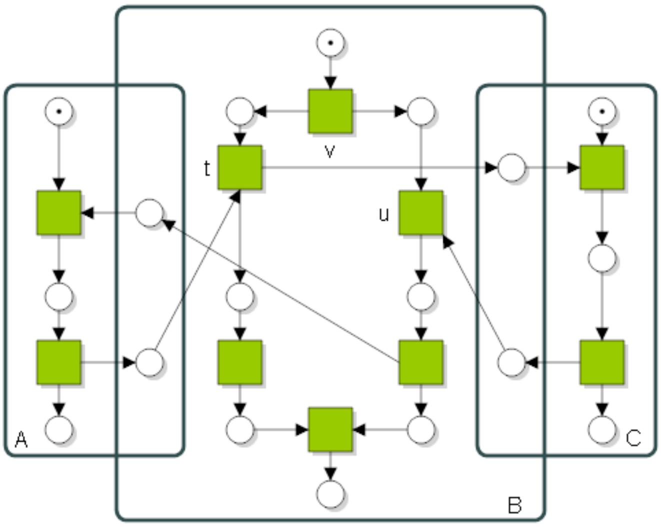


**Fig. 1.** The composition of three composable OWNs

at the implementation level [9]. We use here Petri nets for modeling the orchestration of components in order to analyse the behavioral properties.

Due to dynamic binding, we never know how a service tree will look like, and therefore, we are not able to check a complete service tree beforehand. Thus, if we want ensure proper termination we need a verification method that only considers pairwise compositions of components. Therefore, we propose a *correctness-by-construction* method that guarantees that if each pair of connected components satisfies some *correctness condition*, then the whole service tree will be sound.

We model services and service trees as *open Petri nets* (OPNs), i.e., a Petri net with a set of interface (input or output) places [5, 15, 18]. An OPN has one initial marking and one final marking, which is usually a deadlock. A composition of two OPNs is an OPN again; the corresponding interface places of the two nets are fused and they are not a part of the interface of the resulting net anymore. Sometimes, we consider a more restrictive class of OPNs, called *open workflow nets* (OWNs) which have one initial place, one final place, and an arbitrary number of interface places. Both OPNs and OWNs can be seen as a generalization of the classical workflow nets [1, 10].

The behavioral correctness criterion we consider is the *weak termination* property of services, which can be seen as generalization of the *soundness* concept of workflow nets [1], and therefore we also call it soundness. A stand-alone OPN is called *sound* if for each marking reachable from the initial marking the final marking is reachable, discarding the interface places.

We illustrate the need for compositional soundness conditions for service trees on the example shown in Figure 1. Here, we have three components, $A$, $B$, and $C$. The composition of $A$ with $B$ is sound, as well as the composition of $B$ with

$C$. However, the composition of the three has a deadlock, since this composition introduces a cyclic dependency implying a deadlock.

We will consider two approaches: a *posteriori* approach and a *constructive* approach. In the posteriori approach for each pair of composed components in a service tree we have to verify the correctness condition by checking a specific *simulation* relation. In the constructive approach we apply stepwise refinement. We start with a service tree of two composed components, which are known to satisfy the correctness condition. Then we select two so-called *synchronized* places and we refine them simultaneously by correctly composed components. We can also extend the tree by adding new leaves to it at arbitrary nodes.

In Section 2 we give basic definitions. In Section 3 we introduce the basic concepts of OPNs. In Section 4 we define the composition operator for OPNs and give sufficient conditions for soundness of service trees. In Section 5 we introduce the stepwise refinement approach in general and in Section 6 we present a correct-by-construction method for service trees. Finally, we conclude with related and future work in Section 7.

## 2 Preliminaries

Let $S$ be a (finite) set. The powerset of $S$ is denoted by $\mathcal{P}(S) = \{S' \mid S' \subseteq S\}$. We use $|S|$ for the number of elements in $S$. Two sets $U$ and $V$ are *disjoint* if $U \cap V = \emptyset$. A *bag* $m$ over $S$ is a function $m : S \rightarrow \mathbb{N}$. We denote e.g. the bag $m$ with an element $a$ occurring once, $b$ occurring three times and $c$ occurring twice by $m = [a, b^3, c^2]$. The set of all bags over $S$ is denoted by $\mathbb{N}^S$. Sets can be seen as a special kind of bag were all elements occur only once. We use $+$ and $-$ for the sum and difference of two bags, and $=, <, >, \leq, \geq$ for the comparison of two bags, which are defined in a standard way. The projection of a bag $m \in \mathbb{N}^S$ on elements of a set $U \subseteq S$, is denoted by $m_{|U}$, and is defined by $m_{|U}(u) = m(u)$ for all $u \in U$ and $m_{|U}(u) = 0$ for all $u \in S \setminus U$. Furthermore, if for some $n \in \mathbb{N}$, disjoint sets $U_i \subseteq S$ with $1 \leq i \leq n$ exist such that $S = \bigcup_{i=1}^{n} U_i$, then $m = \sum_{i=1}^{n} m_{|U_i}$.

A *sequence* over $S$ of length $n \in \mathbb{N}$ is a function $\sigma : \{1, \ldots, n\} \rightarrow S$. If $n > 0$ and $\sigma(i) = a_i$ for $i \in \{1, \ldots, n\}$, we write $\sigma = \langle a_1, \ldots, a_n \rangle$, and $\sigma_i$ for $\sigma(i)$. The length of a sequence is denoted by $|\sigma|$. The sequence of length 0 is called the *empty sequence*, and is denoted by $\epsilon$. The set of all finite sequences over $S$ is denoted by $S^*$. Let $\nu, \gamma \in S^*$ be two sequences. *Concatenation*, denoted by $\sigma = \nu; \gamma$ is defined as $\sigma : \{1, \ldots, |\nu| + |\gamma|\} \rightarrow S$, such that for $1 \leq i \leq |\nu|$: $\sigma(i) = \nu(i)$, and for $|\nu| + 1 \leq i \leq |\nu| + |\gamma|$: $\sigma(i) = \gamma(i - |\nu|)$. A projection of a sequence $\sigma \in S^*$ on elements of a set $U \subseteq S$ (i.e. eliminating the elements from $S \setminus U$) is denoted as $\sigma_{|U}$.

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

**Definition 1 (Labeled transition system).** *A* labeled transition system *(LTS) is a* 5*-tuple* $(S, \mathcal{A}, \longrightarrow, s_i, s_f)$ *where*

- *$S$ is a set of* states*;*
- *$\mathcal{A}$ is a set of* actions*;*
- *$\longrightarrow \subseteq S \times (\mathcal{A} \cup \{\tau\}) \times S$ is a* transition relation*, where $\tau \notin \mathcal{A}$ is the silent action [10].*
- *$s_i \in S$ is the initial state;*
- *$s_f \in S$ is the final state.*

For $m, m' \in S$ and $a \in \mathcal{A}$, we write $L : m \xrightarrow{a} m'$ if and only if $(m, a, m') \in \longrightarrow$. If the context is clear, we omit the $L$. A state $m \in S$ is called a *deadlock* if no action $a \in \mathcal{A} \cup \{\tau\}$ and state $m' \in S$ exist such that $m \xrightarrow{a} m'$. We often require that $s_f$ is a deadlock.

We define $\Longrightarrow$ as the smallest relation such that $m \Longrightarrow m'$ if $m = m' \vee \exists m'' \in S : m \Longrightarrow m'' \xrightarrow{\tau} m'$. $m \Longrightarrow m'$ is sometimes also referred to as $m \overset{\tau}{\Longrightarrow} m'$, i.e., a path of zero or more silent actions [10]. We define $\overset{a}{\Longrightarrow}$ as the smallest relation such that $m \overset{a}{\Longrightarrow} m'$ if $\exists m'' \in S : m \Longrightarrow m'' \xrightarrow{a} m'$.

**Definition 2 (Hiding).** *Let $L = (S, \mathcal{A}, \longrightarrow, s_i, s_f)$ be an LTS. Let $H \subseteq \mathcal{A}$. We define the operation $\tau_H$ on an LTS by $\tau_H(L) = (S, \mathcal{A} \setminus H, \longrightarrow', s_i, s_f)$, where for $m, m' \in S$ and $a \in \mathcal{A}$ we have $(m, a, m') \in \longrightarrow'$ if and only if $(m, a, m') \in \longrightarrow$ and $a \notin H$ and $(m, \tau, m') \in \longrightarrow'$ if and only if $(m, \tau, m') \in \longrightarrow$ or $(m, a, m') \in \longrightarrow$ and $a \in H$.*

We define simulation and bisimulation relations on labeled transition systems.

**Definition 3 (Simulation, bisimulation).** *Let $L = (S, \mathcal{A}, \longrightarrow, s_i, s_f)$ and $L' = (S', \mathcal{A}, \longrightarrow', s'_i, s'_f)$ be two LTSs. The relation $R \subseteq S \times S'$ is a* simulation*, denoted by $L \preceq_R L'$, if:*

1. *$s_i \, R \, s'_i$ and $s_f \, R \, s'_f$;*
2. *$\forall m, m' \in S, \bar{m} \in S', a \in \mathcal{A} \cup \{\tau\} : (m \xrightarrow{a} m' \wedge m \, R \, \bar{m}) \Rightarrow (\exists \bar{m}' \in S' : \bar{m} \overset{a}{\Longrightarrow}' \bar{m}' \wedge m' \, R \, \bar{m}')$.*
3. *$\forall m' \in S' : (s_f \, R \, m') \Rightarrow (m' \Longrightarrow s'_f)$.*

*If both $R$ and $R^{-1}$ are simulations, $R$ is a* bisimulation*.*

Note that $L \preceq_R L'$ means that $L'$ can mimic $L$, i.e., $L'$ is able to *simulate* $L$.

**Petri nets.** A *Petri net* is a 3-tuple $N = (P, T, F)$ where (1) $P$ and $T$ are two disjoint sets of *places* and *transitions* respectively; (2) $F \subseteq (P \times T) \cup (T \times P)$ is a *flow relation*. The elements from the set $P \cup T$ are called the *nodes* of $N$. Elements of $F$ are called *arcs*. Places are depicted as circles, transitions as squares. For each element $(n_1, n_2) \in F$, an arc is drawn from $n_1$ to $n_2$. Two Petri nets $N = (P, T, F)$ and $N' = (P', T', F')$ are *disjoint* if and only if $(P \cup T) \cap (P' \cup T') = \emptyset$. Let $N = (P, T, F)$ be a Petri net. Given a node $n \in (P \cup T)$, we define its *preset* $\overset{N}{\bullet} n = \{n' \mid (n', n) \in F\}$, and its *postset* $n \overset{N}{\bullet} = \{n' \mid (n, n') \in F\}$. We lift the notation of preset and postset to sets. Given a set $U \subseteq (P \cup T)$, $\overset{N}{\bullet} U = \bigcup_{n \in U} \overset{N}{\bullet} n$ and $U \overset{N}{\bullet} = \bigcup_{n \in U} n \overset{N}{\bullet}$. If the context is clear, we omit the $N$ in the superscript.

Markings are states of a net. A *marking* $m$ of a Petri net $N = (P, T, F)$ is defined as a bag over $P$. A pair $(N, m)$ is called a *marked Petri net.* A transition $t \in T$ is enabled in a marking $m \in \mathbb{N}^P$ if and only if $^\bullet t \leq m$. An enabled transition may *fire.* A transition firing results in a new marking $m'$ with $m' = m - {}^\bullet t + t^\bullet$, denoted by $N : m \xrightarrow{t} m'$. If the context is clear, we omit the $N$. A sequence $\sigma = \langle t_1, \ldots, t_n \rangle$ is a *firing sequence* of $(N, m)$ if and only if there exist markings $m_1, \ldots, m_n \in \mathbb{N}^P$ such that $N : m \xrightarrow{t_1} m_1 \xrightarrow{t_2} \ldots m_{n-1} \xrightarrow{t_n} m_n$. We write $N : m \xrightarrow{*} m'$ if there exists a (possibly empty) firing sequence $\sigma \in T^*$ such that $N : m \xrightarrow{\sigma} m'$. The set of all reachable markings of a marked Petri net $(N, m)$ is denoted by $\mathcal{R}(N, m) = \{m' \in \mathbb{N}^P \mid N : m \xrightarrow{*} m'\}$.

A place $p \in P$ is *safe* if for any marking $m' \in \mathcal{R}(N, m)$, holds that $m'(p) \leq 1$. A marked Petri net is called safe if all its places are safe. The marking $m$ is a *deadlock* if there exists no transition $t \in T$ such that $^\bullet t \leq m$.

**Definition 4 (Workflow net, soundness).** *A Petri net $N = (P, T, F)$ is called a* workflow net *if (1) there exists exactly one place $i \in P$, called the* initial place*, such that $^\bullet i = \emptyset$, (2) there exists exactly one place, called the* final place*, $f \in P$ such that $f^\bullet = \emptyset$, and (3) all nodes are on a path from $i$ to $f$. $N$ is* sound *if $[f] \in \mathcal{R}(N, m)$ for any marking $m \in \mathcal{R}(N, [i])$.*

## 3 Open Petri Nets

The SOA paradigm builds upon asynchronous communications: a *component* sends messages to communicate with other components. In this approach, we model a component by a Petri net. Communication is done through an interface, modeled by input and output places. We call such a Petri net with an interface an *open Petri net* (OPN) [5,15,18].

**Definition 5 (Open Petri net).** *An* open Petri net *(OPN) is a 7-tuple $(P, I, O, T, F, i, f)$ where*

- *$(P \cup I \cup O, T, F)$ is a Petri net;*
- *$P$ is a set of* internal places*;*
- *$I$ is a set of* input places*, and $^\bullet I = \emptyset$;*
- *$O$ is a set of* output places*, and $O^\bullet = \emptyset$;*
- *$P$, $I$, $O$, and $T$ are pairwise disjoint;*
- *$i \in \mathbb{N}^P$ is the* initial marking*,*
- *$f \in \mathbb{N}^P$ is the* final marking*, and*
- *$f$ is a deadlock.*

*We call the set $I \cup O$ the* interface places *of the OPN. Two OPNs $N$ and $M$ are called* disjoint *if $P_N$, $P_M$, $I_N$, $I_M$, $O_N$, $O_M$, $T_N$ and $T_M$ are pairwise disjoint.*

Note that the initial and final markings cannot mark interface places. Although we allow interface places to have more than one connected transition, it is always possible to transform the nets to equivalent ones with exactly one connected transition for interface places (cf. [4]).

In order to inspect and refer to the internal working of a component, ignoring the communication aspects, we introduce the notion of a skeleton. The skeleton is the Petri net of the component without any interface places.

**Definition 6 (Skeleton).** *Let $N$ be an OPN. The* skeleton *of $N$ is defined as the Petri net $\mathcal{S}(N) = (P_N, T_N, F)$ with $F = F_N \cap ((P_N \times T_N) \cup (T_N \times P_N))$. We use $\mathcal{R}(N)$ as a shorthand notation for $\mathcal{R}(\mathcal{S}(N), i_N)$.*

The semantics of an OPN is given by its LTS.

**Definition 7 (LTS of an OPN).** *Let $N$ be an OPN. Its labeled transition system is defined as: $\mathcal{T}(N) = (\mathcal{R}(N), T, \longrightarrow, i_N, f_N)$ with $(m, t, m') \in \longrightarrow$ if and only if $\mathcal{S}(N) : m \xrightarrow{t} m'$.*

We focus on services that try to reach a goal. We therefore introduce the notion of an open workflow net, i.e., an open Petri net such that the skeleton is a workflow, and only the initial place is marked in the initial marking.

**Definition 8 (Open workflow net).** *Let $N$ be an OPN. It is called an* open workflow net *(OWN) iff its skeleton is a workflow net with initial place $i \in P$ and final place $f \in P$, such that ${}^\bullet i = \emptyset$, $f^\bullet = \emptyset$, $i_N = [i]$, $f_N = [f]$.*

Note that in an OWN $N$, the final marking $f_N$ is always a deadlock. All transitions and internal places lie on a path from $i$ to $f$, since the skeleton is a workflow, whereas interface places cannot not have this property. We would like services to be sound, i.e., always have the possibility to terminate properly. As the open Petri net has interface places, termination depends on the communication partners of the net. Still, we want to express that at least the service disregarding the communication is modelled in a proper way. Therefore, we define soundness on the skeleton of the open Petri net.

**Definition 9 (Soundness of OPNs).** *An OPN $N$ is called* sound *if for any marking $m \in \mathcal{R}(N)$ we have $\mathcal{S}(N) : m \xrightarrow{*} f_N$.*

Note that if an OPN $N$ is sound and $f_N$ is a nonempty deadlock, then the initial marking $i_N$ cannot be the empty marking, since if $f$ is reachable from the empty marking, also $2 \cdot f$ is reachable, but $f$ cannot be reached from $2 \cdot f$ thus invalidating soundness.

## 4 Composition

Two open Petri nets can be composed by fusing interface places with the same name.

**Definition 10 (Composition).** *Let $A$ and $B$ be two OPNs. Their* composition *is an OPN $A \oplus B = (P, I, O, T, F, i, f)$ defined by:*

- $P = P_A \cup P_B \cup (I_A \cap O_B) \cup (I_B \cap O_A)$;
- $I = (I_A \setminus O_B) \cup (I_B \setminus O_A)$;
- $O = (O_A \setminus I_B) \cup (O_B \setminus I_A)$;
- $T = T_A \cup T_B$;
- $F = F_A \cup F_B$;
- $i = i_A + i_B$;
- $f = f_A + f_B$.

Two OPNs are composable if they do not share any internal places, input places and output places. Note that an input place of one net can be an output place of another net.

**Definition 11 (Composable).** *Two OPNs $A$ and $B$ are* composable *if and only if $(P_A \cup I_A \cup O_A \cup T_A) \cap (P_B \cup I_B \cup O_B \cup T_B) = (I_A \cap O_B) \cup (O_A \cap I_B)$.*

The composition operator is commutative and associative for composable OPNs.

**Lemma 12 (Commutativity and associativity of composition).** *Let $A$, $B$ and $C$ be three pairwise composable OPNs. Then $A \oplus B = B \oplus A$ and $(A \oplus B) \oplus C = A \oplus (B \oplus C)$. In addition, if $\mathcal{O} = (\emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$, we have $A \oplus \mathcal{O} = \mathcal{O} \oplus A = A$.*

*Proof.* Follows directly from the definition of $\oplus$. □

Using the composition notion, we define a projection relation and simulation based on this relation.

**Definition 13 (Projection relation, simulation property).** *Let $A$ and $B$ be two composable OPNs. The* projection relation *$R \subseteq \mathbb{N}^{P_A} \times \mathbb{N}^{P_{A\oplus B}}$ is defined as $R = \{(m_{|P_A}, m) \mid m \in \mathcal{R}(A \oplus B)\}$. We use the shorthand notation $A \oplus B \succeq A$ for $\tau_{T_B}(\mathcal{T}(A \oplus B)) \succeq_R \mathcal{T}(A)$. If $A \oplus B \succeq A$, we say that the composition has the* simulation property*.*

Note that the notation $A \oplus B \succeq A$ states that the projection relation is a simulation. If $A \oplus B \succeq A$, then $A \oplus B$ is able to mimic the behavior of $A$ after abstracting away the transitions in $B$.

The next lemma shows that for a sequence $\sigma$ in $A \oplus B$ and a sequence $\tilde{\sigma}$ in $B \oplus C$, such that their projections on $B$ are identical, a composed sequence $\overline{\sigma}$ exists, such that the projection of $\overline{\sigma}$ on $A$ and $B$ is identical to $\sigma$, and the projection of $\overline{\sigma}$ on $B$ and $C$ is identical to $\tilde{\sigma}$.

**Lemma 14 (Combining firing sequences).** *Let $A$, $B$ and $C$ be three pairwise composable OPNs and let $A$ and $C$ have disjoint interface places. Let $m \in \mathcal{R}(A \oplus B \oplus C)$, $\sigma \in (T_A \cup T_B)^*$, $\tilde{\sigma} \in (T_B \cup T_C)^*$, $m' \in \mathcal{R}(A \oplus B)$ and $m'' \in \mathcal{R}(B \oplus C)$ such that $A \oplus B : m_{|P_{A\oplus B}} \xrightarrow{\sigma} m'$, $B \oplus C : m_{|P_{B\oplus C}} \xrightarrow{\tilde{\sigma}} m''$ and $\sigma_{|T_B} = \tilde{\sigma}_{|T_B}$. Then there exist a firing sequence $\overline{\sigma} \in (T_A \cup T_B \cup T_C)^*$ such that $\overline{\sigma}_{|T_A \cup T_B} = \sigma$ and $\overline{\sigma}_{|T_B \cup T_C} = \tilde{\sigma}$, and a marking $\overline{m} \in \mathcal{R}(A \oplus B \oplus C)$ such that $A \oplus B \oplus C : m \xrightarrow{\overline{\sigma}} \overline{m}$, $\overline{m}_{|P_{A\oplus B}} = m'$ and $\overline{m}_{|P_{B\oplus C}} = m''$.*

*Proof.* We prove this lemma by induction on the length of $\sigma_{|T_B}$. If $\sigma_{|T_B} = \epsilon$, then $\sigma$ can execute independently of $\tilde{\sigma}$, since transitions in $A$ and $C$ have no places in common. Hence, the statement holds for length 0.

Let it hold for length $n$ and we consider $m \in \mathcal{R}(A \oplus B \oplus C)$, $\sigma \in (T_A \cup T_B)^*$, $\tilde{\sigma} \in (T_B \cup T_C)^*$, $m' \in \mathcal{R}(A \oplus B)$ and $m'' \in \mathcal{R}(B \oplus C)$ such that $A \oplus B : m_{|P_{A\oplus B}} \xrightarrow{\sigma} m'$, $B \oplus C : m_{|P_{B\oplus C}} \xrightarrow{\tilde{\sigma}} m''$, with $\sigma_{|T_B} = \tilde{\sigma}_{|T_B} = \sigma'; b$ for some $b \in T_B$ and $|\sigma'| = n$. Then $\sigma = \sigma_1; b; \sigma_2$ for some $\sigma_1 \in (T_A \cup T_B)^*$, $\sigma_2 \in (T_A)^*$ and $\tilde{\sigma} = \tilde{\sigma}_1; b; \tilde{\sigma}_2$ for some $\tilde{\sigma}_1 \in (T_B \cup T_C)^*$ and $\tilde{\sigma}_2 \in (T_C)^*$ and $\sigma_{1|T_B} = \tilde{\sigma}_{1|T_B} = \sigma'_{|T_B}$. By the induction hypothesis there exists $\overline{\sigma}_1$ such that $\overline{\sigma}_{1|T_A \cup T_B} = \sigma_1$ and $\overline{\sigma}_{1|T_B \cup T_C} = \tilde{\sigma}_1$ and a marking $\overline{m}_1$ such that $A \oplus B \oplus C : m \xrightarrow{\overline{\sigma}_1} \overline{m}_1$, $\overline{m}_{1|P_{A\oplus B}} = m'_1$ and $\overline{m}_{1|P_{B\oplus C}} = m''_1$.

We have $A \oplus B \oplus C : \overline{m}_1 \xrightarrow{b} \overline{m}_2$, since $b$ is enabled both in $A \oplus B$ and in $B \oplus C$. In the resulting marking $\overline{m}_2$, $\sigma_2$ and $\tilde{\sigma}_2$ are independently enabled, since $A$ and $C$ have no common places. Hence, we can apply the induction hypothesis on $\sigma_2$ and $\tilde{\sigma}_2$ which gives a firing sequence $\overline{\sigma}_2$ such that $\overline{\sigma}_{2|T_A \cup T_B} = \sigma_2$ and $\overline{\sigma}_{2|T_B \cup T_C} = \tilde{\sigma}_2$ and a marking $\overline{m}$ such that $A \oplus B \oplus C : \overline{m}_2 \xrightarrow{\overline{\sigma}_2} \overline{m}$, $\overline{m}_{|P_{A\oplus B}} = m'$ and $\overline{m}_{|P_{B\oplus C}} = m''$. Hence, $\overline{\sigma} = \overline{\sigma}_1; b; \overline{\sigma}_2$ has the desired property. □

**Lemma 15.** *Let $A$ and $B$ be two composable OPNs. Let $m, m' \in \mathcal{R}(A \oplus B)$ and $\sigma \in (T_A \cup T_B)^*$ such that $A \oplus B : m \xrightarrow{\sigma} m'$. Then $A : m_{|P_A} \xrightarrow{\sigma_{|T_A}} m'_{|P_A}$ and $B : m_{|P_B} \xrightarrow{\sigma_{|T_B}} m'_{|P_B}$.*

Since a service tree is not known in advance, we need a condition such that if any two composed services satisfy this condition, the whole service tree is sound. The example in Figure 1 shows that soundness itself is not the right property, since although both $A \oplus B$ and $B \oplus C$ are sound, the composition of the three, $A \oplus B \oplus C$ is not. The reason is that $A \oplus B$ and $B \oplus C$ are acyclic but $A \oplus B \oplus C$ has a cycle which causes a deadlock. This example shows how tricky the asynchronous composition of OPNs is. We use Lemma 14 to come to a sufficient condition for soundness of the composition of three OPNs, by only pairwise checking of the connections between the OPNs.

If a component $B$ is composed with $A$, and this composition is sound, composing it with $C$ should not destroy soundness. This can occur e.g. if $C$ blocks some execution path of $B$. We will show that a sufficient condition for the soundness of $A \oplus B \oplus C$ is: C should not block any execution of $B$ that starts in the initial marking and ends in the final marking.

First, we formally define the sufficient condition.

**Definition 16 (Condition $\Omega_{A,B}$).** *Let $A$ and $B$ be two composable OPNs. Condition $\Omega_{A,B}$ holds if and only if $\forall m \in \mathcal{R}(A \oplus B)$, $\sigma \in (T_A)^* : (A : m_{|P_A} \xrightarrow{\sigma} f_A) \Rightarrow (\exists \tilde{\sigma} \in (T_A \cup T_B)^* : (A \oplus B : m \xrightarrow{\tilde{\sigma}} f_A + f_B) \wedge \tilde{\sigma}_{|T_A} = \sigma)$.*

Note that we do not require soundness of $A$ or $B$ here, but if $A$ is sound, $\Omega_{A,B}$ implies that $A \oplus B$ is sound.

**Lemma 17.** *Let $A$ and $B$ be two composable OPNs. If $A$ is sound and condition $\Omega_{A,B}$ holds, then $A \oplus B$ is sound.*

Now we show that condition $\Omega_{B,C}$ is a sufficient condition allowing to extend an arbitrary composition $A \oplus B$ by $C$ obtaining a sound $A \oplus B \oplus C$.

**Theorem 18 (Sufficient condition).** *Let $A$, $B$ and $C$ be three pairwise composable OPNs such that $A$ and $C$ have disjoint interface places. If the composition $A \oplus B$ is sound and $\Omega_{B,C}$ holds, then $A \oplus B \oplus C$ is sound.*

*Proof.* Let $m \in \mathcal{R}(A \oplus B \oplus C)$. Then, by Lemma 15, $m_{|P_{A\oplus B}} \in \mathcal{R}(A \oplus B)$ and $m_{|P_{B\oplus C}} \in \mathcal{R}(B \oplus C)$. By the soundness of $A \oplus B$ there is a $\sigma \in (T_{A\oplus B})^*$ such that $A \oplus B : m_{|P_{A\oplus B}} \xrightarrow{\sigma} f_A + f_B$. By condition $\Omega_{B,C}$ applied to $m_{|P_{B\oplus C}}$ and $\sigma_{|T_B}$, there is a $\tilde{\sigma} \in (T_B \cup T_C)^*$ such that $\tilde{\sigma}_{|T_B} = \sigma_{|T_B}$ and $B \oplus C : m_{|P_{B\oplus C}} \xrightarrow{\tilde{\sigma}} f_B + f_C$. By Lemma 14, there exist a firing sequence $\overline{\sigma} \in (T_A \cup T_B \cup T_C)^*$ and marking $\overline{m}$ such that $\overline{\sigma}_{|T_A \cup T_B} = \sigma$ and $\overline{\sigma}_{|T_B \cup T_C} = \tilde{\sigma}$ and $A \oplus B \oplus C : m \xrightarrow{\overline{\sigma}} \overline{m}$, $\overline{m}_{|P_{A\oplus B}} = f_A + f_B$ and $\overline{m}_{|P_{B\oplus C}} = f_B + f_C$. Note that the interface places between $A$ and $B$ and between $B$ and $C$ are empty. Since there are no interface places between $A$ and $C$, we have $\overline{m} = f_A + f_B + f_C$. $\square$

To facilitate the check of condition $\Omega_{A,B}$, we prove that $\Omega_{A,B}$ is equivalent to the condition "$A \oplus B$ simulates $A$".

**Theorem 19 (Equivalent condition).** *Let $A$ and $B$ be two composable OPNs and $A$ be sound. Then $\Omega_{A,B}$ holds if and only if $A \oplus B \succeq A$.*

*Proof.* ($\Rightarrow$) Assume that $\Omega_{A,B}$ holds. We show that the projection relation $R = \{(m_{|P_A}, m) \mid m \in \mathcal{R}(A \oplus B)\}$ is a simulation.

1) By definition of $\oplus$, we have $i_A \, R \, i_{A\oplus B}$ and $f_A \, R \, f_{A\oplus B}$ provided that $f_{A\oplus B} \in \mathcal{R}(A \oplus B)$. The latter condition follows from $\Omega_{A,B}$.

2) Let $\overline{m} \in \mathcal{R}(A \oplus B)$ and $m = \overline{m}_{|P_A}$, i.e., $m \, R \, \overline{m}$. Note that by Lemma 15, $m \in \mathcal{R}(A)$. Further let $A : m \xrightarrow{t} m'$ for some $m' \in \mathcal{R}(A)$, $t \in T_A$.

Since $A$ is sound, there exists a sequence $\sigma \in (T_A)^*$ such that $A : m \xrightarrow{t;\sigma} f_A$. By the condition $\Omega_{A,B}$, there is a sequence $\tilde{\sigma} \in (T_A \cup T_B)^*$, such that $A \oplus B : \overline{m} \xrightarrow{\tilde{\sigma}} f_A + f_B$ and $\tilde{\sigma}_{|T_A} = t; \sigma$. Thus, $A \oplus B : \overline{m} \xrightarrow{\tilde{\sigma}'} \overline{m}'' \xrightarrow{t} \overline{m}'$ (i.e. $A \oplus B : \overline{m} \overset{t}{\Longrightarrow} \overline{m}'$) for some markings $\overline{m}', \overline{m}'' \in \mathcal{R}(A \oplus B)$ and sequence $\tilde{\sigma}' \in (T_B)^*$. Since $\tilde{\sigma}' \in (T_B)^*$, for all places $p \in P_A$ holds $m(p) = \overline{m}(p) = \overline{m}''(p)$ and $m'(p) = m(p) - {}^{\bullet}t(p) + t^{\bullet}(p) = \overline{m}''(p) - {}^{\bullet}t(p) + t^{\bullet}(p) = \overline{m}'(p)$. Thus $\overline{m}'_{|P_A} = m'$, and therefore $m' \, R \, \overline{m}'$.

3) The only reachable deadlock possible in $A$ is $f_A$. Let $m \in \mathcal{R}(A \oplus B)$ such that $f_A \, R \, m$. No transition $t \in T_A$ is enabled in marking $f_A$, and hence, also not in $m$. By applying $\Omega_{A,B}$ to $\epsilon$ and $m$, we conclude that there exists a sequence $\tilde{\sigma} \in (T_A \cup T_B)^*$ such that $A \oplus B : m \xrightarrow{\tilde{\sigma}} f_A + f_B$. Since no transition of $A$ can fire, $\tilde{\sigma} \in (T_B)^*$. Hence $A \oplus B : m \Longrightarrow f_A + f_B$.

($\Leftarrow$) Assume $A \oplus B \succeq A$. Let $m \in \mathcal{R}(A \oplus B), \sigma \in (T_A)^*$ such that $A : m_{|P_A} \xrightarrow{\sigma} f_A$. We prove by induction on the length of $\sigma$ that $\exists \tilde{\sigma} \in (T_A \cup T_B)^* : (A \oplus B : m \xrightarrow{\tilde{\sigma}} f_A + f_B) \ \wedge \ \tilde{\sigma}_{|T_A} = \sigma$ to show that $\Omega_{A,B}$ holds.

Suppose $\sigma = \epsilon$, i.e. $m_{|P_A} \xrightarrow{\epsilon} f_A$, which implies $f_A = m_{|P_A}$ and thus $f_A \, R \, m$. Since $A \oplus B \succeq A$, there exists a sequence $\tilde{\sigma} \in (T_B)^*$ such that $A \oplus B : m \xrightarrow{\tilde{\sigma}} f_A + f_B$ with $\tilde{\sigma}_{|T_A} = \epsilon$.

Suppose $\sigma = t; \sigma'$, $t \in T_A$. Then, a marking $m' \in \mathcal{R}(A)$ exists, such that $A : m_{|P_A} \xrightarrow{t} m' \xrightarrow{\sigma'} f_A$. Since $A \oplus B \succeq A$, there exists a marking $\overline{m}' \in \mathcal{R}(A \oplus B)$ such that $A \oplus B : m \overset{t}{\Longrightarrow} \overline{m}'$ and $m' \, R \, \bar{m}'$. Hence, there is a marking $\overline{m}'' \in \mathcal{R}(A \oplus B)$ and a sequence $\tilde{\sigma}' \in (T_B)^*$ such that $A \oplus B : m \xrightarrow{\tilde{\sigma}'} \overline{m}'' \xrightarrow{t} \overline{m}'$. The induction hypothesis applied to $\sigma'$ implies the existence of a sequence $\tilde{\sigma}'' \in (T_A \cup T_B)^*$ such that $\tilde{\sigma}''_{|T_A} = \sigma'$ and $A \oplus B : \overline{m}' \xrightarrow{\tilde{\sigma}''} f_A + f_B$. Hence, $\tilde{\sigma} = \tilde{\sigma}'; t; \tilde{\sigma}''$ is a sequence such that $A \oplus B : m \xrightarrow{\tilde{\sigma}} f_A + f_B$ and $\tilde{\sigma}_{|T_A} = \sigma$. □

We can extend our results to compositions that are more complex than chains of three components. A *service tree* is a tree of components connected to each other such that the higher OPNs can only "subcontract" work to lower level OPNs. The structure of the tree is defined by the tree function $c$. Each node $i$ is an OPN representing a component that is delivering a service to its parent $c(i)$ using services of its children $c^{-1}(i)$. In the remainder of this section, we show that the sufficient condition is enough to only pairwise check the connections in the tree to decide whether the whole service tree is sound.

**Definition 20 (Service tree).** *Let $A_1, \ldots, A_n$ be pairwise composable OPNs. Let $c : \{2, \ldots, n\} \to \{1, \ldots, n-1\}$ be such that:*

- *$\forall i \in \{2, \ldots, n\} : c(i) < i$,*
- *$\forall 1 \le i < j \le n : i \ne c(j) \ \Rightarrow \ I_{A_i} \cap O_{A_j} = \emptyset \ \wedge \ O_{A_i} \cap I_{A_j} = \emptyset$, and*
- *$\forall 1 \le i < j \le n : i = c(j) \ \Rightarrow \ I_{A_i} \cap O_{A_j} \ne \emptyset \ \vee \ O_{A_i} \cap I_{A_j} \ne \emptyset$.*

*We call $A_1 \oplus \ldots \oplus A_n$ a* service tree *with root $A_1$ and tree function $c$.*

Lemma 17 together with Theorem 19 implies that if $B \oplus C \succeq B$ and $B$ is sound, the composition is sound as well. Hence, if we combine the results so far, we can show that if the root of a service tree is sound, and all the connections fulfill the $\Omega$ condition, the whole service tree is sound.

**Theorem 21 (Soundness of service trees).** *Let $A_1, \ldots A_n$ be a service tree with root $A_1$ and tree function $c$. Further, let $A_1$ be sound and for $2 \le i \le n$, it holds that $A_i \oplus A_{c(i)} \succeq A_{c(i)}$. Then $A_1 \oplus \ldots \oplus A_n$ is sound.*

*Proof.* We prove this by induction on $n$. If $n = 1$, it is true by definition. Suppose it is true for $n = k - 1$. Let $n = k$. By the induction hypothesis: $A_1 \oplus \ldots \oplus A_{k-1}$ is sound and always $c(k) < k$. By the associativity and commutativity of $\oplus$ we have that $(A_1 \oplus \ldots \oplus A_{c(k)-1} \oplus A_{c(k)+1} \oplus \ldots \oplus A_{k-1}) \oplus A_{c(k)}$ is sound. We also have $A_k \oplus A_{c(k)} \succeq A_{c(k)}$. By Lemma 17 we have $(A_1 \oplus \ldots \oplus A_{c(k)-1} \oplus A_{c(k)+1} \oplus \ldots \oplus A_{k-1}) \oplus A_{c(k)} \oplus A_k$ is sound. Again by associativity and commutativity, the theorem is proven. □

## 5 Stepwise Refinement

In the previous section we showed that we can build sound service trees compositionally. In this section we present a construction method that guarantees condition $\Omega$ with simultaneous refinements of places in communicating components with communicating subcomponents.

In [12], the authors introduce a notion of place refinement where a place is refined by a workflow net. We define this refinement operation on open Petri nets, and refine internal places by communicating OWNs.

The choice for OWNs with a single initial and a single final place makes the refinement definition natural. The refinement is only defined for the nets that do not overlap, except for (possibly) interface places. In some situations we will additionally require that input places of $A$ may not be output of $B$ or vice-versa, to prevent breaking the composability of $A$ with other components. These requirements are captured in the following definition.

**Definition 22 (Refinable).** *Let $A$ be an OPN, and let $B$ be an OWN. $A$* is refinable by *$B$ if $(P_A \cup I_A \cup O_A \cup T_A) \cap (P_B \cup I_B \cup O_B \cup T_B) = (I_A \cup O_A) \cap (I_B \cup O_B)$.*

*$A$* is strictly refinable by *$B$ if $(P_A \cup I_A \cup O_A \cup T_A) \cap (P_B \cup I_B \cup O_B \cup T_B) = (I_A \cap I_B) \cup (O_A \cap O_B)$.*

When we refine an arbitrary place $p \in P_A$ by an OWN $B$, all transitions of the refined net that produced a token in $p$ now produce a token in the initial place of $B$, and all transitions that consumed a token from $p$ now consume a token from the final place of $B$. If the two nets share interface places, these places are fused.

**Definition 23 (Place refinement).** *Let $A$ be an OPN, and let $B$ be an OWN such that $A$ is refinable by $B$. Let $p \in P_A$ such that $i_A(p) = f_A(p) = 0$. The refined OPN $A \odot_p B = (P, I, O, T, F, i, f)$ is defined as:*

- *$P = (P_A \setminus \{p\}) \cup P_B \cup (I_A \cap O_B) \cup (I_B \cap O_A)$;*
- *$I = (I_A \setminus O_B) \cup (I_B \setminus O_A)$;*
- *$O = (O_A \setminus I_B) \cup (O_B \setminus I_A)$;*
- *$T = T_A \cup T_B$;*
- *$F = F_A \setminus ((^\bullet p \times \{p\}) \cup (\{p\} \times p^\bullet)) \cup F_B \cup (^\bullet p \times \{i_B\}) \cup (\{f_B\} \times p^\bullet)$;*
- *$i = i_A$;*
- *$f = f_A$.*

Note that in case an input place of one net is an output place of another net, this place becomes an internal place of the resulting net.

If in the original net two places are refined, the resulting net does not depend on the order in which the refinements took place. Also, the refinement distributes over the composition.

**Lemma 24.** *Let $N$ be an OPN, let $p, q \in P_N$ and $p \neq q$. Let $C$ and $D$ be two disjoint OWNs such that $N$ is strictly refinable by $C$ and $N$ is strictly refinable by $D$. Then $(N \odot_p C) \odot_q D = (N \odot_q D) \odot_p C$. Furthermore, if $N = A \oplus B$ for some OPNs $A$ and $B$, $p \in P_A$ and $q \in P_B$, we have $((A \oplus B) \odot_p C) \odot_q D = (A \odot_p C) \oplus (B \odot_q D)$.*

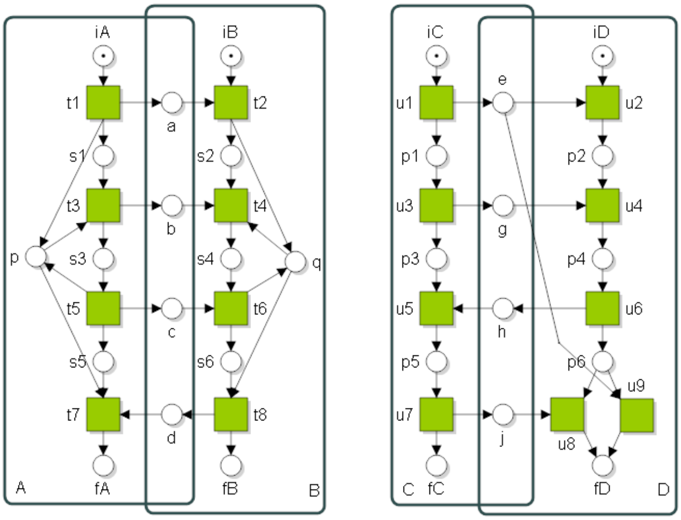


**Fig. 2.** Places $p$ and $q$ are safe, $A \oplus B \succeq A$ and $C \oplus D \succeq C$, but $(A \odot_p C) \oplus (B \odot_q D)$ is not sound

The following statement is a generalization of Theorem 10 from [12] to the case of OPN nets.

**Lemma 25 (Soundness of refinement).** *Let $A$ be a safe and sound OPN, and let $B$ be a sound OWN such that $A$ is strictly refinable by $B$. Let $p \in P_A$. Then the refinement $A \odot_p B$ is sound.*

When constructing services, we might want to refine two places of communicating components with a sound composition of two subcomponents. Refinement of a sound and safe OPN by a sound OWN results in a sound OPN, but refining two places in a sound and safe OPN by a sound composition is in general not sound. An intuitive approach would be to apply this refinement only to "synchronizable" places — such places that if one of the places becomes marked in the execution sequence, then another one already has a token, or will receive it before the token disappears from the first one. A counterexample for this idea is given in Figure 2. Both compositions $A \oplus B$ and $C \oplus D$ are safe and sound and places $p$ and $q$ are "synchronizable", but the composition $(A \odot_p C) \oplus (B \odot_q D)$, i.e. place $p$ is refined by $C$ and place $q$ by $D$, is not sound, which is caused by the fact that $C$ can be *started for the second time before D has finished.* Consider for example the firing sequence $\sigma = \langle t1, u1, u3, t2, u2, u4, u6, u5, u7, t3, t5, u1\rangle$. We have $(A \odot_p C) \oplus (B \odot_q D) : [i_A, i_B] \xrightarrow{\sigma} [s5, b, c, s2, p1, e, j, p6]$. Continuing with the firing sequence $\gamma = \langle u9, u3, t4, t6\rangle$ we obtain marking $[s5, s6, p3, g, j, i_D]$, which is a deadlock. This scenario was not possible in $C \oplus D$ itself.

To solve this problem, we need to ensure a stricter synchronization for places $p$ and $q$, namely that place $q$ only becomes marked if $p$ is marked, and that $p$ can only become unmarked, after $q$ became unmarked. A structure that guarantees

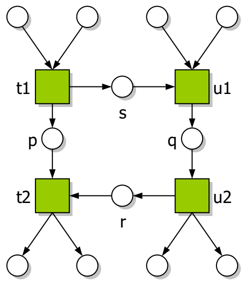


**Fig. 3.** Block structure ensuring that $p$ is synchronized with $q$ ($p \rightleftharpoons_N q$)

this condition is the *block structure* (see Figure 5), which is a handshaking protocol providing the notifications when a place becomes marked or unmarked. If place $p$ becomes marked, this is notified by sending a message. This message is consumed when $q$ becomes marked. As soon as $q$ becomes unmarked, a message is sent, after which $p$ can become unmarked.

**Definition 26 (Synchronized places, $p \rightleftharpoons_N q$).** *Let $N$ be an OPN and $p, q \in P_N$ two distinct places. We say that $p$ is* synchronized *with $q$ in $N$, denoted by $p \rightleftharpoons_N q$, if and only if $p$ and $q$ are safe, $i_N(p) = i_N(q) = f_N(p) = f_N(q) = 0$, and there exist two* communication places *$r, s \in P_N$ and four transitions $t_1, t_2, u_1, u_2 \in T_N$ that together form a so-called* block structure *where:*

– $^\bullet p = \{t_1\} \wedge p^\bullet = \{t_2\} \wedge {}^\bullet q = \{u_1\} \wedge q^\bullet = \{u_2\}$;
– $t_1{}^\bullet = \{p, s\} \wedge {}^\bullet t_2 = \{p, r\} \wedge u_1{}^\bullet = \{q\} \wedge {}^\bullet u_2 = \{q\}$;
– $^\bullet s = \{t_1\} \wedge s^\bullet = \{u_1\} \wedge {}^\bullet r = \{u_2\} \wedge r^\bullet = \{t_2\}$.

The block structure guarantees that if place $p$ becomes marked, it cannot become unmarked before $q$ has been marked. This is ensured via two communication places, place $s$ and place $r$. A token in place $s$ indicates that a token is put in $p$ but not yet in $q$, whereas a token in place $r$ indicates that a token has been consumed from place $q$. I.e., a transition that produces a token in $p$ should also produce a token in place $s$, a transition that consumes from $s$ should place a token in $q$. If a transition consumes from place $q$, it should produce a token in $r$, which can only be consumed by a transition that also consumes from $p$. Note that the block structure is asymmetric, although reversing the direction of the messages will have small local influence on the behaviour when $p$ and $q$ get refined with communicating subcomponents.

It is easy to show that $p = q + r + s$ is a place invariant, i.e., the structure guarantees that the number of tokens in $p$ equals the number of tokens in $q$, $r$, and $s$.

**Lemma 27.** *Let $N$ be an OPN. Let $p, q \in P_N$ such that $p \rightleftharpoons_N q$. Let $s, r \in P_N$ be the communication places of the block structure of $p$ and $q$. Then $p = q + s + r$ is a place invariant.*

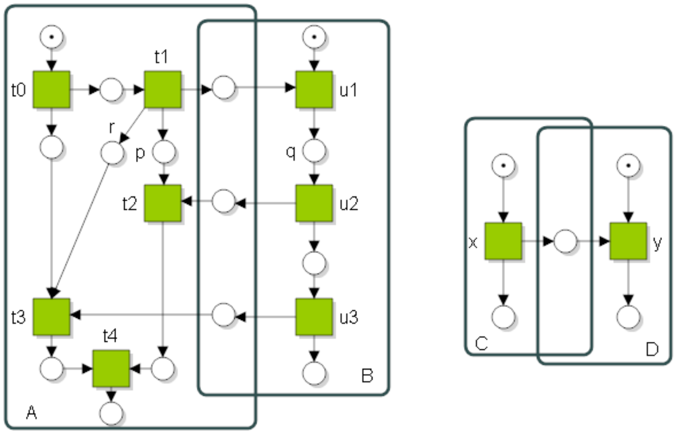


**Fig. 4.** Places $p$ and $q$ are not synchronized

This invariant implies that if the net is sound, the block structure of two synchronized places indeed guarantees the desired property: if $p$ is marked, $q$ is already marked, or it will become marked before the token is gone from $p$.

**Corollary 28.** *Let $N$ be a sound OPN, $p, q \in P_N$ such that $p \rightleftharpoons_N q$ and $m \in \mathcal{R}(N)$ such that $m(p) = 1$. Then there exist a firing sequence $\sigma \in (T_N)^*$ and a marking $m' \in \mathcal{R}(N)$ such that $N : m \xrightarrow{\sigma} m'$ and $m'(p) = m'(r) = 1$ (i.e. $\sigma$ enables $t_2$). Moreover, if $N = A \oplus B$ for some composable OPNs $A$ and $B$ with $A \oplus B \succeq A$, $p \in P_A$ and $q \in P_B$, then there exists such a sequence $\sigma$ with just transitions of $B$ (i.e., $\sigma \in (T_B)^*$).*

In Figure 4, we consider a slight extension of the block structure by allowing more output places for transition $t_1$. This extension does not work. First note that $A$, $B$ and $A \oplus B$ are all sound and $A \oplus B \succeq A$. However, if we refine places $p$ and $q$ with the very simple net $C \oplus D$, we loose the simulation property for the whole system. In $A \odot_p C$ we can have the firing sequence $\langle t_0, t_1, t_3 \rangle$ while in the whole system, $(A \odot_p C) \oplus (B \odot_q D)$, this firing sequence cannot be simulated, since the firing of transition $x$ is needed in order to put a token on the third input place of $t_3$ in $(A \odot_p C) \oplus (B \odot_q D)$.

As in an OPN only a safe place can be refined by a sound OWN, we show that if a net is safe, and we refine two places by a safe composition of two OWNs, then the refined net is safe again.

**Theorem 29 (Refinement preserves safety).** *Let $N$ be a safe OPN, let $p, q \in P_N$ such that $p \rightleftharpoons_N q$. Let $C$ and $D$ be two composable OWNs such that $N$ is strictly refinable by $C$ and $N$ is strictly refinable by $D$, and $C \oplus D$ is safe. Then the refinement $N' = (N \odot_p C) \odot_q D$ is safe.*

*Proof.* Since $N$ is safe, $p$ and $q$ are never marked with two or more tokens. Hence, if $m(p) > 0$, transition $t_2 \in p\overset{N}{\bullet}$ fires before transition $t_1 \in \overset{N}{\bullet}p$ can fire again. The same holds for $u_1$ and $u_2$ because of the block structure. Therefore, $C \oplus D$ can never be restarted before it is finished. Hence, since both $N$ and $C \oplus D$ are safe, $N'$ is safe. $\square$

We use Corollary 28 to show that if in a sound OPN $N$ two synchronized places are refined by a sound composition, then the refined OPN is sound as well.

**Theorem 30 (Refinement preserves soundness).** *Let $N$ be a sound OPN with $p, q \in P_N$ such that $p \rightleftharpoons_N q$. Let $C$ and $D$ be composable OWNs such that $C \oplus D$ is sound and $N$ strictly refinable by $C$ and $N$ strictly refinable by $D$. Then $N' = (N \odot_p C) \odot_q D$ is sound.*

*Proof.* (Idea) We map an arbitrary token marking $m' \in \mathcal{R}(N')$ to the marking $m$ of $N$ by putting a token on $p$ when $C$ is marked, a token on $q$ when $D$ is marked and keeping the rest of the marking as is; note that $m$ is reachable in $N$. Due to the soundness of $N$, $m \xrightarrow{\sigma} f_N$ for some $\sigma \in T_N^*$. Then we use the fact that $p$ and $q$ are safe and synchronized, and $C \oplus D$ is sound to fill in $\sigma$ up to $\sigma'$ such that $m' \xrightarrow{\sigma'} f_{N'}$. $\square$

The next theorem exploits the composition structure. If $A \oplus B$ simulates $A$ and $C \oplus D$ simulates $C$, then $((A \odot_p C) \oplus (B \odot_q D))$ simulates $A \odot_p C$.

**Theorem 31 (Refinement preserves simulation).** *Let $A$ and $B$ be two composable OPNs such that $N = A \oplus B$, $N \succeq A$, and $A$ is sound. Let $p \in P_A$ and $q \in P_B$ such that $p \rightleftharpoons_N q$. Let $C$ and $D$ be two composable OWNs such that $N$ is strictly refinable by $C$ and $N$ is strictly refinable by $D$, $C \oplus D \succeq C$ and $C$ is sound. Let $A' = A \odot_p C$ and $B' = (B \odot_q D)$. Then $A' \oplus B' \succeq A'$.*

*Proof.* (Sketch) Let $N' = A' \oplus B'$. We prove that $R = \{(m_{|P_{A'}}, m) \mid m \in \mathcal{R}(N')\}$ is a simulation, assuming all transitions of $B'$ are $\tau$-labeled. Let $m, m' \in \mathcal{R}(A')$, $t \in T_{A'}$ and $\overline{m} \in \mathcal{R}(N')$. Then either $t \in T_A$ or $t \in T_C$. Suppose $t \in T_A$. Then either (1) $C$ and $D$ do not contain any marked place in $m$, or (2) there is at least one place marked in $C$. In the first case, the firing of $t$ does not depend on the marking in $p$. $A \oplus B \succeq A$ implies the existence of a marking $\overline{m}' \in \mathcal{R}(N')$ such that $N' \colon \overline{m} \overset{t}{\Longrightarrow} \overline{m}'$ and $m' \, R \, \overline{m}'$. In the second case, we need to consider two subcases: either $p$ is in the preset of $t$ in $A$, or not. If $p$ is not in the preset, the argument is similar to case (1). In case $p$ is in the preset of $t$, we have $t = t_2$. If there is a place marked in $D$, $C \oplus D \succeq C$ implies that a marking $\overline{m}'' \in \mathcal{R}(N')$ enabling transition $t$ can by reached by the firings of transitions of $B$ and $D$ only. Hence, there exists a marking $\overline{m}' \in \mathcal{R}(N')$ such that $N' : \overline{m} \overset{t}{\Longrightarrow} \overline{m}'$ and $m' \, R \, \overline{m}'$. If there is no place marked in $D$, then by Corollary 28, either $t$ is enabled in $\overline{m}'$, or there exists a firing sequence in $B$ marking a place in $D$. In both cases we can conclude the existence of a marking $\overline{m}' \in \mathcal{R}(N')$ such that $N' : \overline{m} \overset{t}{\Longrightarrow} \overline{m}'$ and $m' \, R \, \overline{m}'$.

Suppose $t \in T_C$. Then either (1) there exists a place in $D$ that is marked in $\overline{m}$ or (2) no place in $D$ is marked in $\overline{m}$. In the first case, $C \oplus D \succeq C$ implies the existence of a marking $\overline{m}'$ such that $N' : \overline{m} \overset{t}{\Longrightarrow} \overline{m}'$ and $m' \, R \, \overline{m}'$. If there is no place in $D$ that is marked in $\overline{m}$, then by Corollary 28, either there is a firing sequence in $B$ that marks a place in $D$ or $D$ has already finished. In the first case, a marking $\overline{m}'' \in \mathcal{R}(N')$ with $N'\overline{m} \Longrightarrow \overline{m}''$ is reached, marking a place in $D$, hence we can apply case (1) to marking $\overline{m}''$. In the second case, $D$ has already produced all necessary tokens for $C$, thus $t \leq \overline{m}$. In both cases we can conclude the existence of a marking $\overline{m}' \in \mathcal{R}(N')$ such that $N' : \overline{m} \overset{t}{\Longrightarrow} \overline{m}'$ and $m' \, R \, \overline{m}'$. □

Theorems 21 and 31 imply compositionality of our construction method with respect to soundness:

**Corollary 32.** *Let $A$ and $B$ be two composable OPNs such that $N = A \oplus B$, $N \succeq A$, and $A$ sound. Let $p \in P_A$ and $q \in P_B$ such that $p \rightleftharpoons_N q$. Let $C$ and $D$ be two composable OWNs such that $N$ is strictly refinable by $C$ and $N$ is strictly refinable by $D$, $C \oplus D \succeq C$, and $C$ sound. Then $((A \odot_p C) \oplus (B \odot_q D))$ is sound.*

## 6 Construction of Service Trees

In Section 4 we defined a sufficient condition for the soundness of service trees and showed that this condition is equivalent to the simulation property. In Section 5, we showed that if two places in a composition are synchronized, they can be refined by a composition, such that the refined net is sound again. In this section, we combine the results obtained so far to construct service trees in a soundness-by-construction fashion. We show two examples of possible approaches, one for OWNs and another one for a specific subclass of OPNs.

The first approach is inspired by the concept of outsourcing. Consider a sound OWN $N$. Suppose some place $x \in P_N$ is not just a state marker, but it represents the execution of an activity outside the scope of the OWN, e.g. the place is called "producing an item". Now suppose there is a service that produces this "item". Then we can "outsource" the activity to the service we found. We modify the net, and refine the place by a start transition and an end transition, indicating the start and end of the activity. The start transition initiates the service, and as soon as the service finishes, the end transition is triggered.

Consider Figure 5, where place $x$ represents "producing an item" and we assume that it is safe. By Lemma 25, we can refine place $x$ by a sound workflow net so that the refined net is sound. Consider the OWN $M_1 = (\{i_M, p, f_M\}, \{r\}, \{s\}, \{t_1, t_2\}, \{(i_M, t_1), (t_1, p), (t_1, s), (p, t_2), (r, t_2), (t_2, f_M)\}, [i_M], [f_M])$. The refinement $N \odot_x M_1$ is sound, and, since place $x$ is safe, place $p$ is safe. Now, consider the OPN $M_2 = (\{q\}, \{s\}, \{r\}, \{t_3, t_4\}, \{(s, t_3), (t_3, q), (q, t_4), (t_4, r)\}, \emptyset, \emptyset)$. Although we need to drop the requirement that the final marking is a deadlock, it is easy to show that the net $(N \odot_x M_1) \oplus M_2$ is sound and so is $(N \odot_x M_1) \oplus M_2 \succeq (N \odot_x M_1)$. By this composition, we introduced a block structure around places $p$ and $q$ (see Figure 5). By Lemma 27 and the safety of $p$, we know that $q$ is safe, and thus $p \rightleftharpoons_{(N \odot_x M_1) \oplus M_2} q$. Now we can refine places $p$ and $q$ by any sound composition of OWNs. This way, we can construct an arbitrary large sound service tree, without the need to check condition $\Omega_{A,B}$ for any pair of composed OPNs $A$ and $B$.

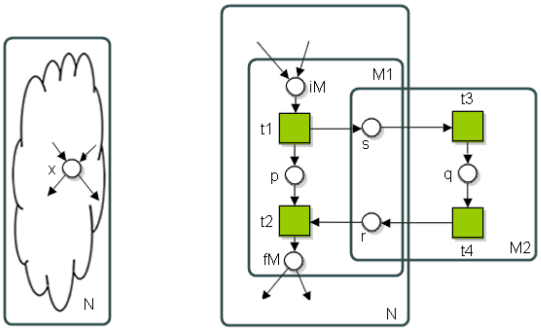


**Fig. 5.** The "outsourcing" method: place $x$ is refined by $M_1$

**Theorem 33.** *Let $N$ be an arbitrary OPN and $x \in P_N$. Consider the two OPNs $M_1$ and $M_2$ from Figure 5. Let $N' = N \odot_x (M_1 \oplus M_2) = (N \odot_x M_1) \oplus M_2$. Then:*

1. *if $N$ is safe, then $N'$ is safe;*
2. *if $N$ is sound, then $N'$ is sound;*
3. *$N' \succeq (N \odot_x M_1)$;*
4. *if $N$ is an OWN, then $N'$ is also an OWN.*

*Proof.* This follows immediately from the structure of $M_1$ and $M_2$. □

A second approach is based on pairs of composed OPNs belonging to a special subclass $\mathcal{N}$ of OPNs. This class is recursively defined, starting with two composed OPNs that are either both acyclic T-nets[1] or both isomorphic S-nets[2] with some additional requirements on their composition. The class $\mathcal{N}$ is defined in such a way that any pair of OPNs in this class is safe, sound and has the simulation property. In these pairs of composed OPNs we can easily detect the synchronized places and then each pair of these can be refined with nets of $\mathcal{N}$, resulting in an element that is again in $\mathcal{N}$. For details, see [4].

As an example, consider the approach presented in [13]. All services have the same protocol (OPN) for bargaining about the outsourcing of a service. Internally each service has its OWN orchestration process of the service. A task of this orchestration is modeled by a start transition, an end transition and a place in between like in Figure 5. The whole orchestration is modeled as an OWN. Some tasks may be outsourced to another service. The next step is refinement of the synchronized places by the standard bargaining protocol, which is in fact a sound composition of two OWNs, one for the service client and one for the service provider. In the OWN of the service provider there is one place that represents the execution of the service. This place may be refined by an arbitrary sound OWN which represents the orchestration of the service and by Lemma 25 this conserves the soundness of the whole system. Now we may select a task in this

[1] A Petri net in which all places have maximal one input and output transition.
[2] A Petri net in which all transitions have maximal one input and one output place.

orchestration process and we may repeat the outsourcing process. Hence, we build up a service tree as an OWN.

Another example is that we have three parties with interaction between all three. For instance a Buyer invokes a service at a Seller and the Seller invokes a service at a Shipper which is the direct delivery of the goods to the Buyer. Now, the question is: "does this fit into the framework?". In fact the Seller process will wait in some place $q$ until the Shipper has delivered the goods at the Buyer and then the Shipper will notify the Seller by sending a token to a transition $t$ in the Seller, where $t$ is an output transition of $q$. We may refine $q$ by a OWN that only intercepts and passes the communication between Shipper and Buyer and thus all interaction between the Shipper and the Buyer is now via the Seller. This reflects the responsibility: the Seller is in fact responsible for the delivery. Hence, now it is a tree structure again and it fits into our framework.

## 7 Conclusions

In this paper, we presented a method for compositional verification of a global termination property of an arbitrary tree of communicating components, where only checks of pairs of directly linked components are required. Another dimension where our construction goes is place refinements. Note that our method can easily be extended to the simultaneous refinement of several (more than two) places in communicating components with communicating subcomponents. Finally, we gave a method to construct such a tree in a correctness-by construction fashion, based on the composition and refinement.

**Related Work.** In [10] the authors give a constructive method preserving the inheritance of behavior. As shown in [2] this can be used to guarantee the correctness of interorganizational processes. Other formalisms, like I/O automata [17] or interface automata [7] use synchronous communication, whereas we focus on asynchronous communication, which is essential for our application domain, since the communication in SOA is asynchronous.

In [20], the author introduces place composition to model asynchronous communication focusing on the question which subnets can be exchanged such that the behavior of the whole net is preserved. Open Petri Nets are very similar to the concept of Petri net components, see e.g. [14], in which a module consists of a Petri net and interface places to be connected to other components. Open Petri nets were introduced and studied in [5, 6, 15, 16, 18]. In [16] the authors focus on deciding *controllability* of an OPN and computing its *operating guidelines.* Operating guidelines can be used to decide substitutability of services [19], or to prove that an implementation of a service meets its specification [6].

The major advantage of our approach compared to the operating guideline approach is its compositionality. In our setting it is sufficient to analyze only directly connected services of the tree, while the overall operating guidelines of the tree would have to be re-computed before a new service can be checked for a harmless addition to the tree. Moreover, the construction of the service tree remains flexible — any component can be replaced by another component, provided that the composition of this component with its direct neighbors satisfies our condition. This flexibility comes however with a price label, namely, the condition we define is a sufficient but not necessary condition, i.e. we might not be able to approve some service trees, although they were sound.

In [11], the authors propose to model choreographies using Interaction Petri nets, which is a special class of Petri nets, where transitions are labeled with the source and target component, and the message type being sent. To check whether the composition is functioning correctly, the whole network of components needs to be checked, whereas in our approach the check is done compositionally.

**Future Work.** The sufficient condition provided in Section 4 requires that an OPN $B$ does not restrict the behavior of $A$ in the composition $A \oplus B$. This condition might be relaxed by requiring that $A \oplus B$ mimics all *visible* behavior. The main research question here is defining a set of visible actions so that the approach would remain compositional and this set would be as small as possible. Note that such a relaxation will not influence the framework.

In Section 6, we have shown how the obtained results can be used to build service trees that are sound by construction. Although we only apply our results on Petri nets, our method can be extended to languages like BPEL, to facilitate the construction of web services in development environments like Oracle BPEL or IBM Websphere.

## References

1. van der Aalst, W.M.P.: The Application of Petri Nets to Workflow Management. The Journal of Circuits, Systems and Computers 8(1), 21–66 (1998)
2. van der Aalst, W.M.P.: Inheritance of Interorganizational Workflows: How to agree to disagree without loosing control? Information Technology and Management Journal 4(4), 345–389 (2003)
3. van der Aalst, W.M.P., Beisiegel, M., van Hee, K.M., König, D., Stahl, C.: An SOA-Based Architecture Framework. International Journal of Business Process Integration and Management 2(2), 91–101 (2007)
4. van der Aalst, W.M.P., van Hee, K.M., Massuthe, P., Sidorova, N., van der Werf, J.M.E.M.: Compositional service trees. Technical Report CSR 09/01, Technische Universiteit Eindhoven (January 2009)
5. van der Aalst, W.M.P., Lohmann, N., Massuthe, P., Stahl, C., Wolf, K.: From Public Views to Private Views: Correctness-by-Design for Services. In: Dumas, M., Heckel, R. (eds.) WS-FM 2007. LNCS, vol. 4937, pp. 139–153. Springer, Heidelberg (2008)
6. van der Aalst, W.M.P., Lohmann, N., Massuthe, P., Stahl, C., Wolf, K.: Multiparty Contracts: Agreeing and Implementing Interorganizational Processes. The Computer Journal (2009) (accepted for publication)
7. de Alfaro, L., Henzinger, T.A.: Interface automata. SIGSOFT Softw. Eng. Notes 26(5), 109–120 (2001)
8. Alonso, G., Casati, F., Kuno, H., Machiraju, V.: Web Services Concepts, Architectures and Applications. Springer, Heidelberg (2004)

9. Alves, A., Arkin, A., Askary, S., et al.: Web Services Business Process Execution Language Version 2.0 (OASIS Standard). WS-BPEL TC OASIS (2007), `http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html`
10. Basten, T., van der Aalst, W.M.P.: Inheritance of Behavior. Journal of Logic and Algebraic Programming 47(2), 47–145 (2001)
11. Decker, G., Weske, M.: Local enforceability in interaction petri nets. In: Alonso, G., Dadam, P., Rosemann, M. (eds.) BPM 2007. LNCS, vol. 4714, pp. 305–319. Springer, Heidelberg (2007)
12. van Hee, K.M., Sidorova, N., Voorhoeve, M.: Soundness and Separability of Workflow Nets in the Stepwise Refinement Approach. In: van der Aalst, W.M.P., Best, E. (eds.) ICATPN 2003. LNCS, vol. 2679, pp. 335–354. Springer, Heidelberg (2003)
13. van Hee, K.M., Verbeek, H.M.W., Stahl, C., Sidorova, N.: A framework for linking and pricing no-cure-no-pay services. Transactions on Petri Nets and Other Models of Concurrency (to appear, 2009)
14. Kindler, E.: A compositional partial order semantics for Petri net components. In: Azéma, P., Balbo, G. (eds.) ICATPN 1997. LNCS, vol. 1248, pp. 235–252. Springer, Heidelberg (1997)
15. Lohmann, N., Massuthe, P., Stahl, C., Weinberg, D.: Analyzing Interacting BPEL Processes. In: Dustdar, S., Fiadeiro, J.L., Sheth, A.P. (eds.) BPM 2006. LNCS, vol. 4102, pp. 17–32. Springer, Heidelberg (2006)
16. Lohmann, N., Massuthe, P., Wolf, K.: Operating Guidelines for Finite-State Services. In: Kleijn, J., Yakovlev, A. (eds.) ICATPN 2007. LNCS, vol. 4546, pp. 321–341. Springer, Heidelberg (2007)
17. Lynch, N.A., Tuttle, M.: Hierarchical correctness proofs for distributed algorithms. In: 6th Annual ACM Symposium on Principles of Distributed Computing (1987)
18. Massuthe, P., Reisig, W., Schmidt, K.: An Operating Guideline Approach to the SOA. Annals of Mathematics, Computing & Teleinformatics 1(3), 35–43 (2005)
19. Stahl, C., Massuthe, P., Bretschneider, J.: Deciding Substitutability of Services with Operating Guidelines. Transactions on Petri Nets and Other Models of Concurrency (2008) (accepted for publication)
20. Vogler, W.: Asynchronous communication of petri nets and the refinement of transitions. In: Kuich, W. (ed.) ICALP 1992. LNCS, vol. 623, pp. 605–616. Springer, Heidelberg (1992)

# ASAP: An Extensible Platform for State Space Analysis[*]

Michael Westergaard[1], Sami Evangelista[1], and Lars Michael Kristensen[2]

[1] Department of Computer Science, Aarhus University, Denmark
mw@cs.au.dk, evangeli@cs.au.dk

[2] Department of Computer Engineering, Bergen University College, Norway
lmkr@hib.no

**Abstract.** The ASCoVeCo State space Analysis Platform (ASAP) is a tool for performing explicit state space analysis of coloured Petri nets (CPNs) and other formalisms. ASAP supports a wide range of state space reduction techniques and is intended to be easy to extend and to use, making it a suitable tool for students, researchers, and industrial users that would like to analyze protocols and/or experiment with different algorithms. This paper presents ASAP from these two perspectives.

## 1 Introduction

State space analysis (or model checking) is one of the main approaches to model-based verification of concurrent systems and is one of the most successfully applied analysis methods for formal models. Its main limitation is the *state explosion problem*, i.e., that state spaces of systems may have a large number of reachable states, meaning that they are too large to be handled with the available computing power (CPU speed and memory). Methods for alleviating this inherent complexity problem is an active area of research and has led to the development of a large collection of *state space reduction methods*. These methods have significantly broadened the class of systems that can be verified and state spaces can now be used to verify systems of industrial size. A computer tool supporting state space analysis must implement a wide range of reduction algorithms since no single method works well on all systems. The software architectures of many such tools, e.g., CPN Tools [5], SPIN [1], make it difficult to support a collection of state space reduction methods in a coherent manner and to extend the tools.

This paper presents the ASCoVeCo State Space Analysis Platform (ASAP) [2] which is currently being developed in the context of the ASCoVeCo research project [3]. ASAP represents the next generation of tool support for state space exploration and analysis of CPN models [13] and other formal models. The aim and vision of ASAP is to provide an open platform suited for research, educational, and industrial use of state space exploration. This means that ASAP

[*] Supported by the Danish Research Council for Technology and Production.

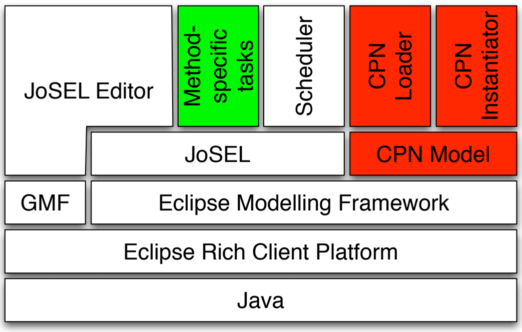
supports a wide collection of state space exploration methods and has an architecture that allows the research community to extend the set of supported methods. Furthermore, we aim at making ASAP sufficiently mature to be used for educational purposes, including teaching of advanced state space methods, and for use in industrial projects as has been the case with CPN Tools and Design/CPN [6].

This paper is structured as follows. The next section is an overview of the architecture of our tool. Section 3 briefly describes how the tool can be extended with new verification algorithms or modeling languages. A few benchmarks comparing our tool with CPN Tools and DiVinE [7] are presented in Section 4. Section 5 concludes this work and presents some future extensions to our tool.

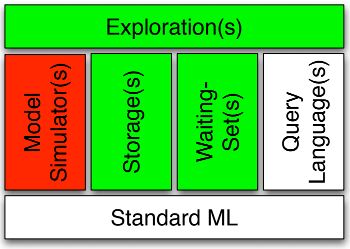
## 2 Architecture of ASAP

The ASAP platform consists of a graphical user interface (GUI) and a state space exploration engine (SSE engine). Figure 1(a) shows the software architecture of the graphical user interface which is implemented in Java based on the Eclipse Rich Client Platform [8]. The software architecture of the SSE engine is shown in Fig. 1(b). It is based on Standard ML and implements the state space Exploration and model checking algorithms supported by ASAP. The choice of SML for the SSE engine is primarily motivated by its ability to easily specify and extend algorithms. The state space exploration and model checking algorithms implemented rely on a set of Storage and Waiting Set components for efficient storage and exploration of state spaces. Furthermore, the SSE engine implements the Query Languages(s) used for writing state space queries and to verify properties.

**User interface.** The ASAP GUI makes it possible to create and manage *verification projects* consisting of a collection of *verification jobs*. Verification jobs are



(a) Graphical user interface



(b) SSE engine

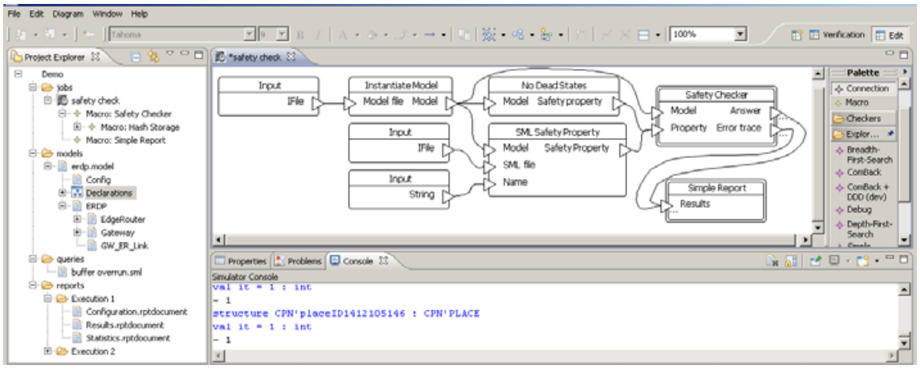
**Fig. 1.** ASAP platform architecture

constructed and specified using the verification Job Specification and Execution Language (JoSEL) [17] and the JoSEL Editor. We will briefly highlight the key ideas of JoSEL later. JoSEL and the JoSEL Editor are implemented using the Eclipse Modeling Framework and GMF, the Graphical Modeling Framework. The ASAP GUI additionally has a Model Loader component and a Model Instantiator component that can load and instantiate, e.g., CPN models [13] created with CPN Tools [5]. It is worth noticing that only the dark gray (red) boxes in Fig. 1 (CPN Model Loader, CPN Model Instantiator, and CPN Model Representation as well as the Model Simulator(s) component of the SSE engine) are language specific; all other components are independent of any concrete modeling language, and indeed we have implemented components for loading models specified in DVE, the input language of the DiVinE model checker.

The GUI has two different *perspectives* for working with verification projects: An *editing perspective* for creating and editing verification jobs, and a *verification perspective* for inspecting and interpreting verification results. Figure 2(a) shows a snapshot of the graphical user interface in the editing perspective. The user interface consists of three main parts apart from the usual menus and tool-bars at the top. To the left is an overview of the verification projects loaded, in this case just a single project named Demo is loaded. A verification project consists of a number of verification jobs, models, queries, and generated reports. In this case there is one verification job, safety check, concerned with checking safety properties. A CPN model named ERDP is loaded, which is a CPN model of an edge router discovery protocol from an industrial case study [14]. We have one query, checking if buffers of the model overflow, and two reports from two different verification job executions. At the bottom is a console making it possible to interact directly with the SSE engine using SML. This allows experimenting with the tool and issuing queries that need not be stored as part of the verification project. The area at the top-right is the editing area used to edit queries and verification jobs. Here, the safety checker job is being edited and the window shows its graphical representation in JoSEL. Other components may be added to this job by using the tool palette adjacent to the editing area.

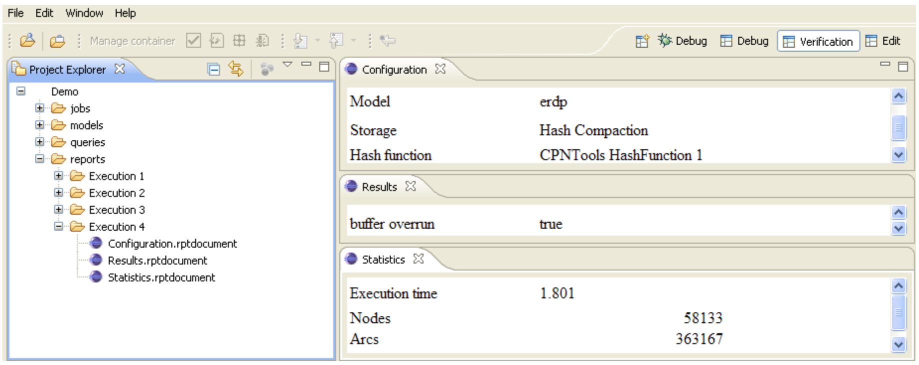
A snapshot of the verification perspective is shown in Figure 2(b). Here, a verification report is opened. It consists of three parts. The Configuration report lists general information like the model name or the different reduction techniques enabled, e.g., hash compaction in this case. The Results report specifies which properties were checked and whether or not they hold. In case of error, it also displays a counter-example that proves why the property does not hold. The Statistics report gives information on the state space, the exploration time and additional information depending on the reduction techniques used.

**The Job Specification and Execution Language.** JoSEL [17] is a graphical language inspired by data-flow diagrams that makes it possible to specify the formal models, queries, state space explorations, and processing of analysis results that constitute a verification job. The top-right panel in Fig. 2(a) shows a graphical representation of a JoSEL job that we use to illustrate the different key ideas behind this language.



```
val it = 1 : int
- 1
structure CPN'placeID1412105146 : CPN'PLACE
val it = 1 : int
- 1
```

(a) The editing perspective



| | |
|---|---|
| Model | erdp |
| Storage | Hash Compaction |
| Hash function | CPNTools HashFunction 1 |

| | |
|---|---|
| buffer overrun | true |

| | |
|---|---|
| Execution time | 1.801 |
| Nodes | 58133 |
| Arcs | 363167 |

(b) The verification perspective

**Fig. 2.** Snapshots of the graphical user interface

In JoSEL, *tasks* are the basic units of computation used as blocks to construct jobs. A task can, for instance, load a CPN model from a file or explore the state space of a model using a specific search algorithm. Tasks are graphically represented by rounded boxes.

A *job* consists of a set of interconnected tasks. Connections are used to specify a producer/consumer scenario: tasks can produce data that can be in turn used by another task. Each task has a set of *input ports* that specify the type of data it waits for in order to be executed. Once its execution is finished, the production of the task is specified by *output ports*. Both are graphically represented using small triangles placed at the left of tasks for inputs and at the right for outputs. In our example, the Instantiate Model task takes as input a CPN Model file, and from it, produces a Model which is an SML representation of the CPN model usable by the SSE engine (see the next paragraph and Section 3). This one can be consumed by the No dead states and SML Safety Property tasks that instantiate two properties, absence of deadlock and a user defined property. These are analyzed by the Safety checker task.

At this level, the type of algorithm or reduction techniques used by the Safety checker are not visible to the user. This is because this task has been defined as a *macro*. A macro is at the same time a job (described by the user with tasks and connections) and a task that can be part of other jobs. The graphical representation of macros differs from the one of tasks in that the rounded box is drawn using a double outline. Besides the advantages of clarifying the view of jobs and allowing the reuse of macros along different jobs, their use allows different levels of abstraction. Many users are not interested in the details of the safety checker whereas some with more background in model checking would perhaps like, for a specific model, to use a specialized search algorithm assumed to be especially efficient in that particular case. Double-clicking on the Safety checker macro expands the view of the macro and allows the user to tune the way properties are checked.

The main motivation of JoSEL is to provide the user with an intuitive and graphical language that allows different level of abstractions for users with different background in model checking such as: students, researchers, and industrial users.

**The state space search engine.** It is commonly agreed upon in the model checking community that no reduction technique works well on all models and that algorithms and methods are usually designed for a specific stage of the verification process. Therefore tools have to support several algorithms in order to prove useful.

Currently, ASAP supports checking deadlocks in systems and user specified safety properties. ASAP implements a broad range of techniques and below we briefly mention some of them.

*Bit-state hashing* and *hash-compaction* [12] are incomplete methods based on hashing and are generally used prior to any other analysis technique for their ability to quickly discover errors rather than proving their absence. The *sweep-line* method [15] exploits a notion of progression exhibited by many protocols to garbage collect states during the search thereby lowering peak memory requirements. The recently developed *ComBack* method [18,11] is especially suited to models having complex states, e.g., with many tokens in places for CPNs. It makes it possible to represent states with 16–24 bytes independently from the model, hence achieving a memory reduction close to the one provided by hash compaction without the loss of precision associated with that method. If, using these techniques, memory is still lacking, the user can switch to efficient *disk-based* algorithms [4,10]. In experiments reported in [10] we were able to explore large state spaces with $10^8$–$10^9$ states in a reasonable time (4–20 hours).

## 3 Extending ASAP

An important design guideline of ASAP is to provide a flexible and modular architecture that can be easily extended with new algorithms or modeling languages. We give in this section a brief overview of how this can be done.

```
functor SweepLineExploration(
   structure Model   : MODEL
   structure Storage: STORAGE
   val progressValue: Model.state -> int): EXPLORATION =
struct
   fun explore initialStates = ...
end
```

(a) First of lines of the sweep-line search algorithm in the SSE engine.

```
class SweepLineExplorationTask implements FunctorTask {
   String getName () { return "Sweep Line Exploration"; }
   String getFunctor () { return "SweepLineExploration"; }
   Value getReturnType () {
        return new Value ("Traversal", Exploration.class); }
   Value[] getParameters () {
        return new Value[] {
           new Value ("Model", Model.class),
           new Value ("Storage", Storage.class),
           new Value ("Progress Measure", Measure.class) }; }
   Exploration exec (Model m, Storage s, Measure p) {
        Exploration e = new Exploration (m.getSimulator ());
        m.getSimulator ().evaluate (
           e.getDeclaration () + " = " + getFunctor () +
           "(structure Model   = " + m.getStructure () +
           " structure Storage = " + s.getStructure () +
           " val progressValue = " + p.getName () + ")");
        return e; }
}
```

(b) Creation of a Sweep-Line Exploration task (see Fig 4) in the JoSEL editor.

**Fig. 3.** Integration of the sweep-line method in ASAP

**Integrating new algorithms.** Let us suppose that we wish to integrate the sweep-line method [15] into ASAP. Only the light gray (green) boxes in Fig. 1 are method specific and have to be considered for this integration. On the SSE engine side we have to implement the search algorithm used by this method. Since this one is independent from any storage method, and uses its own waiting set component to store unvisited states, we only have to implement an exploration component. The engine is based on a number of SML signatures (the equivalent of JAVA interfaces) among which the most important ones are: EXPLORATION that describes search algorithms, e.g., depth-first search; STORAGE that describes data structures used to store visited states, e.g., hash tables; and MODEL used to describe language dependent features of the analyzed model, e.g., how states are represented (see the next paragraph). The SweepLineExploration of Figure 3 is a generic EXPLORATION, i.e., an SML functor, that requires three parameters to be instantiated: Model, the model of which the state space is explored; a Storage data structure used to store visited states; and a function, progressValue, that maps each state to a progress value, here an integer (see [15] for details). As this functor implements the EXPLORATION signature, it has to define a function, explore, that explores the state space from some starting state(s) and returns a storage containing the set of visited states upon termination of the algorithm. Because of space limitations we have left out the implementation of the explore function.

For these changes to be visible in the graphical interface, we then have to extend the JoSEL language with the Method-specific tasks of Fig. 1. The main one, Sweep Line Exploration is graphically represented in Fig. 4. It corresponds to the instantiation of functor SweepLineExploration and is implemented in the JoSEL editor by the SweepLineExplorationTask class of Fig. 3. This one inherits from FunctorTask, which is used to describe JoSEL tasks that simply consist of the instantiation of a functor. The methods getName and getFunctor return the name of the task, i.e., the label appearing in the graphical representation of the task, and the name of the underlying SML functor. The input and output ports (their names and types) of the task are specified by the getParameters and getReturnType methods. Note that a FunctorTask can only have one output port, namely, the SML structure resulting from the instantiation of the functor. Also, there usually is a one-to-one mapping between the parameters of the functor and the items returned by method getParameters, as it is the case here. The last method, exec, specifies the SML code that is interpreted as the task is executed. Its parameters match the list of output ports specified by method getParameters.

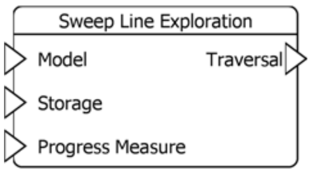


**Fig. 4.** Graphical representation of the Sweep Line Exploration task

**Integrating new modeling languages.** All search algorithms implemented by the SSE engine receive a model, which from the SSE engine point of view, is an SML structure implementing the MODEL signature (see Fig 5). To be valid, such a structure must define two types: the type of states and the type of events. For CPNs, the state type consists of a set of multi-sets over typed tokens and an event is a pair composed of a transition identifier and an instantiation

```
signature MODEL = sig
   type state
   type event
   val initialStates:  unit -> (state * event list) list
   val succ:  state * event -> (state * event list) list
   val stateToString: state -> string
   val eventToString: state -> string
end
```

**Fig. 5.** The MODEL signature

for the variables of this transition. To be able to explore the state space of the model, the engine must know its initial state(s) and from a given state how to calculate its successor(s). This is the purpose of functions initialStates and succ. Non-deterministic systems are supported. Indeed, both initialStates and succ return a list of states (rather than single states) along with their enabled events. Functions stateToString and eventToString return a user readable representation of states and events that can be used, for instance, to display counter-examples.

Note that providing a model structure is the minimal requirement to be able to use the SSE engine. Many algorithms or reduction techniques expect more information, e.g, a state serializer for external algorithms or an independence relation for partial order reduction.

The easiest way to integrate a new specification language is to write a compiler that, from a specification file, can produce an SML MODEL structure. Since the other components of the engine are independent of any concrete language, those will remain unchanged. Although our work focuses on the development of language independent algorithms, the architecture of the SSE engine does not prevent us from integrating algorithms or reduction techniques specifically tailored for a specific language, e.g., search algorithms that exploit Petri net invariants. It is sufficient to define a new signature that extends MODEL with the desired features.

## 4 Benchmarks

The ability of ASAP to load CPN and DVE models makes it possible to experimentally compare different algorithms and reduction techniques on models from our own collection [3], e.g., [9,14], and on the numerous models of the BEEM database [16].

For comparison, we have shown in Table 1 the performance of ASAP compared to CPN Tools and DiVinE. For each Model, the table shows its number

**Table 1.** Performance of ASAP, CPN Tools, and DiVinE on some models

| | Model | States | Time | | | Time (ComBack) | |
|---|---|---|---|---|---|---|---|
| | | | Basis | ASAP | Speedup | ASAP | Speedup |
| CPN Tools | Dining Philosophers | $40 \cdot 10^3$ | 6,614 | 27 | 245 | 55 | 120 |
| | Simple Protocol | $204 \cdot 10^3$ | 7,084 | 33 | 215 | 54 | 131 |
| | ERDP | $207 \cdot 10^3$ | 19,351 | 112 | 173 | 197 | 98 |
| | DYMO | $114 \cdot 10^3$ | 7,403 | 308 | 24 | 355 | 21 |
| | | Average on 4 models | | | 164 | | 92 |
| DiVinE | brp2.6 | $5.7 \cdot 10^6$ | 39 | 17 | 2.29 | 90 | 0.43 |
| | firewire_tree.5 | $3.8 \cdot 10^6$ | 227 | 525 | 0.43 | 388 | 0.59 |
| | plc.4 | $3.7 \cdot 10^6$ | 55 | 45 | 1.22 | 67 | 0.81 |
| | rether.4 | $9.5 \cdot 10^6$ | 51 | 34 | 1.52 | 191 | 0.27 |
| | | Average on 50 models | | | 1.39 | | 0.72 |

of States, the full exploration time with the Basis tool (CPN Tools or DiVinE) and with ASAP (with and without the ComBack method). Times are in seconds, and Speedup is the ratio between Basis and ASAP time. We see that ASAP performs significantly better than CPN Tools, achieving speedups of several orders of magnitude for both full state space generation and the ComBack method compared to full generation in CPN Tools. The performances of DiVinE and ASAP are comparable although slightly in favor of ASAP. On 50 models we observed an average speedup of 1.4 without using reduction. Even with the ComBack method, ASAP was able to perform at 0.7 of the speed of DiVinE despite the time overhead of the ComBack method.

## 5 Conclusion

ASAP is a graphical tool based on Eclipse for the analysis of CPN models and other formalisms. It provides the user with an intuitive and graphical language, JoSEL, for specification of verification jobs. To alleviate the state explosion problem, ASAP implements several algorithms and reduction techniques. Among these are: hash compaction, the sweep-line method, the ComBack method and external memory algorithms. The tool has been designed to be easily extended with new algorithms or specification languages and its modular architecture allowed us to write a sweep-line plug-in and a DVE plug-in to load DVE models in a few days without modifying the rest of the code. ASAP is also very useful for experimenting with and comparing algorithms as it gives the possibility to analyze more than 60 CPN and DVE models from our test-suite or from the BEEM database. Last but not least, ASAP significantly outperforms CPN Tools regarding performance and performs as well as DiVinE for DVE models. For these reasons we believe the tool to be suitable for students, researchers, and industrial users who would like to analyze CPN models or to experiment with different verification algorithms.

ASAP has replaced CPN Tools in our group to perform verification tasks and has been used to analyze an edge router discovery protocol [14] and a dynamic mobile ad-hoc network routing protocol [9]; and to experiment with state space algorithms [18,10,11]. It is also intended to be used in a future advanced state space course at Aarhus University.

We are currently considering adding new features to ASAP. In its current version, ASAP can analyze deadlocks and verify user defined safety properties. In the design phase of communication protocols, properties are, however, often specified in a temporal logic, e.g., LTL or CTL. The integration of temporal logic in ASAP is therefore one of our main priorities.

Since parallel machines are nowadays widespread, it is crucial that model checkers take advantage of this additional computational power. As the SSE engine of ASAP is currently single-threaded we consider extending it with parallel and distributed algorithms. In particular, we are working on a parallel version of the ComBack method of which we briefly mentioned the principle in [11].

**Availability.** ASAP is a stand-alone tool available for Windows XP/Vista, Linux, and Mac OS X. The current version, 1.0, can be freely downloaded from our web page [2].

## References

1. SPIN homepage, http://www.spinroot.com/
2. ASAP download, http://www.daimi.au.dk/~ascoveco/download.html
3. The ASCoVeCo Project: Advanced State Space Methods and Computer tools for Verification of Communication Protocols, http://www.daimi.au.dk/~ascoveco/
4. Bao, T., Jones, M.: Time-Efficient Model Checking with Magnetic Disk. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 526–540. Springer, Heidelberg (2005)
5. CPN Tools homepage, http://www.daimi.au.dk/CPNTools/
6. Design/CPN, http://www.daimi.au.dk/designCPN/
7. DiVinE homepage, http://divine.fi.muni.cz/
8. Eclipse homepage, http://www.eclipse.org/
9. Espensen, K.L., Kjeldsen, M.K., Kristensen, L.M.: Modelling and Initial Validation of the DYMO Routing Protocol for Mobile Ad-Hoc Networks. In: van Hee, K.M., Valk, R. (eds.) PETRI NETS 2008. LNCS, vol. 5062, pp. 152–170. Springer, Heidelberg (2008)
10. Evangelista, S.: Dynamic Delayed Duplicate Detection for External Memory Model Checking. In: Havelund, K., Majumdar, R., Palsberg, J. (eds.) SPIN 2008. LNCS, vol. 5156, pp. 77–94. Springer, Heidelberg (2008)
11. Evangelista, S., Westergaard, M., Kristensen, L.M.: The ComBack Method Revisited: Caching Strategies and Extension with Delayed Duplicate Detection. Transactions on Petri Nets and Other Models of Concurrency (to appear, 2009)
12. Holzmann, G.J.: An Analysis of Bitstate Hashing. In: FMSD 1998, vol. 13, pp. 289–307 (1998)
13. Jensen, K., Kristensen, L.M.: Coloured Petri Nets – Modelling and Validation of Concurrent Systems. Springer, Heidelberg (in preparation)
14. Kristensen, L.M., Jensen, K.: Specification and Validation of an Edge Router Discovery Protocol for Mobile Ad Hoc Networks. In: Ehrig, H., Damm, W., Desel, J., Große-Rhode, M., Reif, W., Schnieder, E., Westkämper, E. (eds.) INT 2004. LNCS, vol. 3147, pp. 248–269. Springer, Heidelberg (2004)
15. Kristensen, L.M., Mailund, T.: A Generalised Sweep-Line Method for Safety Properties. In: Eriksson, L.-H., Lindsay, P.A. (eds.) FME 2002. LNCS, vol. 2391, pp. 549–567. Springer, Heidelberg (2002)
16. Pelánek, R.: BEEM: Benchmarks for Explicit Model Checkers. In: Bošnački, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 263–267. Springer, Heidelberg (2007), http://anna.fi.muni.cz/models/
17. Westergaard, M., Kristensen, L.M.: JoSEL: A Job Specification and Execution Language for Model Checking. In: CPN 2008. DAIMI-PB, vol. 588, pp. 83–102 (2008)
18. Westergaard, M., Kristensen, L.M., Brodal, G.S., Arge, L.: The ComBack Method – Extending Hash Compaction with Backtracking. In: Kleijn, J., Yakovlev, A. (eds.) ICATPN 2007. LNCS, vol. 4546, pp. 445–464. Springer, Heidelberg (2007)

# The Access/CPN Framework: A Tool for Interacting with the CPN Tools Simulator*

Michael Westergaard[1] and Lars Michael Kristensen[2]

[1] Department of Computer Science, Aarhus University, Denmark
mw@cs.au.dk

[2] Department of Computer Engineering, Bergen University College, Norway
lmkr@hib.no

**Abstract.** Coloured Petri nets (CP-nets or CPNs) is a widely used formalism for describing concurrent systems. CPN Tools provides a mature environment for constructing, simulating, and performing analysis of CPN models. CPN Tools also has limitations if, for example, one wishes to extend the analysis capabilities or to integrate CPN models into external applications. In this paper we present Access/CPN, a framework that facilitates such extensions. Access/CPN consists of two interfaces: one written in Standard ML, which is very close to the simulator component of CPN Tools, and one written in Java, providing an object-oriented representation of CPN models, a means to load models created using CPN Tools, and an interface to the simulator. We illustrate Access/CPN by providing the complete implementation of a simple command-line state space exploration tool.

## 1 Introduction

Coloured Petri nets (CP-nets or CPNs) provide a useful formalism for describing concurrent systems, such as network protocols and workflow systems. CPN Tools [2] provides an environment for editing and simulating CPN models, and for verifying correctness using state space analysis. Sometimes, this is not enough, however. As CPN Tools is inherently graphical it cannot be controlled by external applications, so it is difficult to use CPN Tools in settings that are outside its scope of interactive use by one user. Such examples include repeated simulation on multiple servers in a grid, describing a complex decision procedure as a CPN model for use in an application, and allowing users to set parameters of a model using a custom user interface. Due to the architecture of the simulator and state space tool in CPN Tools, it is also difficult to implement new analysis techniques like new more efficient state space methods.

CPN Tools basically consists of two components (see Fig. 1 (top)): a graphical editor (middle) and a simulator daemon (right). The editor allows users to interactively construct a CPN model that is transmitted to the simulator, which checks it for syntactical errors and generates model-specific code to simulate the

* Supported by the Danish Research Council for Technology and Production.

CPN model. The editor invokes the generated simulator code and presents results graphically. The editor can load and save models using an XML format (left in Fig. 1 (top)). The editor imposes most of the restrictions on the use of CPN Tools mentioned above. Replacing the editor with our own application, we can remove the limitations imposed by the editor. This has been done in several settings: In [10] simulation code is augmented with code to let it run within a web-server, allowing users to interact with the CPN model via a web-site to perform operational planning. In [8] the editor is replaced by a custom application to allow military planning, and in [12] the editor is replaced by a more general-purpose application, which makes it possible to make domain specific visualisations of CPN models. Each of these examples use their own ad-hoc way to interact with the simulator. The simulator suffers from two problems making such interaction difficult. Firstly, the protocol used for communication between the editor and the simulator is low-level and complex to implement. Secondly, the CPN simulator is optimised for simulation and incremental code generation making it difficult to use for other purposes.

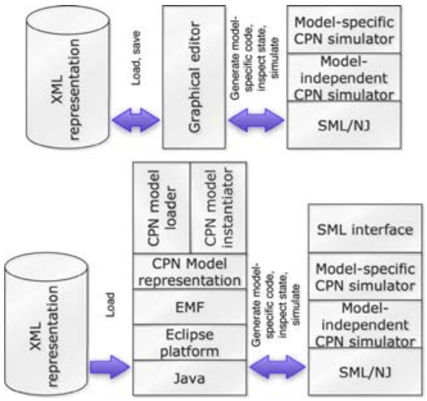

XML representation
Load, save
Graphical editor
Generate model-specific code, inspect state, simulate
Model-specific CPN simulator
Model-independent CPN simulator
SML/NJ
CPN model loader
CPN model instantiator
CPN Model representation
EMF
Eclipse platform
Java
XML representation
Load
Generate model-specific code, inspect state, simulate
SML interface
Model-specific CPN simulator
Model-independent CPN simulator
SML/NJ

**Fig. 1.** Architecture of CPN Tools (top) and Access/CPN (bottom)

In this paper we propose Access/CPN [1] which comprises two interfaces to the CPN simulator, alleviating the problems mentioned above. Access/CPN does not aim to replace CPN Tools as an editor for CPN models, but rather to allow researchers and developers to make experiments with the CPN formalism and use it as part of application development. Access/CPN has been developed as part of the ASAP model checking platform [9] and will be presented in this context, but is applicable also in general. One interface of Access/CPN is written in Java and the other in Standard ML (SML). Fig. 1 (bottom) shows how Access/CPN augments and replaces parts of CPN Tools. The Java interface (middle) consists of an object-oriented representation of CPN models, the ability to transmit this representation to the simulator and to perform simulation and inspection of the current state in the simulator. Furthermore, it includes a loader which can import models created using CPN Tools. The SML interface (right in Fig. 1 (bottom)) encapsulates the data-structures used in the simulator and provides an interface to a CPN model facilitating fast simulation, useful for efficient analysis and other applications executing transitions with little or no user-interaction.

In this paper, we first describe and exemplify the SML interface using a simple example, and then turn to the Java interfaces. The two parts can be read independently, and each assumes a knowledge of the language used. Finally, we conclude, compare with related work, and provide directions for further work.

## 2 The SML CPN Model Interface

In this section we describe the SML interface of Access/CPN. The aim of the interface is to provide efficient access to the CPN simulator, in particular with the goal of implementing new and more efficient analysis methods. To support this, the SML interface provides an interface to the state of a CPN model and to execute enabled transitions. For performance reasons, this interface is written in SML like the CPN simulator.

**Example CPN Model.** To exemplify the SML interface, we use a CPN model of a simple stop-and-wait protocol with one sender and two receivers as an example. The top module of the model can be seen in Fig. 2 (top), where we have a substitution transition for the sender, the network, and one for each receiver. The network has a maximum capacity modeled by the Limit place. If the network has available capacity, the sender (Fig. 2 (bottom left)) transmits packets from Send onto the A place. The network (Fig. 2 (bottom middle)) transmits the packet to B1 and B2, optionally dropping one or both of the packets. The receivers (Fig. 2 (bottom right)) receive the packets on Received and transmit back acknowledgment onto C1 or C2, which the network transmits to D, optionally dropping one or both. When the sender receives acknowledgements from both receivers, the NextSend counter is updated and the transmission is repeated for the next packet. The model consists of four modules: Top, Sender, Network, and Receiver. The Receiver module is instantiated twice in the module Top corresponding to the substitution transitions Receiver 1 and Receiver 2.

### 2.1 Model Interface

The SML interface is designed with state space analysis in mind, but can be used for other purposes. It is designed to be independent of the actual formalism, making it possible to develop tools that are formalism-independent. The interface can be seen in Listing 1. It defines the concepts of states and events (ll. 2–3). The most important functions are getInitialStates (l. 7) and nextStates (l. 9). getInitialStates returns a list of initial states (in order to support non-deterministic formalisms, this is not restricted to being a single state) and a list of enabled events for each state. nextStates takes a state and an event, and returns successors using the same format as getInitialStates. If the given event is not enabled, the exception EventNotEnabled (l. 4) is raised. Additionally, the interface has a function for executing a sequence of events, executeSequence (l. 11), which works like nextStates except it executes any number of events, and two functions, stateToString and eventToString (ll. 13–14) for converting states and events to their string representation.

In addition to providing an implementation of the Model signature, the SML interface also provides utility functions like a hash-function, a partial order, and marshalling functions. Additionally, an interface for inspecting the model is provided, allowing users to create news model-specific functions, but not to modify the model; for this we suggest using the Java interface instead.

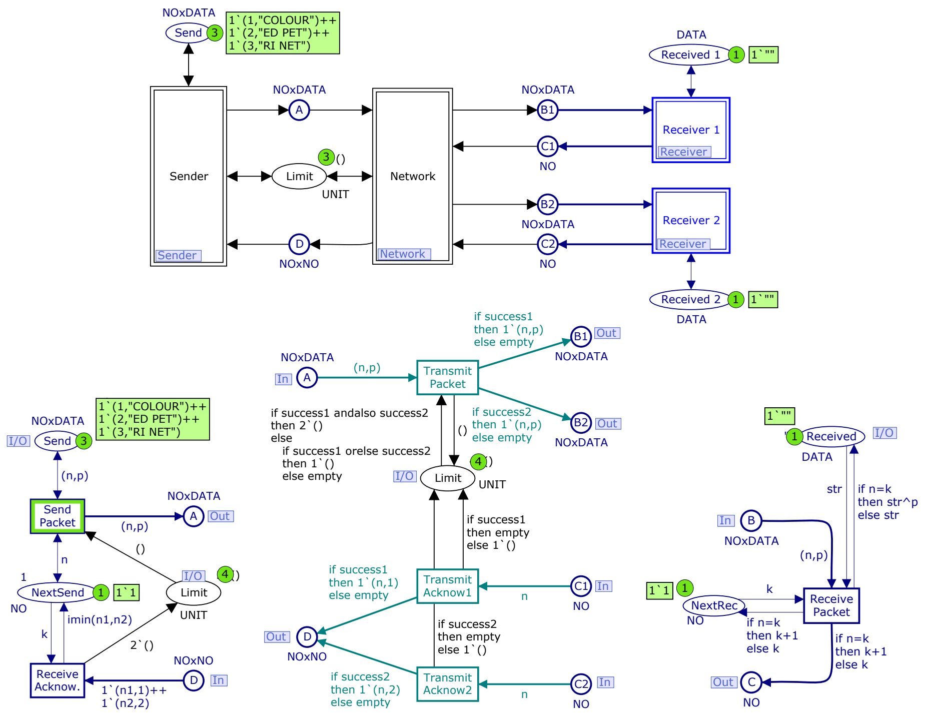


**Fig. 2.** Top (top), sender (bottom left), network (bottom middle), and receiver (bottom right) modules of a simple stop-and-wait protocol model with two receivers

**Listing 1.** Model interface

```
signature MODEL = sig
    eqtype state
    eqtype event
    exception EventNotEnabled

    val getInitialStates: unit -> (state * event list) list  (* Get initial states and enabled events *)
    val nextStates: state * event -> (state * event list) list  (* Get successor states + enabled events *)
    val executeSequence: state * event list -> (state * event list) list  (* Execute event sequence *)
    val stateToString: state -> string  (* String representation of states *)
    val eventToString: event -> string  (* String representation of events *)
end
```

**Listing 2.** State representation

```
structure Mark : sig
  type Sender = { NextSend: NO ms }
  type Network = { }
  type Receiver = { NextRec: NO ms }
  type Top = { A: NOxDATA ms, B1: NOxDATA ms, B2: NOxDATA ms, C1: NO ms, C2: NO ms, D: NOxNO ms,
             Limit: UNIT ms, Received_1: DATA ms, Received_2: DATA ms, Send: NOxDATA ms, Network: Network,
             Receiver_1: Receiver, Receiver_2: Receiver, Sender: Sender }
  type state = { Top: Top, time: time }
  val get'Top'Receiver_1'NextRec : state -> NO ms  val set'Top'Receiver_1'NextRec : state -> NO ms -> state
  val get'Top'Receiver_2'NextRec : state -> NO ms  val set'Top'Receiver_2'NextRec : state -> NO ms -> state
  val get'Top'Receiver_1'B : state -> NOxDATA ms   val set'Top'Receiver_1'B : state -> NOxDATA ms -> state
  ... (* several more accessor functions *)
end
```

**States.** The interface in Listing 1 is formalism-independent. In order to instantiate the interface for CPN models, we need to define the types state and event. To increase familiarity for users of the state space tool of CPN Tools [2], we define a structure, Mark, with data types and functions for manipulating states. We want the type to reflect the hierarchical structure of the CPN model. Listing 2 shows (most of) the Mark structure for the model in Fig. 2. The type of the state is defined inductively in the hierarchy of the model. For each module, we define a record which contains entries for all places and sub-modules of the module. For example, in Listing 2 (l. 2) we see the record defined for the Sender module in Fig. 2 (bottom left). We see that we have only included real (non-port and non-fusion) places, i.e., so only the NextSend place is present. The type uses names from the model, and NextSend is thus represented using the record entry NextSend. The type of the NextSend is NO ms, i.e., multi-sets over the color NO. The multi-set type is the same as used by CPN Tools. Similarly, types are defined for Network (l. 3) which contains no real places, and Receiver (l. 4) which contains one real place. The Top module is more complex (ll. 5–7), but uses the same structure. It contains entries for all real (i.e., all) places (ll. 5–6), and entries for all sub-modules (ll. 6–7). The entries for sub-modules are named after the substitution transition and the type is that of the sub-module. For example, we see that the sub-module defined by the substitution transition Receiver 1 is represented by the entry Receiver_1 of type Receiver. At the top-level, we define the type of the state itself. As it is possible for a model to contain more than one top module, we add a new top level (l. 8), containing all top modules (here Top), an entry for all reference declarations (here there are none), and the model time. For example, see the initial state of the network protocol in Listing 3.

State records, like the one in Listing 3, can be used with SML pattern matching, built-in accessor functions, and by building new structures. For convenience, we have also created set- and get-functions to access all modules and places of the structure. These functions use the naming convention: the function name (get or set) followed by the path of the place or module, separated by quotes. The functions take a state as argument, and getter functions return either a multi-set or a record describing the selected module. Setter functions also take a parameter of multi-set or record type and returns a new state like the one given with the selected place/module replaced. Examples of setter and getter functions can be seen in Listing 2 in ll. 9–11. In addition to providing accessors for the real places represented in the state record, we also provide accessors to port and fusion places, so it is possible to use, e.g., get'Top'Receiver_1'B, to get the marking of the port place B in the receiver (really B1 on the Top module).

**Listing 3.** Initial state of the network protocol example

```
val initial = { Top = { A = empty, B1 = empty, B2 = empty, C1 = empty, C2 = empty, D = empty, Limit = 3`(),
  Received_1 = 1`"", Received_2 = 1`"", Send = 1`(1,"COLOUR")++1`(2,"ED PET")++1`(3,"RI NET"), Network = {},
  Receiver_1 = {NextRec = 1`1}, Receiver_2 = {NextRec = 1`1},  Sender = {NextSend = 1`1} }, time = 0 }
```

**Events.** Events are defined by the data-type in Listing 4. We define a constructor for each transition named after the module it resides on and the name of the transition (same naming convention as for accessor functions in the state representation). Each constructor is a pair of an instance number and a record with all variables of the transition. To alleviate the use of instance numbers, we define symbolic constants (l. 8) for the path to each module instance. Using this, we can refer to Receive_Acknow on Sender as `Bind.Sender'Receive_Acknow (Bind.Top.Sender, {k, n1, n2})`.

**Listing 4.** New representation of events

```
1 structure Bind : sig
2   datatype event = Network'Transmit_Acknow1 of int * {n: INT, success1: BOOL}
3                  | Network'Transmit_Acknow2 of int * {n: INT, success2: BOOL}
4                  | Network'Transmit_Packet of int * {n: INT, p: STRING, success1: BOOL, success2: BOOL}
5                  | Receiver'Receive_Packet of int * {k: INT, n: INT, p: STRING, str: STRING}
6                  | Sender'Receive_Acknow of int * {k: INT, n1: INT, n2: INT}
7                  | Sender'Send_Packet of int * {n: INT, p: STRING}
8   val Top: int  val Top'Network: int  val Top'Receiver_1: int  val Top'Receiver_2: int  val Top'Sender: int
9 end
```

**Listing 5.** Implementation of a simple state space exploration algorithm

```
 1 exception Violating of CPNToolsModel.state
 2 fun combinator (h2, h1) = Word.<<(h1, 0w2) + h1 + h2 + 0w17
 3 val hash = CPNToolsHashFunction combinator
 4 fun none _ = false
 5 fun dead (_, events) = List.null events

 7 fun dfs predicate states =
 8 let fun equals (a, b) = a = b
 9     val storage = HashTable.mkTable (hash, equals) (1000, LibBase.NotFound)
10     fun dfs'' state [] = ()
11       | dfs'' state (event::events) = let val successors = CPNToolsModel.nextStates (state, event)
12                                           val _ = dfs' successors
13                                       in dfs'' state events
14                                       end
15     and dfs' [] = ()
16       | dfs' ((state, events)::rest) = if Option.isSome (HashTable.find storage state)
17                                        then dfs' rest
18                                        else let val _ = HashTable.insert storage (state, ())
19                                                 val violates = predicate (state, events)
20                                             in if violates
21                                                then raise Violating state
22                                                else (dfs'' state events; dfs' rest)
23                                             end
24 in (dfs' states; (NONE, storage)) handle Violating state => (SOME state, storage)
25 end
```

## 2.2 Example: State-Space Exploration

To illustrate the use of the SML interface, consider the implementation of a simple state space exploration algorithm in Listing 5 using the primitives presented above. The algorithm performs a recursive depth-first traversal of the state space and stores already expanded states in a hash-table. If a state not satisfying the property is found an exception is raised. The code first (l. 1) defines an exception to raise if a violating state is found, a built-in hash-function for states is instantiated (ll.2–3), and we define a predicate that is never satisfied (l. 4) and one that checks for dead-locks (l. 5). The rest is the actual algorithm, which takes a predicate to apply to each state and a list of states from which to start the exploration. The function defines the storage using SML's built-in HashTable (ll. 9). Then two mutually recursive functions dfs' and dfs'' are defined. dfs' (ll.15–23) traverses a list of states. It starts by checking if we have already traversed the state (l. 16), and, if so, continues with the next state (l. 17). If the state is new, it is stored (l. 18) and the predicate is checked (l. 19), and an exception is raised (l. 21) on violation. Otherwise we call dfs", which explores successors resulting from executing all enabled events for the given state using the nextStates primitive of the SML interface. dfs" calculates successor states for each event (l. 11), explores them using dfs' (l. 12), and traverses the remaining events (l. 13). The top function calls dfs' with the given state (l. 24) and returns the storage. If a violation is found it is also reported.

## 3 The Java CPN Model Interface

Many applications can benefit from tight integration with the CPN simulator. For non-algorithmic applications, Access/CPN has a Java interface providing an object-oriented representation of CPN models, an importer to load models from CPN Tools, and an implementation of the protocol used to communicate with the CPN simulator. We have created this interface in Java as Java is widely used and provides many frameworks and tools for creating user-friendly applications.

**Object Model.** Our object model builds on version 1.1.5 of ISO/IEC 15909-2, in particular the *PNML Core Model* (Fig. 2 in [6]) and the *High-Level Core Structure* (Fig. 8 in [6]). We have extended the PNML Core Model with a simplified version of *Modular PNML* [7] to support hierarchical nets. The model is not shown here due to space limitations, but is a straightforward representation of a CPN model. Basically, we have a PetriNet containing one or more Pages, which can contain any number of Arcs, Places and Transitions, each containing appropriate Labels (e.g., name, place type, and arc inscription). The net structure is basically an implementation of the PNML Core Model, and Labels is an implementation of the High-Level Core Structure. Pages can also contain Instances, corresponding to substitution transitions in CPN Tools. Instances contain ParameterAssignments corresponding to port/socket assignments in CPN Tools, and are simplified versions of ModInstance and ParamAssign from Modular PNML. Implementation of the object model is done using the Eclipse Modeling Framework (EMF) [3], is a framework for implementing object models. EMF generates implementation code from Java interfaces and provides features like automatic generation of XML marshaling (saving/loading models as XML) and an adaptor architecture making it possible to observe the object model for changes and to add new functionality without changing the classes.

*CPN Tools Importer.* Instances of the object model can be created programmatically, but it is desirable to create models using a graphical editor instead. For this reason we have created an importer, which allows programmers and users to import models created with CPN Tools. The importer imports the net-structure of the model and is able to load graphical information as well, as we have made a preliminary implementation of the *Graphical Information* from Fig. 3 in [6].

**Protocol Implementation.** The CPN Tools editor communicates with the simulator using a proprietary protocol, which is an implementation of a remote

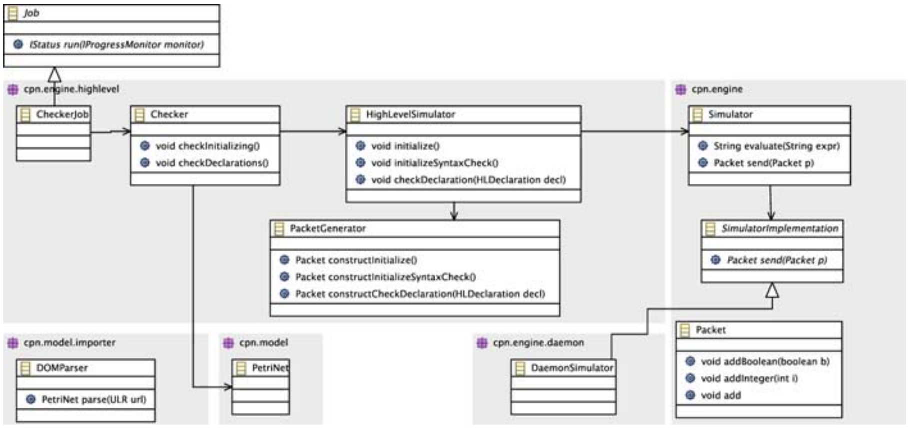


**Fig. 3.** Implementation of the protocol used to communicate with the simulator

procedure call (RPC) system. The protocol sends packets over a TCP/IP stream in the custom BIS (boolean, integer, string) format, which is a binary format that marshals simple data types. Packets have an integer op-code indicating the type of packet and some have an additional integer to indicate the command to execute. Commands must be combined in the correct way to syntax check a CPN model and generate simulator code for it. In order to implement this protocol, one must implement the BIS packet format, high-level constructs translating to the lower-level command integers, as well as a component that can take a CPN object model and send it all to the simulator for syntax check and simulation.

In Fig. 3 we see how the BIS protocol has been implemented in the Java interface of Access/CPN. It consists of five packages. cpn.model represents the object model described earlier, and cpn.model.importer contains the importer. The class Job, which is outside of any of the packages, is part of Eclipse. The remaining three packages implement the protocol used to communicate with the CPN simulator. The classes are listed with the most high-level to the left. Only the classes at the top are meant to be used by application developers. At the bottom-right, we have Packet, which implements the BIS package format. Such packets can be sent to a Simulator. The Simulator uses a delegate, DaemonSimulator, to communicate with the simulator via TCP/IP in the same way as CPN Tools. The Simulator class provides communication at the level of packets. The HighLevelSimulator provides stubs for calls supported by the simulator, making it possible to communicate using named methods. It uses a PacketGenerator factory to create packets as needed. The Checker class ties this to the object model hierarchy, and makes it possible to perform higher-level operations, such as syntax checking all declarations of a CPN model. CheckerJob further lifts this and makes it possible to syntax check an entire net using a single call. The checker job integrates with the Eclipse platform and can provide feedback to the user. If this is undesired, one can use the simpler Checker class, which can be used independently of the platform used. For simulation, one uses the HighLevelSimulator. One rarely needs to consider Simulator, PacketGenerator, and underlying classes.

### 3.1 Example: Command-Line State-Space Analyser

To illustrate the use of the Java interface, we implement a simple command-line application that uses the state-space algorithm from Sect. 2.2 to check models for dead-locks. We load a model given as a parameter, load the SML code shown previously, perform the exploration, and show the result to the user. The Java implementation can be seen in Listing 6. We start by importing classes needed (ll. 1–5). The code starts by obtaining the name of the file containing the CPN model to analyse (l. 9). The file is loaded as a Petri net (l. 10), and we create a HighLevelSimulator. As we are running this outside of an Eclipse application, we need to supply a simulator manually. The simulator requires a delegate, which must have information about which host and port to connect to as well as the name of the run-time system to load. All of this is handled in ll. 11–12. If we are using the interface as part of an Eclipse application, we can leave out the parameter in line 12. We then create a new CheckerJob (l. 13), which requires a name (here the string My model), a Petri net, and a high-level simulator. We start (schedule) the job and wait for it to terminate (l. 14). We then load the state-space algorithm developed in Sect. 2.2 (l. 15), and launch an exploration (ll. 16–18). We process the exploration result and show the user the violating state (if any) and the number of states explored. When done, the simulator is shut down (l. 20). The application can be executed as java StateSpaceTool protocol.cpn.

**Listing 6.** Implementation of a command-line state space exploration tool

```java
 1  import java.io.*; import java.net.*;
 2  import dk.au.daimi.ascoveco.cpn.engine.Simulator; import dk.au.daimi.ascoveco.cpn.engine.daemon.DaemonSimulator;
 3  import dk.au.daimi.ascoveco.cpn.engine.highlevel.HighLevelSimulator;
 4  import dk.au.daimi.ascoveco.cpn.engine.highlevel.checker.CheckerJob;
 5  import dk.au.daimi.ascoveco.cpn.model.PetriNet; import dk.au.daimi.ascoveco.cpn.model.importer.DOMParser;

 7  public class StateSpaceTool {
 8    public static void main(String[] args) throws Exception {
 9      String file = args[0];
10      PetriNet petriNet = DOMParser.parse(new URL("file://" + file));
11      HighLevelSimulator s = HighLevelSimulator.getHighLevelSimulator(
12          new Simulator(new DaemonSimulator(InetAddress.getLocalHost(), 23456, new File("cpn.ML"))));
13      try { CheckerJob checkerJob = new CheckerJob("My model", petriNet, s);
14            checkerJob.schedule(); checkerJob.join();
15            s.evaluate("use \"simple-dfs.sml\"");
16            System.out.println(s.evaluate("let val (s, storage) = dfs dead (CPNToolsModel.getInitialStates())" +
17                                          " in (s, HashTable.numItems storage) " +
18                                          "end"));
19      } finally {
20        s.destroy();
21  } } }
```

## 4 Conclusion and Future Work

In this paper we have described Access/CPN, which provides two interfaces to the CPN Tools simulator. One is close to the simulator and written in Standard ML providing fast access to the simulator, which is useful for analysis methods and other algorithmic applications. The other interface is written in Java and provides an object-oriented representation of CPN models, a means to import models created using CPN Tools, and abstractions of the communication with the CPN Tools simulator, making it possible to integrate CPN simulation into Java applications. Access/CPN is currently used as part of the ASAP platform [9] and in a master's thesis on automatic code generation of CPN models [4]. Access/CPN has already been distributed to several interested parties and is available from [1].

The BRITNeY Suite [12], originally developed for visualisation purposes, resembles Access/CPN as it also allows programmers to interact with the simulator of CPN Tools. BRITNeY requires that programmers implement their programs as extensions of BRITNeY, whereas Access/CPN makes it possible to embed the CPN simulator into other programs, allowing greater freedom. In that respect, Access/CPN is a significant improvement over the interface provided by BRITNeY. The Petri Net Kernel (PNK) [11] shares many of the same traits as Access/CPN, i.e., a representation of Petri nets and ability to use Petri net simulators. PNK, like BRITNeY, however, makes programmers write their programs within PNK and is more focused on making it easy to make Petri net tools rather than using Petri nets within external applications.

As part of future work, it would be useful to integrate the incremental syntax-checking capabilities of the CPN Tools simulator with the object model, so when the object model is altered it is automatically syntax-checked and the simulation code is regenerated. This would be useful for editors and model generating applications. It would also be interesting to use Access/CPN for integrating CPNs into meta-modelling tools like PNK or the High-Level Architecture [5].

## References

1. Access/CPN download, `http://www.daimi.au.dk/~ascoveco/accesscpn/`
2. CPN Tools webpage, `www.daimi.au.dk/CPNTools/`
3. Eclipse Modelling Framework (EMF), `www.eclipse.org/modeling/emf/`
4. Espersen, K.L., Kjeldsen, M.K.: Automatic Code Generation from Process-Partitioned Coloured Petri Net Models. Master's thesis, Dept. of Computer Science, University of Aarhus (2008)
5. Modeling and Simulation High Level Architecture. IEEE-1516
6. ISO/JTC1/SC7/WG19. Software and System Engineering—High-level Petri nets—Part 2: Transfer Format, version 1.1.5
7. Kindler, E., Weber, M.: A universal module Concept for Petri nets. In: Proc. des 8.Workshops Algorithmen und Werkzeuge fr Petrinetze, pp. 7–12 (2001)
8. Kristensen, L.M., Mechlenborg, P., Zhang, L., Mitchell, B., Gallasch, G.E.: Model-based Development of a Course of Action Scheduling Tool. STTT 10(1), 5–14 (2007)
9. Kristensen, L.M., Westergaard, M.: The ASCoVeCo State Space Analysis Platform. In: Proc. of 8th CPN Workshop. DAIMI-PB, vol. 584, pp. 1–6 (2007)
10. Lindstrøm, B.: Web-based interfaces for simulation of coloured petri net models. STTT 3(4), 405–416 (2001)
11. Weber, M., Kindler, E.: The Petri Net Kernel. In: Ehrig, H., Reisig, W., Rozenberg, G., Weber, H. (eds.) Petri Net Technology for Communication-Based Systems. LNCS, vol. 2472, pp. 109–123. Springer, Heidelberg (2003)
12. Westergaard, M., Lassen, K.B.: The BRITNeY Suite Animation Tool. In: Donatelli, S., Thiagarajan, P.S. (eds.) ICATPN 2006. LNCS, vol. 4024, pp. 431–440. Springer, Heidelberg (2006)

# DSSZ-MC – A Tool for Symbolic Analysis of Extended Petri Nets

Monika Heiner, Martin Schwarick, and Alexej Tovchigrechko

Computing Science Institute, Brandenburg University of Technology
Postbox 10 13 44, 03013 Cottbus, Germany
dsszmc@informatik.tu-cottbus.de

**Abstract.** DSSZ-MC supports the symbolic analysis of bounded place/transition Petri nets extended by read, inhibitor, equal, and reset arcs. No previous knowledge of the precise boundedness degree is required. It contains tools for the efficient analysis of standard properties (boundedness, liveness, reversibility) and CTL model checking, built on an object-oriented implementation of Zero-suppressed Binary Decision Diagrams and Interval Decision Diagrams. The main features are saturation-based state space generation, analysis of strongly connected components, dead state analysis with trace generation, and CTL model checking by limited backward reachability analysis. The tool is available for Windows, Linux, and Mac/OS.

## 1 Motivation

Considering efficient implementations of model checking tools for Petri nets, most research efforts so far are aimed at techniques for 1-bounded nets.

Reports on applying symbolic techniques to k-bounded nets can be occasionally found in the literature, but the only available tool implementing symbolic CTL model checking is SMART [CS03]. Though every bounded net can be simulated by a 1-bounded net, in practice the transformation is generally complicated and produces huge nets which can no longer be efficiently analysed.

However, biochemical networks in systems and synthetic biology often result in k-bounded models, caused by stoichiometry and the number of molecules or discrete concentration levels involved; see e.g. [KJH05], [GH06], [HGD08]. Many of these models could not be analysed by existing model checking tools and new techniques were required.

This paper gives an overview on the basic functionality of our new symbolic analysis tool that supports the efficient analysis of 1-bounded (zbdd-mc) and k-bounded (idd-mc) Petri nets extended by four non-standard arc types. It replaces our former symbolic CTL model checker of 1-bounded place/transition Petri nets [Noa99], which has been part of the model checking kit [SSE03] for quite a while.

We deliberately confine ourselves to an informal presentation; see [Tov08] for more detailed information concerning data structures and algorithms, as well as all related formal definitions.

## 2 Inputs

### 2.1 Extended Petri Nets

The tool supports the analysis of standard place/transition Petri nets extended for convenience by four special arc types: read arcs (identified by a black dot), inhibitor arcs (hollow dot), equal arcs (two black dots), and reset arcs (double arrow), which can be used simultaneously and always go from places to transitions. The standard firing rule is adapted accordingly. The enabling condition is extended in the following way: if there is an arc $a$ with a weight $w = V(p,t)$ connecting a place $p$ with a transition $t$, then $t$ can be enabled in a marking $m$ if the following conditions are satisfied:

- $a$ is a read arc $\wedge\, m(p) \geq w$,
- $a$ is an inhibitor arc $\wedge\, m(p) < w$,
- $a$ is an equal arc $\wedge\, m(p) = w$.

The token situation on $p$ is not changed by the firing of $t$, i.e. $m'(p) = m(p)$. Contrary, reset arcs do not alter the enabling condition, but involve a change of the marking on $p$ by firing of $t$:

- $m'(p) = 0$, if $p$ is not also a postplace of $t$
- $m'(p) = V(t,p)$, if $p$ is also a postplace of $t$

This net class is strictly more powerful than the class of standard place/transition Petri nets. Non-standard arcs help to express context conditions and allow themselves an elegant implementation of the analysis algorithms. For illustration consider the extended Petri nets in Figure 1, showing typical components of software verification models.

In the tradition of our previous model checking tools, the Petri nets have to be given in the Abstract Petri Net Notation (APNN) [BKK94] which is a language for the description of different classes of Petri nets. Keywords are similar to LaTeX commands. APNN has been adapted to allow the specification of the non-standard arc types. The input format can be generated, e.g., by the export feature of our hierarchical Petri net editor Snoopy [HRS08] which supports standard place/transition Petri nets as well as the extended Petri net class.

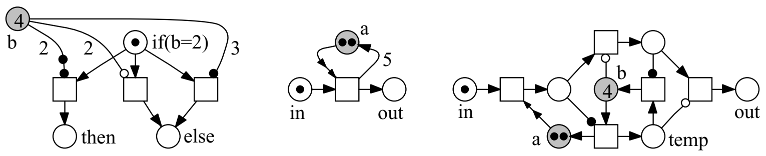


**Fig. 1.** Extended Petri nets components for software modelling (from left to right): if (b=2) then . . . else . . . , a := 5, a := b

All analyses supported by DSSZ-MC involve the construction of the state space, so the models have to be bounded. However, no previous knowledge of the exact boundedness degree is required. Please note, the tool does not perform a coverability check. If the Petri net is unbounded, a run will not terminate before the memory is exhausted (physically or as set by the memory option).

## 2.2 CTL Model Checking

We support model checking of Computational Tree Logic (CTL) according to the standard semantics; for a formal semantics definition see e.g. [CGP01]. The following grammar specifies a valid input for our CTL model checker.

```
ctl_input              = ctl_formula ';'
                       | ctl_formula ';' ctl_input .

ctl_formula            = '(' ctl_formula ')'
                       | unop '(' ctl_formula')'
                       | '('ctl_formula')' binop '('ctl_formula')'
                       | ae '[' ctl_formula 'U' ctl_formula ']'
                       | untemp '(' ctl_formula ')'
                       | ap .

unop                   = '!' .

binop                  = '*' | '+' | '->' | '<-' | '<->' .

ae                     = 'A' | 'E' .

untemp                 = 'AX' | 'EX' | 'AF' | 'EF' | 'AG' | 'EG' .

(* when using zbdd-mc *)
ap                     = PLACE .
--------------------------------------------------------------------
(* when using idd-mc *)
ap                     = PLACE cmp num
                       | PLACE 'in' interval .

cmp                    = '==' | '!=' | '>=' | '>' | '<=' | '<' .

interval               = '[' num ',' num ')' .

num                    = [0-9]+ .
```

PLACE has to be a valid place name of the Petri net to be analysed and in conformity with the standard conventions of $C^{++}$ identifiers. Places are read as Boolean variables in the case of 1-bounded Petri nets (zbdd-mc) and as integer variables in the case of k-bounded Petri nets (idd-mc).

Intervals are left-closed and right-open; thus, the lower bound is included and the upper bound is excluded from the specified interval. A CTL input file may also contain an arbitrary number of single-line and multi-line comments in $C^{++}$ style, allowing for better readable requirement specifications.

Additionally, there are a couple of non-standard temporal operators (EY, EH, FwdUntil, FwdGlobal), which, however, are beyond the scope of this introductory overview; see [Tov08] for details. They specifically allow for efficient forward traversal model checking (in preparation); compare Section 4.3.

## 3 Basic Data Structures

The tool is built upon an object-oriented implementation of Zero-suppressed Binary Decision Diagrams (ZBDDs) and Interval Decision Diagrams (IDDs). Both data structures are crucial for the reached performance, see Section 6. Here we sketch the basic principles to illustrate the compression effect, which we hope to gain generally by symbolic representations of very large state spaces.

ZBDDs [Min93] trail the success story of Binary Decision Diagrams (BDDs) [Bry86]. They are a special variation dedicated to the efficient representation of vast sets of sparse arrays, thus they are perfectly suited for the analysis of 1-bounded Petri Nets. The efficiency gain in comparison to standard BDDs comes from a special reduction rule which eliminates all variables that do not occur in a given marking (place is empty), compare Figure 2.

IDDs are a rather straightforward generalization of BDDs [Rid97], [ST98]. Arcs are labelled by (possibly) real intervals, the number of outgoing arcs of a node can vary, and values of IDD variables are not bounded. To analyse k-bounded Petri Nets, Boolean IDDs (IDDs with only two terminal nodes: 0 and 1) over integer intervals are used, compare Figure 3. Every Boolean IDD over $n$ variables represents a function $f$ that can be written as an interval logic formula over $n$ variables, i.e., as a formula containing only atomic propositions over integer variables combined by logic operators. To find the result of a function for the given values $a_i$ of all variables $x_i$, one follows a path through the graph from the root to a terminal node. In a non-terminal node $v$, an edge labelled with $I_j$ must be chosen if $var(v) = x_m$ and $a_m \in I_j$ . The result of the function is defined by the label of the terminal node reached. As usual, all paths to the terminal node 1 can be read as the encoding of a state set.

ZBDDs and IDDs enjoy the same powerful property in that they yield a canonical representation of Boolean or interval logic functions, respectively, if they are ordered and reduced. Boolean functions naturally encode sets of states of 1-bounded Petri nets, while interval logic functions allow a natural encoding of sets of states of k-bounded Petri nets. Both types of decision diagrams allow

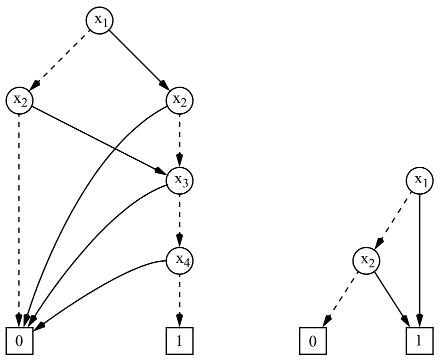


**Fig. 2.** The state set $\{ (1, 0, 0, 0), (0, 1, 0, 0) \}$ in BDD and ZBDD representation

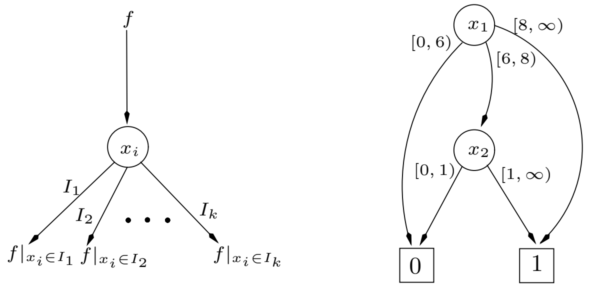


**Fig. 3.** Bool-Shannon decomposition for IDDs; general principle (left), and the IDD encoding the interval logic function $f = (x_1 \geq 8) \vee (x_1 \in [6, 8) \wedge x_2 > 0)$ (right)

straightforward implementation of the basic operations required for the various Petri net analyses.

All algorithms and options sketched in the next section are equally available for both data structures and, thus, for 1-bounded and k-bounded Petri nets.

## 4 Main Features

There is a wide variety of tool options among which the user can choose. We sketch here only the most important ones. According to our experience up to now, there seems to be no general rule for the best choice of options.

### 4.1 Preliminaries

The **variable ordering** (i.e. place ordering) is known to have a strong influence on the decision diagrams' size and, thus, on the computation speed. A bad choice may even totally prevent the state space's constructability. There is no general rule for the best method and we provide several options:

1. plain order as read from the APNN file
2. reverse order to the one read from the APNN file
3. a random order
4. heuristic 1 – an algorithm computing weights for places based on the net structure [Noa99]
5. heuristic 2 – an adaption of the previous one [Tov08] (default)
6. derived from the transition ordering
7. read from a file as specified by the user

Likewise, the **transition ordering** has a crucial impact on the performance of the chaining and the saturation algorithms. There are several options: plain, random, read from file, and three heuristics derived from the net structure (one of the heuristics is the default).

There are three categories of **state space generation algorithms.**

1. Common Breadth-First Search (BFS): an iteration fires sequentially all transitions (according to the transition ordering) before adding the new states to the state space.
2. Transitions chaining: like BFS, but the state space is updated after the firing of each single transition.
3. Saturation algorithm (SAT): transitions are fired in conformance with the decision diagram, i.e. according to an ordering, which is defined by the variable ordering. A transition is saturated if its firing does not add new states to the current state space. Transitions are bottom-up saturated (i.e. starting at the terminal nodes and going towards the root). Having fired a given transition, all preceding transitions have to be saturated again, either after a single firing (single) or the exhausted firing (fixpoint) of the current transition; see Section 6 for some benchmarks.

## 4.2 Analysis of General Behavioural Properties

No a priori knowledge of the precise **boundedness** degree is required. The boundedness of each individual place as determined by the state space construction can be written to a file.

Symbolic algorithms for the computation and enumeration of the terminal **strongly connected components** allow one to determine efficiently liveness and reversibility. All live/nonlive transitions can be written to a file.

The **dead state analysis** with trace generation determines all reachable dead states, which can be written to file as an interval logic expression, i.e., as an expression containing only atomic propositions over integer variables combined by logic operators. All empty places do not appear in the expression. A transition sequence producing one of the reachable dead states can be written to a file as well.

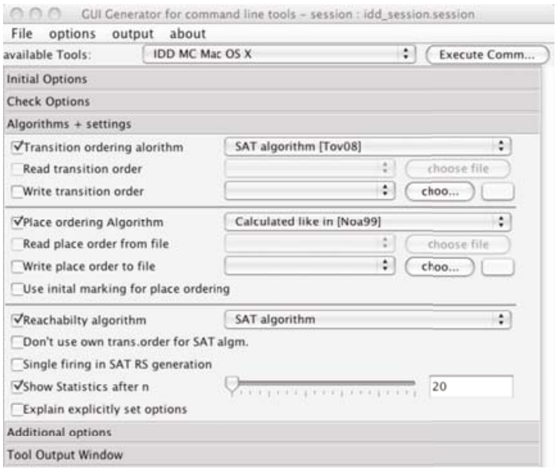
## 4.3 Model Checking of Special Behavioural Properties

**Limited backward reachability analysis**. The traditional backward analysis to check the operator *EU* can often be improved by deleting all states after each backward step, which are not reachable from the initial marking. We support this established technique complemented by efficient **saturation-based reachability analysis**.

Furthermore, **forward traversal reachability analysis** is under preparation, which is expected to be more efficient in some cases than any backward analysis technique. It will be applied for all formulae containing the non-standard forward operators *FwdUntil* and *FwdGlobal*, the past tense operators *EY* and *EH*, and for all formulae which can be transformed using them; see [Tov08] for transformation rules.

## 5 Graphical User Interface

The symbolic analyser DSSZ-MC per se is implemented as a command line tool. However, it comes along with an optional Graphical User Interface (GUI) which is designed to assist the user in choosing among the various tool features [Fra08]. Actually, it is a general GUI generator for command line tools configured by an xml file specifying the individual tool options and their interdependencies, so it can be easily translated into other languages or adjusted to new tool options or even other tools. Figure 4 gives a snapshot of the sub-window offering the choice amongst the possible algorithms and related settings of DSSZ-MC.



available Tools: IDD MC Mac OS X

- [✓] Transition ordering alorithm: SAT algorithm [Tov08]
- [ ] Read transition order
- [ ] Write transition order
- [✓] Place ordering Algorithm: Calculated like in [Noa99]
- [ ] Read place order from file
- [ ] Write place order to file
- [ ] Use inital marking for place ordering
- [✓] Reachabilty algorithm: SAT algorithm
- [ ] Don't use own trans.order for SAT algm.
- [ ] Single firing in SAT RS generation
- [✓] Show Statistics after n: 20
- [ ] Explain explicitly set options

**Fig. 4.** A snapshot of the GUI to choose among the various tool options

## 6 Benchmarks

We compare our tool with SMART [CS03] which is the best tool known for the symbolic analysis of k-bounded Petri nets with extended arcs. It deploys Multi-valued Decision Diagrams (MDDs) and also implements a saturation-based reachability algorithm. In contrast to DSSZ-MC, which handles monolithic Petri nets, SMART requires a suitable partitioning of the place set of the net to achieve good results. Unfortunately, defining a good partitioning is generally not a trivial task for non-regular models. SMART's saturation algorithm saturates MDD nodes, while DSSZ-MC applies the saturation strategy transition-wise to the whole decision diagram.

Our test suite comprises six Petri net models. The first three (philosophers, kanban, FMS) are taken from SMART's examples archive, which come along with a partitioning [MC99]. We added two biochemical networks [GH06], [HGD08], and a Petri net weakly computing the Ackermann function [PW03]. We tried our very best to find suitable partitionings for them.

**Table 1.** Some benchmarks for the ZBDD tool

| model | net | | | ZBDD-MC time secs | | SMART time secs |
|---|---|---|---|---|---|---|
| | \|P\| | \|T\| | \|states\| | fixpoint | single | |
| phils_N500 | 3000 | 2000 | 3.03e+313 | 12.77 | 12.83 | 7.60 |
| phils_N1000 | 6000 | 4000 | 9.18e+626 | 97.14 | 97.28 | 15.74 |

**Table 2.** Some benchmarks for the IDD tool [a)]

| model | net | | | IDD-MC time secs | | SMART time secs |
|---|---|---|---|---|---|---|
| | \|P\| | \|T\| | \|states\| | fixpoint | single | |
| kanban_N50 | 16 | 16 | 1.04e+16 | 14.68 | 0.54 | 386.00 |
| kanban_N75 | | | 7.83e+17 | - | 4.17 | 674.66 |
| kanban_N100 | | | 1.73e+19 | - | 6.50 | - |
| kanban_N200 | | | 3.17e+22 | - | 37.86 | - |
| kanban_N300 | | | 2.65e+24 | - | 265.01 | - |
| FMS_N100 | 22 | 21 | 2.70e+21 | 10.28 | 6.60 | 3.11 |
| FMS_N200 | | | 1.95e+25 | 38.09 | 57.41 | 37.56 |
| FMS_N250 | | | 3.46e+26 | 107.11 | 170.11 | 70.33 |
| FMS_N300 | | | 3.65e+28 | 907.37 | - | - |
| erk_N50 | 11 | 11 | 2.83e+8 | 2.74 | 2.27 | 25.23 |
| erk_N100 | | | 1.59e+10 | 3.99 | 2.51 | 231.41 |
| erk_N200 | | | 9.52e+11 | 29.80 | 4.08 | - |
| erk_N700 | | | 1.67e+15 | - | 55.18 | - |
| levchenko_N20 | 22 | 30 | 8.81e+10 | 2.34 | 2.39 | 6.82 |
| levchenko_N40 | | | 4.78e+14 | 3.00 | 2.44 | 133.23 |
| levchenko_N80 | | | 5.63e+18 | 17.44 | 3.34 | † |
| levchenko_N120 | | | 1.62e+21 | 153.95 | 5.73 | † |
| levchenko_N160 | | | 1.06e+23 | - | 10.88 | † |
| levchenko_N320 | | | 2.62e+27 | - | 133.54 | † |
| ack(3,2) | 23 | 24 | 1.44e+07 | 2.81 | 4.14 | 76.75 |
| ack(3,3) | | | 1.34e+09 | 6.79 | 32.97 | - |
| ack(3,4) | | | 1.42e+11 | 43.50 | - | - |

[a)] '–' means that physical memory was exhausted,
† we did not get results within one hour computation time.

The benchmarks were done on a 2 GHz Pentium M with 2 GB RAM running a 32bit Linux. In general, we used the default settings, which are in most cases the best possible choice. The memory option was set to 4 (up to 2.6 GB memory). It may be worthwhile to try different place ordering options to find the most suitable one. We did so for the kanban and FMS nets, for which we used option 6. Furthermore we tried the SAT algorithm with single firing first, which usually speeds up the construction significantly. Tables 1 and 2 show the total processing time for the state space computation using the saturation algorithms (fixpoint, single). These figures do not include the precious time a tool user spends to look for good options and suitable net partitionings.

The results suggest that the two tools under consideration may complement each other, depending on the power of the variable/transition orderings and net partitionings found.

## 7 Technicalities

The tool is a complete re-implementation in C$^{++}$, using the GNU MB Bignum Library (GMP). The parser of CTL formulae has been generated by the lexical analyser and parser generator *flex* and *bison*, respectively. The source code comprises about 22,800 LOC (Lines of Code, including comments) and has been tested on Windows, Linux, and MAC/OS.

The command line tool comes as two all-inclusive binaries (statically linked libraries), therefore, no special installation is required. Each takes about 1-2 MB memory, depending on the platform.

The GUI is written in Java and delivered by an installer as a Java jar file, so it requires the Java run-time environment (1.6 or higher).

The tool is available free of charge for scientific purposes at www-dssz.informatik.tu-cottbus.de. We provide the binaries for Windows, Linux, and Mac/OS. At the same website one also finds all the Petri net examples (in Snoopy, APNN, and SMART syntax) which we used as benchmarks in the preceding section. Maybe the reader finds better partitionings and/or options?

## 8 Conclusions

We have presented a tool for the symbolic analysis of extended Petri nets that supports the efficient analysis of general behavioural properties and CTL model checking as well. The models have to be bounded, however, no a priori knowledge of the precise boundedness degree is required. Crucial points for the tool's performance are the data structures used for the symbolic state space representation, and the algorithms, which exploit strongly connected components and the saturation principle.

We are working on a more detailed comparison with related tools, including liveness and reversibility decision as well as a representative set of model checking queries.

Besides the forward traversal strategy for efficient model checking mentioned in Subsection 4.3, we consider an extension of the current implementation by allowing a set of initial states. This set has then to be specified by an interval logic expression.

Continuing the encouraging results we have gotten so far, we are developing IDD-based model checking of Continuous time Stochastic Logic (CSL), see [Sch08]. A first prototype is available on the same website as the tool described in this paper.

## References

[BKK94] Bause, F., Kemper, P., Kritzinger, P.: Abstract Petri Net Notation. Technical report, Univ. Dortmund, CS Dep. (1994)

[Bry86] Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. IEEE Trans. on Computers C-35(8), 677–691 (1986)

[CGP01] Clarke, E.M., Grumberg, O., Peled, D.A.: Model checking. MIT Press, Cambridge (1999) (third printing, 2001)

[CS03] Ciardo, G., Siminiceanu, R.: Structural symbolic CTL model checking of asynchronous systems. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 40–53. Springer, Heidelberg (2003)

[Fra08] Franzke, A.: A concept for redesigning Charlie. Technical report, BTU Cottbus, Dep. of CS (2008)

[GH06] Gilbert, D., Heiner, M.: From Petri nets to differential equations - an integrative approach for biochemical network analysis. In: Donatelli, S., Thiagarajan, P.S. (eds.) ICATPN 2006. LNCS, vol. 4024, pp. 181–200. Springer, Heidelberg (2006)

[HGD08] Heiner, M., Gilbert, D., Donaldson, R.: Petri nets in systems and synthetic biology. In: Bernardo, M., Degano, P., Zavattaro, G. (eds.) SFM 2008. LNCS, vol. 5016, pp. 215–264. Springer, Heidelberg (2008)

[HRS08] Heiner, M., Richter, R., Schwarick, M.: Snoopy - a tool to design and animate/simulate graph-based formalisms. In: Proc. PNTAP 2008, associated to SIMUTools 2008. ACM digital library, New York (2008)

[KJH05] Koch, I., Junker, B.H., Heiner, M.: Application of Petri Net Theory for Modeling and Validation of the Sucrose Breakdown Pathway in the Potato Tuber. Bioinformatics 21(7), 1219–1226 (2005)

[MC99] Miner, A.S., Ciardo, G.: Efficient reachability set generation and storage using decision diagrams. In: Donatelli, S., Kleijn, J. (eds.) ICATPN 1999. LNCS, vol. 1639, pp. 6–25. Springer, Heidelberg (1999)

[Min93] Minato, S.: Zero-suppressed BDDs for set manipulation in combinatorial problems. In: Proc. 30th ACM/IEEE Design Automation Conference (DAC), pp. 272–277. ACM Press, New York (1993)

[Noa99] Noack, A.: A ZBDD package for efficient model checking of Petri nets. Technical report, BTU Cottbus, Dep. of CS (1999) (in German)

[PW03] Priese, L., Wimmel, H.: Theoretical Informatics - Petri Nets. Springer, Heidelberg (2003) (in German)

[Rid97] Ridder, H.: Analysis of Petri Net Models with Decision Diagrams. PhD thesis, University Koblenz-Landau (1997) (in German)

[Sch08] Schwarick, M.: Transient Analysis of Stochastic Petri Nets With Interval Decision Diagrams. In: Proc. 15th German Workshop on Algorithms and Tools for Petri Nets (AWPN 2008), September 2008. CEUR Workshop Proceedings, vol. 380, pp. 43–48. CEUR-WS.org (2008)

[SSE03] Schröter, C., Schwoon, S., Esparza, J.: The Model Checking Kit. In: van der Aalst, W.M.P., Best, E. (eds.) ICATPN 2003. LNCS, vol. 2679, pp. 463–472. Springer, Heidelberg (2003)

[ST98] Strehl, K., Thiele, L.: Symbolic model checking using interval diagram techniques. Technical report, Computer Engineering and Networks Lab (TIK), Swiss Federal Institute of Technology (ETH) Zurich (1998)

[Tov08] Tovchigrechko, A.: Model Checking Using Interval Decision Diagrams. PhD thesis, BTU Cottbus, Dep. of CS (2008)

# WORKCRAFT – A Framework for Interpreted Graph Models

Ivan Poliakov, Victor Khomenko, and Alex Yakovlev

Newcastle University, United Kingdom
{ivan.poliakov, victor.khomenko, alex.yakovlev}@ncl.ac.uk

**Abstract.** A large number of models that are employed in the field of concurrent systems design, such as Petri Nets, gate-level circuits, Static Data Flow Structures and Conditional Partial Order Graphs have an underlying static graph structure. Their semantics, however, is defined using additional entities, e.g. tokens or node/arc states, which in turn form the overall state of the system. We jointly refer to such formalisms as *Interpreted Graph Models*. The similarities in notation allow for links between different models to be created, such as interfaces between different formalisms or conversion from one model type into another, which greatly extend the range of applicable analysis techniques.

This paper presents the new version of the WORKCRAFT tool designed to provide a flexible common framework for development of Interpreted Graph Models, including visual editing, (co-)simulation and analysis. The latter can be carried out either directly or by mapping a model into a behaviourally equivalent model of a different type (usually a Petri Net). Hence the user can design a system using the most appropriate formalism (or even different formalisms for the subsystems), while still utilising the power of Petri Net analysis techniques. The tool is platform-independent, highly customisable by means of plug-ins, and is freely available for academic use.

## 1 Introduction

Petri nets (PNs) have been used for a long time as a formalism that, while simple enough to understand intuitively, is yet quite expressive and natural for modelling and designing concurrent systems. The value of PNs arises from the fact that there exists mature theory and numerous tools that are able to efficiently verify various behavioural properties of a PN. In particular, model checking [3] is an automatic technique able to either prove that a certain property (deadlock freeness, mutual exclusion of places, etc.) holds for the given PN or generate a trace demonstrating that the property is violated. This information is very useful for troubleshooting, and often allows to detect and fix errors early in the system design process.

However, PNs are a low-level formalism, much like an assembly language. The size of a PN required to describe the behaviour of a realistic system can become so large that a designer is unable to comprehend it. Hence, using them directly

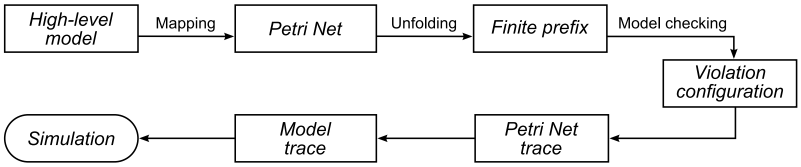


**Fig. 1.** Model verification workflow using WORKCRAFT

to describe a system is often unworkable and error-prone, and in practice higher level formalisms are often employed, e.g. gate-level digital circuits, networks of handshake components [2], Static Data Flow Structures [14] and Conditional Partial Order Graphs [9]. Such formalisms, in turn, require analysis and verification methods; however, development of detailed theory and special tools for each of them is often impractical – it may be more efficient to express a formalism in terms of another one, e.g. a PN, for which mature theory and tools have already been developed. Then, the result of the analysis (e.g. a violation trace) can be propagated back, interpreted in terms of the original model and shown to the designer (see Figure 1). Naturally, PNs are a good choice for the target model, as their compositions are well understood, and efficient model checking tools for PNs are readily available.

A common feature of the high-level models mentioned above, as well as PNs themselves, is the presence of an underlying static graph structure. Their semantics are defined using additional entities, such as tokens or node/arc states, which together form the overall state of the system. We jointly refer to such formalisms as *Interpreted Graph Models (IGM).* The similarities in notation and expressive power allow a number of basic operations on these formalisms, such as serialisation, visualisation, editing and translation from one formalism into another, to be generalised. More complex operations on the models can also be used, such as interfacing one model type with another. This enables the designer to model subsystems using the most appropriate formalism, while still maintaining the ability to simulate and analyse the overall system.

This paper presents the new version of the WORKCRAFT tool that is designed to provide a flexible common framework for Interpreted Graph Models. The tool is built using a plug-in driven architecture, and thus is easily extendable to other IGMs, as well as new analysis/verification modules. WORKCRAFT provides a GUI environment that facilitates model entry, supports interactive visual simulation, convenient "single-click" verification using external tools, and automatic back-propagation of violation traces from one model to another. The latter enables such traces to be immediately simulated in the high-level model, greatly increasing a designer's productivity.

## 2 Objectives

The primary design goal of the WORKCRAFT framework is twofold. One target category of users are the researchers who would like to define new Interpreted

Graph Models, while the other category are those who wish to design, analyse and verify a system using already implemented formalisms. To appeal to the former category, a plug-in based architecture was designed, which allows new formalisms to be introduced with the minimum effort – the benefits of the visual editing and simulation are inherited from the framework. Also, some of the important tools, such as the Petri Net mapping engine, were designed as general-purpose: a developer of a model only needs to write a mapping specification file using a straightforward XML-based format to receive the benefits of automated PN based verification features of the framework.

A major focus in development was a clean and fast visualisation engine, which is based on custom-written hardware-accelerated vector graphics renderer. The visualisation engine is used to support graphical editing and interactive simulation; it also supports graphics output into widely used vector file formats (SVG, EPS and PDF). The interactive simulation is one of the main features of the framework, that allows one to simulate the implemented IGMs in a variety of ways, such as user-controlled step-by-step simulation, automatic simulation with random choice of steps, or violation trace replay.

In contrast to the previous version of WORKCRAFT that focused exclusively on the analysis of Static Data Flow Structures [12], the new version aims to provide a general-purpose framework, not focused on one particular formalism, but exploiting the enormous potential of different aspects of IGMs interoperability.

## 3 Functionality

In this section we give a brief overview of WORKCRAFT's functionality and present a number of screenshots to give a general impression of the GUI.

### 3.1 Overview

The main window of WORKCRAFT is shown in Figure 2. The main menu (1), besides the standard file and editing operations, includes the automatically selected set of tools which are applicable to the current model. The editor settings (2) allow the user to enable or disable editing features such as snap-to-grid. The document view (3) presents the current model graphically. It is used for navigating the model and provides scaling (using the mouse wheel) and panning (holding right/middle mouse button and dragging) operations to control the viewport. The same view is also used for the interactive simulation. Editor tools (4) are used to switch between editing modes, such as creating and deleting the model components (represented by the vertices in the graph with specific semantics, e.g. places and transitions in a PN or gates and latches in a gate-level circuit), as well as connecting the components (e.g. by arcs in a PN or by wires in a gate-level circuit).

The property editor (5) displays the properties of the currently selected component and allows the user to edit them, e.g. to change the label or number of tokens in a PN place, or the type and parameters of a circuit gate. The component list (6) shows the set of all components supported by current model. A

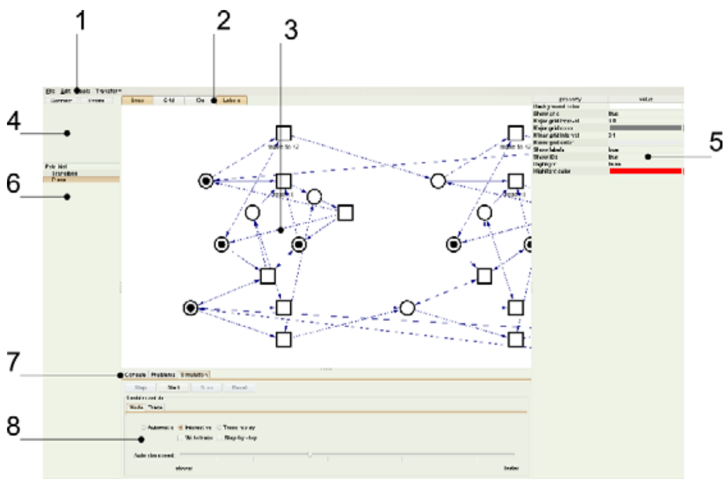
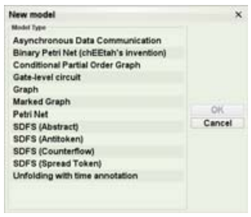


**Fig. 2.** The main GUI window of WORKCRAFT

component can be added to the document either by dragging it from the component list and dropping onto the document view, or by using a hotkey, which is optionally specified by the component definition. All components are further assigned numeric hotkeys from '1' to '9', corresponding to their order in component list for faster access. The utility area (7) has three tabs: the *console*, which is used to display various information during normal execution of the program and also allows to execute scripts; the *problem list* displaying errors which might have occurred during execution; and the *simulation control panel* which is discussed in detail below.

## 3.2 Working with Models

The new model dialogue (Figure 3) accessible through the file menu is used to start working on a new model. The dialogue presents the user with a choice of currently supported models. Each of these models is specified using a plug-in, so the user is able to customise the selection by removing the models that he/she is not going to use. After the model has been created, the user can add, connect and edit the components using the document view (Figure 2).



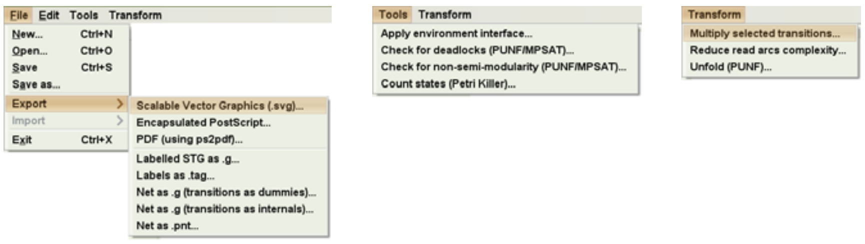
New model ×

Model Type

Asynchronous Data Communication
Binary Petri Net (chEEtah's invention)
Conditional Partial Order Graph
Gate-level circuit
Graph
Marked Graph
Petri Net
SDFS (Abstract)
SDFS (Antitoken)
SDFS (Counterflow)
SDFS (Spread Token)
Unfolding with time annotation

OK
Cancel

**Fig. 3.** New model dialogue


File
Edit
Tools
Transform
New... Ctrl+N
Open... Ctrl+O
Save Ctrl+S
Save as...
Export
Import
Exit Ctrl+X
Scalable Vector Graphics (.svg)...
Encapsulated PostScript...
PDF (using ps2pdf)...
Labelled STG as .g...
Labels as .tag...
Net as .g (transitions as dummies)...
Net as .g (transitions as internals)...
Net as .pnt...
Tools
Transform
Apply environment interface...
Check for deadlocks (PUNF/MPSAT)...
Check for non-semi-modularity (PUNF/MPSAT)...
Count states (Petri Killer)...
Transform
Multiply selected transitions...
Reduce read arcs complexity...
Unfold (PUNF)...

(a) File menu (b) Tools menu (c) Model transform menu

**Fig. 4.** Items of the main menu

*Import/export menu* is accessible from the File menu, see Figure 4(a). There are two kinds of export: graphics export and model export. Graphics export is used to save the visual snapshot in a vector format, such as SVG or EPS. This feature greatly facilitates creation of figures that can be used for writing articles or documentation. Model export, on the other hand, is used to store the model in a format usable by other tools. *Tools menu*, see Figure 4(b), contains the currently applicable tool plug-ins that are used for model analysis. The set of available tools depends on the current model type and installed plug-ins. For example, if the currently opened model is a gate-level circuit, tools that check the circuit for deadlocks and hazards will automatically be inserted into this menu. *Transform menu*, see Figure 4(c), contains tools that are used to change the model structure or to convert it into another model type. For example, a circuit can be converted into a Petri net by using the "convert into Petri Net" command from this menu.

*Simulation controls*, see Figure 2(area 8), allow the user to perform the interactive simulation of the current model in several ways. *Start, Step, Stop* and *Reset* buttons are used to switch between the simulation and editing modes. When the simulation is started using the Start button, all editing is disabled until the simulation is stopped using the Stop button. After the simulation has been stopped, the system can be reset to the pre-simulation state by using the Reset button. *Automatic simulation* will periodically fire components that are currently enabled. If several such components exist simultaneously, one of them will be chosen randomly. The interval at which the components are fired is set using the Simulation speed slider. This simulation mode is convenient for large, pipeline-style systems, and helps the user to see how the events propagate through the system. *Interactive simulation* highlights the components that can be fired at each step. The enabled components are only fired when the user clicks on them (the components with straightforward behaviour, such as consecutive combinational logic blocks in SDFS model, are still fired automatically). This simulation mode is convenient for manual inspection of execution scenarios or for examining the response of the system to certain input stimuli. *Trace replay* mode allows the user to replay either previously saved traces or the violation traces returned by verification tools. In this way, a sequence of events that leads to an undesired state of the system can be examined by the user, which is very helpful for debugging. If Step-by-step option is selected, only one event at a time is fired when the user presses the Step button.

## 4 Use Cases

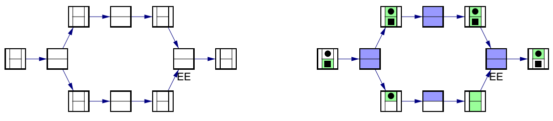
Workcraft has been successfully used for numerous practical applications, some of them employing complex interactions between several different model types. Below we present a number of examples illustrating this.

### 4.1 Static Data Flow Structures Simulation and Verification

Workcraft played a crucial role during the development of the asynchronous circuits datapath models based on Static Data Flow Structures (SDFS) [14], which are a high-level formalism for asynchronous datapaths; SDFS can be viewed as an equivalent of register transfer level (RTL) in synchronous design.

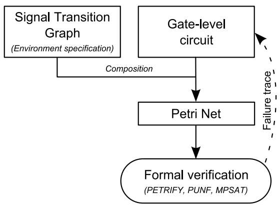
The SDFS model is a directed graph that has two types of nodes: registers (depicted as boxes with two vertical lines) and combinational logic (depicted as plain boxes). Registers can be marked with tokens (shown as black circles or squares), and their movement from register to register models the propagation of data inside the datapath. The logic nodes represent combinational logic and model the effect of its delay on the data pipeline. An essential property of the logic nodes is the possibility of *early evaluation (EE)* – the situation where just a subset of inputs is sufficient to start producing the computation result. In such a case, all the other inputs are no longer required, and it is best to send a signal to terminate their computation in order to save power and time. There are several types of SDFSs capable of expressing datapaths with EE, such as Spread Token, Anti-token and Counterflow [14]. They have a similar graphical notation but different token game rules.

Systems with EE often have very intricate behaviour, and it is very easy to introduce subtle errors when designing them. For example, the shortest trace leading to a deadlock in a (rather small) Counterflow SDFS model in Figure 5 contains 29 steps. This problem would be fairly hard to discover using manual interactive simulation, due to a very long and peculiar sequence of events that leads to a deadlock. Hence formal verification is essential in SDFS design. In this example, Workcraft was able not only to detect a deadlock, but also to

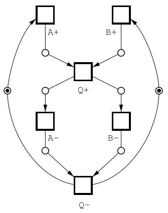


(a) Initial state (b) Deadlock state

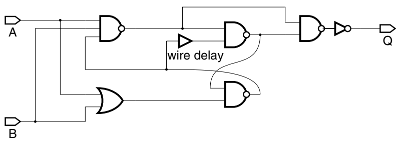
**Fig. 5.** Counterflow SDFS verification example



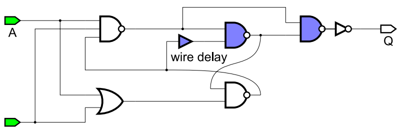
(a) Verification flow



(b) Environment STG



(c) Circuit that implements the specification (a)



(d) A potential hazard state

**Fig. 6.** Circuit verification example

graphically reproduce, step-by-step, the problematic event trace. This has led to a better understanding of the limitations of the Counterflow SDFS model, and provided the motivation and essential ideas for further adjustment of the token game rules.

## 4.2 Asynchronous Circuits Verification

An important problem when designing a gate-level asynchronous circuit is the efficient detection of hazards. However, whether the circuit is hazardous or not almost always depends on the behaviour of its environment, i.e. a circuit can be hazardous in one environment and not hazardous in another. The problem here is that it is usually impractical or even impossible to provide the specification of the environment as a gate-level circuit. In practice, the *abstraction* of the environment is often given in the form of a Signal Transition Graph (STG) [8], which is a special type of a labelled PN. That is, the overall specification is inherently heterogeneous, as a part of it is a gate-level circuit and another part is an STG. Hence WORKCRAFT's ability to interface different formalisms becomes essential for this scenario. It translates the circuit into an STG [11], which is then composed with the environment specification. To alleviate the state space explosion, the verification is performed using efficient external tools based on PN unfolding prefixes, namely PUNF [6] and MPSAT [7]. The shortest PN trace leading to a hazard is then mapped back onto the gate-level representation of the circuit, presenting the designer with a readable trace that can be visually simulated in WORKCRAFT.

An example of circuit verification is shown in Figure 6. The circuit that is verified is a possible implementation of Muller's C-element. If one does not take the wire delays into consideration then the implementation is correct. However, if a wire delay is introduced (modelled by a buffer in parts (c,d) of the figure), the excited AND-gate that follows the wire can become disabled without firing, causing a hazard.

Circuit verification features of WORKCRAFT have helped us to discover a potential hazard in a previously published Counterflow Stage Controller circuit [1,14], as well as assisted in the development of a multiresource arbiter [5].

## 4.3 Asynchronous Circuit Synthesis Based on Conditional Partial Order Graphs

Conditional Partial Order Graph [9] is a formalism for circuit specification which combines certain advantages of Petri nets and Finite State Machines: it does not have the explicit notion of states (like PNs) and models the choice on the level of logic conditions (like FSMs). The specification size of a highly concurrent system with multiple combinational choice is often much smaller in the CPOG model than in PN or FSM one.

CPOGs were implemented in WORKCRAFT, and appear to be the formalism with most links to other model types, see Figure 7. Asynchronous circuits can be synthesised directly from CPOG specifications, and verified for speed-independence using the PN mapping technique [11] (the environment behaviour in this case is specified using STGs). A CPOG model can also be directly converted into a PN, and checked for properties such as deadlocks. PETRIFY tool can be used as an alternative method of synthesising the same specification, so that its result can be compared with that of CPOG-based synthesis, and the user can chose the best one.

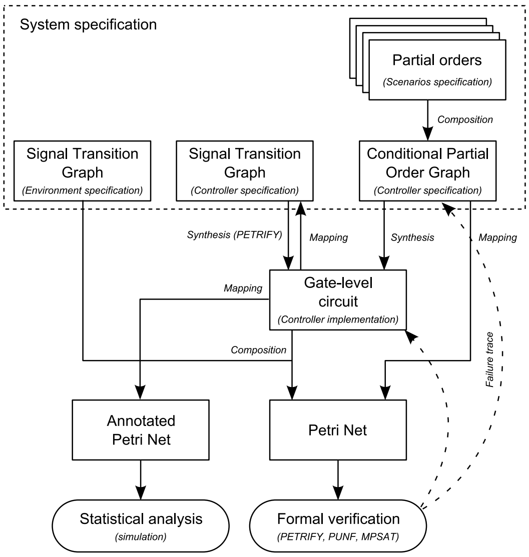


**Fig. 7.** A complex model interoperability example

## 5 Comparison with Other Tools

There is a large number of tools available that are able to manipulate and analyse particular Interpreted Graph Model types. The tool closest to WORKCRAFT with respect to the design philosophy is probably the PEP tool [10]. It is a comprehensive and extendable framework that includes a set of utilities for verification of Petri nets. PEP tool supports a considerable number of models, including process algebrae, high-level and low-level Petri nets; it can also export the models into a variety of formats (SPIN, INA etc.). The only models in PEP that support visual representation are high-level and low-level Petri nets; in particular, there is no support for circuits.

PETRIFY [4] is a command-line utility that generates circuits from STGs. While quite powerful at that task, it lacks a GUI mode to work with circuits or STGs. PETRIFY is able to detect problems in the input STG, such as deadlocks and hazards, but it does not produce violation traces to help the designer pinpoint the problem. It also cannot read circuits and convert them back into STGs, e.g. for the purpose of verification.

VERSIFY [13] is a tool that accepts circuits and can efficiently check if the circuit is speed-independent under a given environment. It is a command-line tool, which makes it hard to browse its output.

WORKCRAFT is different from other tools in that it does not focus on algorithms for a particular IGM type, but aims to provide a common environment that helps to "glue" existing tools together in a consistent manner. For example, WORKCRAFT provides the visualisation and editing functionality for Petri nets, but does not have any internal verification routines. Instead, it relies on externals tools such as PUNF [6] and MPSAT [7] to carry out verification. Then it is able to parse the verification output and present it to the user in a graphical manner. Without much effort, WORKCRAFT could be interfaced with tools like PETRIFY — and benefit from their algorithmic power while at the same time providing them with the user-friendly front-end.

Another notable difference is that while other tools generally support minimal links between the models, such as direct translation from one model to another, WORKCRAFT supports more complicated model interoperability. Different parts of the system can be specified using different formalisms, and then automatically be merged and verified. For example, as explained above, it is often convenient to specify a circuit as a gate netlist and its environment as an STG. Then the verification result (i.e. the violation trace) is propagated back and presented to the user as a trace of the original model, rather than that of the PN to which the model was translated for verification.

In contrast to the version of WORKCRAFT that has been presented earlier in [12] and focused solely on Static Data Flow Structures, the version presented in this paper has its paradigm shifted to general IGMs. As of time of writing, WORKCRAFT supports editing, simulation and verification of the following models: PNs, STGs, gate-level circuits, SDFSs of several types and Conditional Partial Order Graphs. We also plan to add networks of handshake components in near future.

## 6 Availability

The Workcraft tool supports all major OS platforms (Windows, Linux, Mac OS) and is freely available for academic use. The latest version can be downloaded from Workcraft's homepage `http://workcraft.org`.

*Acknowledgments.* The authors would like to thank Andrey Mokhov, Ashur Rafiev, Danil Sokolov and Stanislavs Golubcovs who contributed to the development of Workcraft. This research was supported by the Epsrc grant EP/D053064/1 (Sedate) and Royal Academy of Engineering / Epsrc post-doctoral research fellowship EP/C53400X/1 (Davac).

## References

1. Ampalam, M., Singh, M.: Counterflow pipelining: architectural support for preemption in asynchronous systems using anti-tokens. In: Proc. CAD 2006 (2006)
2. Bardsley, A., Edwards, D.: The Balsa asynchronous circuit synthesis system. In: Forum on Design Languages (2000)
3. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge (1999)
4. Cortadella, J., Kishinevsky, M., Kondratyev, A., Lavagno, L., Yakovlev, A.: Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. IEICE Trans. Inf. and Syst. E80-D(3), 315–325 (1997)
5. Golubcovs, S., Mokhov, A., Yakovlev, A.: Multi-resource Arbiter Design. In: Proc. 20th UK Asynchronous Forum (2008)
6. Khomenko, V.: Model Checking Based on Prefixes of Petri Net Unfoldings. PhD thesis, School of Computing Science, Newcastle University (2003)
7. Khomenko, V., Koutny, M., Yakovlev, A.: Detecting state encoding conflicts in STG unfoldings using SAT. Fundam. Inf. 62(2), 221–241 (2004)
8. Kishinevsky, M.A., Kondratyev, A.Y., Taubin, A.R., Varshavsky, V.I.: On self-timed behavior verification. In: ACM Intl. Workshop on Timing Issues in the Specification and Synthesis of Digital Systems (1992)
9. Mokhov, A., Yakovlev, A.: Conditional Partial Order Graphs and Dynamically Reconfigurable Control Synthesis. In: Proc. DATE 2008 (2008)
10. Pep homepage, `http://theoretica.informatik.uni-oldenburg.de/~pep/`
11. Poliakov, I., Mokhov, A., Rafiev, A., Sokolov, D., Yakovlev, A.: Automated verification of asynchronous circuits using circuit Petri nets. In: Proc. ASYNC 2008, pp. 161–170. IEEE Computer Society, Los Alamitos (2008)
12. Poliakov, I., Sokolov, D., Mokhov, A.: Workcraft: a static data flow structure editing, visualisation and analysis tool. In: Kleijn, J., Yakovlev, A. (eds.) ICATPN 2007. LNCS, vol. 4546, pp. 505–514. Springer, Heidelberg (2007)
13. Roig, O.: Formal Verification and Testing of Asynchronous Circuits. PhD thesis, Universitat Politecnica de Catalunya (1997)
14. Sokolov, D., Poliakov, I., Yakovlev, A.: Asynchronous data path models. In: Proc. ACSD 2007 (2007)

# PDETool: A Multi-formalism Modeling Tool for Discrete-Event Systems Based on SDES Description

Ali Khalili, Amir Jalaly Bidgoly, and Mohammad Abdollahi Azgomi

Performance and Dependability Engineering Lab., Department of Computer Engineering, Iran University of Sceince and Technology, Tehran, Iran
{al_khalili, amir_jalaly}@comp.iust.ac.ir, azgomi@iust.ac.ir

**Abstract.** Discrete-event systems have gained a lot of interest due to their wide range of applications. SDES is a unified description for stochastic discrete-event systems. The aim is to use this description as a basis of a new multi-formalism modeling framework. In this paper, we introduce PDETool, which is based on SDES description. This modeling tool provides features for construction and translation of models into the XML-based input language of an SDES-based simulation engine that is developed for this purpose. Currently, we have implemented some useful extensions of Petri nets, such as stochastic Petri nets (SPNs), stochastic reward nets (SRNs) and stochastic activity networks (SANs) in this framework. PDETool is easily extensible to support a wide range of graphical and non-graphical formalisms. Furthermore, it facilitates the construction, animation and simulation of models. This tool has some advantages over the existing multi-formalism modeling and simulation tools, which will be mentioned in this paper.

**Keywords:** Stochastic discrete-event systems, SDES description, PDETool, Petri nets.

## 1 Introduction

In stochastic discrete-event systems, the events are responsible for changing the state of the system. The behavior of these systems can be described using stochastic models. Discrete-event simulation (DES) of such a model helps the modeler to evaluate some important performance or dependability measures of the corresponding system.

In addition to DES, several formal languages have been used for modeling stochastic discrete-event systems. Stochastic extensions of Petri nets, including stochastic Petri nets (SPNs) [5], stochastic reward nets (SRNs) [3] and stochastic activity networks (SANs) [8], and stochastic extensions of process algebra have widely been used in a wide range of applications. Each formal modeling language depicts a way of describing systems with a specified level of abstraction. In practice, to use a formal modeling language, it is necessary to have a modeling tool. Most of the existing formal modeling languages share some common characteristics. In addition, the analysis methods of these models are same or quite similar.

In the literature, many efforts have been reported to develop multi-formalism modeling tools. SHARPE [7] is probably the first one, which supports multiple models, including Markov models, generalized SPNs (GSPNs), queueing networks, etc. After SHARPE, some other tools have been developed with the same goal. SMART [2], Möbius [4, 10] and OsMoSys [12] are other multi-formalism modeling tools. Although the existing tools provide various capabilities, there are many difficulties to extend them to support new models. Most of the multi-formalism tools can only be extended by their developers.

Stochastic discrete-event systems (SDES) [13] is a unified and abstract description for stochastic discrete-event systems that can be regarded as a blueprint for an abstract data type with virtual elements, which are instantiated for a certain model class by substituting the attributes with net-class dependent values and functions.

In this paper, we introduce PDETool, which is a multi-formalism modeling tool based on SDES. This modeling tool provides features for construction and translation of models into the XML-based input language of an SDES-based simulation engine that we have developed for this purpose.

Currently, we have implemented some useful extensions of Petri nets, such as SPNs, SRNs and SANs in this framework. PDETool is easily extensible to support a wide range of graphical and non-graphical formalisms. Furthermore, it facilitates the construction, animation and simulation of models.

The remainder of this paper is organized as follows. In section 2, the SDES unified description is briefly reviewed. Section 3, describes PDETool, its application and architecture. A case study using SANs is presented in section 4. Finally, some concluding remarks and a list of future works are mentioned in section 5.

## 2 A Brief Introduction to SDES Description

A discrete-event system is a system that is in a state during some time interval, after which an atomic event might happen that changes the state of the system immediately. Several stochastic discrete-event models have been proposed, which all share some common characteristics and many algorithms and methods that have been developed for one model are applicable for many of them. SDES [13], introduced by Zimmermann, is a unified description for stochastic discrete-event systems. Popular model classes like automata, queuing networks, and Petri nets of different kinds with stochastic extensions are subclasses of stochastic discrete-event systems and can be translated into the SDES description.

In [13], a stochastic discrete-event system, *SDES*, is defined as a tuple $SDES = (SV^*, A^*, S^*, RV^*)$, where $SV^*$ describes a *finite set of state variables* and *actions* $A^*$ together with the *sort function* $S^*$ and the *reward variables* $RV^*$ corresponds to the quantitative evaluation of the model.

For more information about SDES unified description, please see [13].

## 3 The PDETool Framework

There exist many modeling and simulation tools, which most of them support only a single simulation or modeling language and a few simulation or solution methods. It is interested to develop a multi-formalism modeling framework to support a wide range of models and easily be extensible to support new formalisms.

In the following subsections, we firstly introduce an SDES-based simulation engine. Then, we review the software architecture and some main functionalities of PDETool.

## 3.1 An SDES-Based Simulation Engine

Before starting to develop PDETool, we developed a simulation engine based on SDES description, named *SimGine*, which is an abbreviation for *simulation engine*. As the best of our knowledge, this is the first engine that has been developed based on the SDES description.

As mentioned before, SDES is a unified description. Many formal modeling languages for stochastic discrete-event systems can be translated to SDES. Therefore, SimGine can be used for simulation of a wide range of formal modeling languages. A model can be simulated by SimGine if a mapping can be provided into the SDES description and thus the input language of the engine.

SimGine uses an XML-based input language whose syntax and semantics are designed to resemble the SDES description. In this input language, each model consists of four parts: (1) *auxiliaries*, for the required functions and constants, (2) *model state variables*, to define the states of the system, (3) *a number of events*, that are responsible for changing the system states, and (4) *some reward structures*, that are used for evaluation of the modeler's interested measures during the evaluation of the model. The most important part of the model is the events part (i.e. $A^*$ in the SDES description), which defines a set of events and everything needed about them. An event represents the basic unit of a model that facilitates changing the state of the system by modifying the values of state variables. For each event, *the event precondition*, *delay*, *priority*, *weight*, *action*, and *reactivation predicate* should be defined in the input language.

After definition of events, rewards must be specified. SimGine supports performance rewards, which include both *rate rewards* and *impulse rewards* [9]. For each reward variable, the type of rewards should also be specified, which indicates the time of reward computation, which can be *transient* or *steady-state*.

The XML-based input language of SimGine defines some methods. The body of these methods can be written in a programming language and can be constructed of common programming structures, such as local variable declarations, repetition or conditional statements, assignment statements, etc. This provides a powerful and flexible way for definition of events with complex behaviors. Currently, C#.Net is used as the programming language of SimGine's input models.

To support more modeling languages, we have partially extended SDES with *event reactivation concept*, which exists in SANs [8]. However, we have assumed that the degree of each activity is one and each action had exactly one variant and therefore, does not distinguish between an action and its modes.

SimGine can easily be used in third-party Microsoft .NET applications as a library or as a stand alone application with a graphical user interface (GUI). To obtain the SimGine and more information about it, please visit the SimGine homepage [11].

## 3.2 The PDETool Software

The aim of developing PDETool is to introduce an SDES-based multi-formalism modelling and simulation framework for construction, simulation and solution of stochastic discrete-event models. The tool has the following components:

- *Model Editor*: Model Editor is a GUI that allows modelers to load, construct, edit and save models. Each formal modeling language, which is supported by PDETool, has its corresponding model editor that can use its own file format and graphical or textual representation. Currently, model editors for SANs, SPNs and SRNs have been developed. For example, *SAN Editor* can be used to construct and edit SAN models. Fig. 1 shows a screenshot of the *SAN Editor*.
- *Reward Variable Editor*: This editor provides a user interface to define and edit reward variables. The tool supports *rate* and *impulse* rewards. Each of these types can be defined as *steady-state*, *instance-of-time*, *average-of-time*, or *interval-of-time*, which specify the interested time of the reward computation.
- *Global Variable Editor*: PDETool allows modelers to define and use global variables and constants. Using this feature, the modeler can define the desired global variables, which should be initialized in a simulation study.
- *Model Animator*: PDETool can animate models. This feature is known as *token-game animation* in tools for Petri nets. Fig. 2 shows a sample view of the model animator for SANs.
- *Model Simulator*: This feature is used for discrete-event simulation of models. To do so, first the model will be translated into the SDES description and then, the SimGine is used to parse the input file and generate the simulation code, and then the simulation model will be run and the reward variables will be evaluated. The information of reward variables can be watched during the simulation progress. This information includes the confidence mean value and the confidence interval and also the variance of each reward variable in the current time.
- *SimGine Interface*: PDETool has an interface for direct working with SimGine. This feature is useful when the modeler likes to simulate a SimGine file. In this case, the modeler can load, edit and simulate a SimGine file. A simple model animator is also available for this purpose.

As mentioned before, SimGine can be used for simulation of stochastic formalisms by a mapping into the input language of the engine. So, PDETool by using this engine can be extended to support these model classes. It currently has the ability to simulate SAN, SPN and SRN models. In the next section, we will briefly describe how the tool can be extended to support a new formalism.

The PDETool software is executable on Microsoft Windows XP and Vista. This tool and its user manual are available for download in PDETool homepage [6].

## 3.3 The Software Architecture of PDETool

The overall software architecture of the tool, including its main modules and their relations, is depicted in Fig. 3. As shown in the figure, the software architecture is composed of two layers: a *front-end layer* and a *back-end layer* corresponding to the PDETool interface and the SimGine, respectively. These layers are described below:

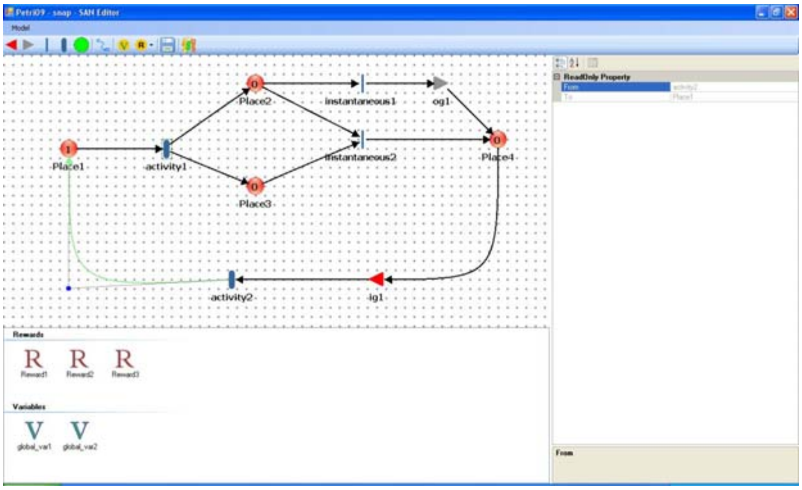
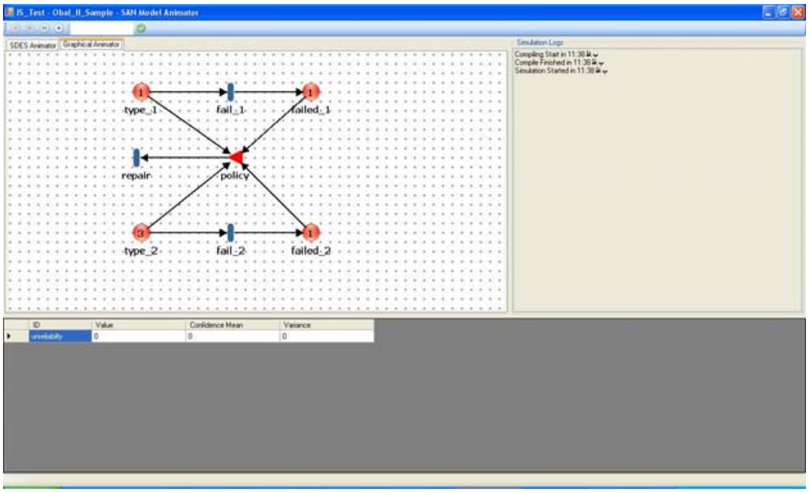


**Fig. 1.** Graphical user interface of *SAN Editor*



Compiling Start in 11:38
Compile Finished in 11:38
Simulation Started in 11:38

| | ID | Value | Confidence Mean | Variance |
|---|---|---|---|---|
| ▸ | unreliability | 0 | 0 | 0 |

**Fig. 2.** Graphical user interface of *SAN Animator*

- *The Front-End Layer*: This layer can be viewed as an interface between the users and the back-end layer. This layer provides the required GUIs and a model translator. The user interfaces includes editors (model editor, reward variable editor, global variable editor), SimGine interface, simulation and animation GUI, as described in the previous subsection. For each supported modeling language, its editor, translator and an optional graphical animator should be provided, which are responsible for communication with the back-end layer.
- *The Back-End Layer*: PDETool uses SimGine as the underlying layer. This back-end layer is responsible for simulation of the model and evaluating reward variables and also provides model animation facility. As we mentioned before, for simulating a model based on particular stochastic discrete-event formalism, the

model translator (in the front-end layer) prepares the input model of the simulation engine by mapping the original model (constructed in the PDETool interface) into the input language of SimGine. As depicted in Fig. 3, the internal architecture of SimGine is consists of parser, code generator, utility library, simulator, animator and SDES simulation manager.

## 3.4 How to Extend PDETool to Support a New Formalism

PDETool is extensible to support new stochastic formal modeling languages. It uses an object-oriented architecture, which helps the developers to easily extend the front-end layer of the tool that has communications with SimGine. For this purpose, PDETool provides some basic abstract classes that should be extended using inheritance property and overriding the required methods. The class diagram of the PDETool class hierarchy and their relationships with SimGine are depicted in Fig. 4.

*GraphicalModel* and *TextualModel* are two basic main classes that are used to represent an individual graphical and textual model. They have the required abstract methods to work with a particular formalism (such as, saving, loading and validating models). For graphical models, each graphical element of the model should be implemented as an extension of *BaseGraphicalObject* class, which includes the necessary properties and methods, such as displaying the element or its validation (e.g. validation of its connections with other elements or properties assigned to the element). Construction of a model is performed in a GUI (window) that is inherited from *TextualEditorGUI* or *GraphicalEditorGUI*, which has the capabilities of presentation and edition of textual and graphical models, respectively.

*ModelSimulationGUI* is the base class for handling the simulation process (i.e. starting, pausing, displaying reward variables during simulation, etc.). The simulation progress can also be animated by using the SimGine animation event handlers defined in SimGine *SDESAnimator* class. An optional GUI animator based on *ModelAnimationGUI* can be developed for this purpose.

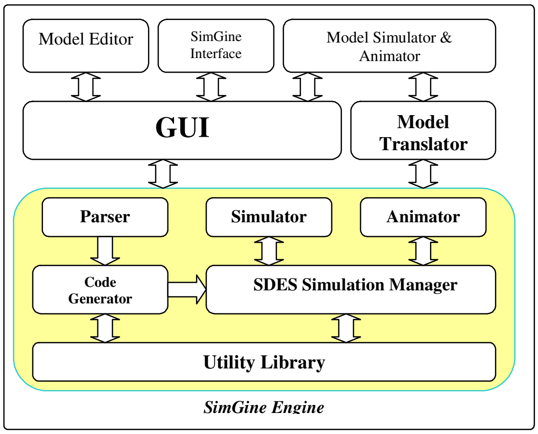


**Fig. 3.** The software architecture of PDETool

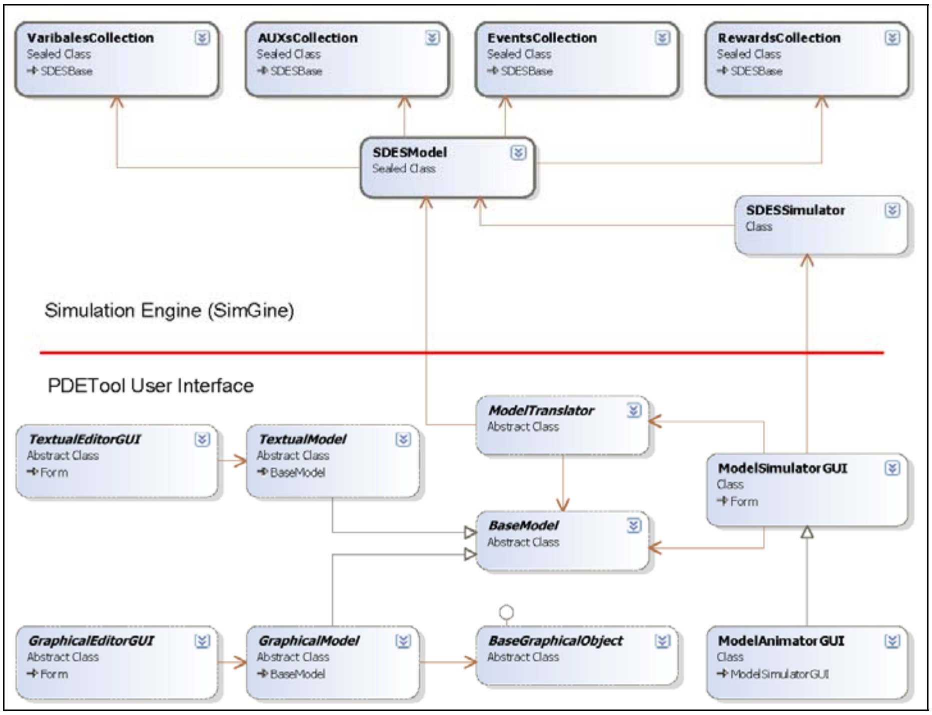


**Fig. 4.** The class diagram of base classes and their relations with SimGine

A translator class inherited from the abstract class *ModelTranslator* should also be developed to translate this particular formalism into the SDES description based on the SimGine input language, which can be simulated by the engine.

### 3.5 Comparison

By utilizing SimGine as an SDES-based simulation engine, PDETool provides features for definition and translation of models into the XML-based input language of SimGine and it can easily be extended to support a wide range of formalisms.

Comparing to other modeling tools and frameworks, PDETool and its underlying layer, SimGine, can easily be used as a simulation and modeling engine in third party applications. For example, using SimGine classes and methods, an application can simulate stochastic discrete-event models by translating them into the input language of SimGine. SDES unified description is simple and understandable, which makes SimGine to be an easy-to-use simulation engine.

## 4 A Case Study Using SANs

Modeling with a particular formalism using PDETool can be considered from both *developers' viewpoint* and *modelers' viewpoint*. In this section, we review the implementation of SANs in the PDETool framework (i.e. developers' viewpoint). Then, we take a glance at simulation of a simple model using the tool (i.e. modelers' viewpoint).

*SANModel* class, inherited from the *Model* class, uses a PNML-based file format for SANs, called SANML [1], to load and save SAN models. For construction and edition of SAN models, *SANModelEditor* is developed based on *ModelEditorGUI* class. This editor contains a reward and global variable editor too. Note that the reward and global variable editor exists in *ModelEditorGUI* and is accessible for all graphical editors, including the *SANModelEditor*. *SANAnimationGUI* is a class obligated to animate SAN models (i.e. token-game animation). Note that the graphical model animation is an optional facility which can be provided for a model type. *SANTranslator* class, inherited from the *ModelTranslator*, is responsible for translation of SAN models into the SimGine input language. This translation is performed by conversion of the model stored as an object of type *SANModel*. In a nutshell, the tool translates the input gate predicates of an activity into precondition of its corresponding event and the related input/output gate functions into the action method of that event. Places and global variables are translated into the state variables.

From the modelers' viewpoint, as an example, consider a computer system with two processors, in which one of them is faster than the other. Tasks enter the system based on a Poisson process and route to a processor for execution. The fast processor has priority over the slow one for processor allocation. If both processors are busy, the tasks wait in a bounded buffer with a capacity equal to three. If this buffer is full, the tasks will be rejected. The necessary service time of different tasks is exponentially distributed.

A SAN model for this system, which is presented in Fig. 5, has been constructed by PDETool. The gate table of the model is also presented in Table 1.

The aim of simulation is to compute the average number of tokens in place *SlowAlloc* and place *FastAlloc* (i.e. $E(Mark_{SlowAlloc})$ and $E(Mark_{FastAlloc})$), and the probability of being *0* and *2* tokens inside *Jobs* (i.e. $P(Mark_{Jobs}=0)$ and $P(Mark_{Jobs}=2)$). The simulation results of the model with confidence level *99%* are presented in Table 2.

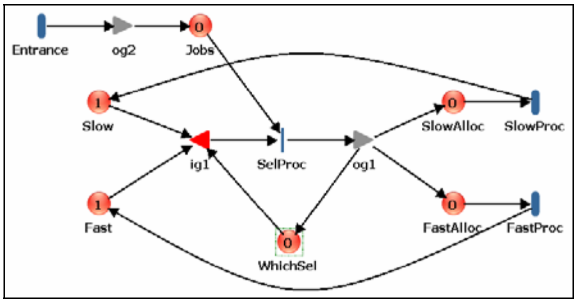


**Fig. 5.** An example of a computer system with two processors modeled in SANs

**Table 1.** Gate table for the model of Fig. 5

| Gate name | Input predicate | Gate function |
|---|---|---|
| *ig1* | if (Fast.Mark != 0<br>  \|\| Slow.Mark != 0)<br>  return true;<br>else<br>  return false; | if (Fast.Mark != 0) {<br>  Fast.Mark = 0;<br>  WhichSel.Mark = 0;<br>}<br>else {<br>  Slow.Mark = 0;<br>  WhichSel.Mark = 1;<br>} |
| *og1* | n/a | if (WhichSel.Mark == 0) FastAlloc.Mark = 1;<br>else SlowAlloc.Mark = 1; |
| *og2* | n/a | if (Jobs.Mark < 3) Jobs.Mark++; |

**Table 2.** The simulation results

| Measure | Value | Confidence | Variance |
|---|---|---|---|
| $E(Mark_{SlowAlloc})$ | 0.409990 | 0.008849 | 0.241898 |
| $E(Mark_{FastAlloc})$ | 0.484920 | 0.008992 | 0.249772 |
| $P(Mark_{Jobs}=0)$ | 0.877465 | 0.005769 | 0.107519 |
| $P(Mark_{Jobs}=2)$ | 0.035370 | 0.003323 | 0.034119 |

## 5 Conclusions

Discrete-event simulation is a popular solution for performance and dependability evaluation of systems. In this paper, we introduced PDETool as a new multi-formalism modeling and simulation framework which uses a simulation engine based on SDES unified description called SimGine. Providing some features for constructing and translating models into the XML-based input language of SimGine, PDETool can easily be extended to support wide range of formalisms. Currently, it can be used for modeling and simulation of some useful extensions of Petri nets such as SPNs, SRNs and SANs. The tool and its underlying layer, SimGine, can also be used as a library in other applications.

We are currently working to improve the tool in several ways:

- *Analytic solution*: An important step in further developments of the framework is to implement state-space generators and a solution engine including various analytic steady-state and transient solvers for Markovian models.
- *Hierarchical modeling*: Based on the SDES description, PDETool currently supports only flat models. We intend to implement capabilities for hierarchical model construction, animation and simulation.
- *Fast simulation*: As a separate project, we have extensively worked on fast simulation techniques of rare events. We intend to implement techniques like importance sampling and partition-of-the-region in the framework.
- *Model checking*: We intend to extend the PDETool framework with a model checking engine.
- *Implementation of more formal languages*: By now, we have implemented some useful extensions of Petri nets and an actor-based formalism in the framework. We intend to implement more models, including process algebra, etc.

## References

1. Abdollahi Azgomi, M., Movaghar, A.: An Interchange Format for Stochastic Activity Networks Based on PNML Definition. In: Proc. of the ICATPN'04 Satellite Workshop on the Definition, Implementation and Application of a Standard Interchange Format for Petri Nets (XML4PN 2004), Italy, pp. 1–10 (2004)
2. Ciardo, G., Miner, A.S.: SMART: Simulation and Markovian analyzer for reliability and timing. In: Proc. of the 2nd Int'l Computer Performance and Dependability Symposium (IPDS 1996), p. 60 (1996)
3. Ciardo, G., Mupalla, J., Trivedi, K.S.: Analyzing Concurrent and Fault-Tolerant Software Using Stochastic Reward Nets. Journal of Parallel and Distributed Systems 15, 252–269 (1992)
4. Deavours, D.D.: Formal Specification of the Möbius Modeling Framework. Ph.D. Dissertation, University of Illinois (2001)
5. Molloy, M.K.: Performance Analysis Using Stochastic Petri Nets. IEEE Trans. on Comp. C-31, 913–917 (1982)
6. PDETool Framework Homepage, `http://pdel.iust.ac.ir/Projects/PDETool.html`
7. Sahner, R.A.: Combinatorial-Markov methods of solving performance and reliability models. Ph.D. dissertation, Duke University, Durham, North Carolina (1986)
8. Sanders, W.H., Meyer, J.F.: Stochastic Activity Networks: Formal Definitions and Concepts. In: Brinksma, E., Hermanns, H., Katoen, J.-P. (eds.) EEF School 2000 and FMPA 2000. LNCS, vol. 2090, pp. 315–343. Springer, Heidelberg (2001)
9. Sanders, W.H., Meyer, J.F.: A Unified Approach for Specifying Measures of Performance, Dependability and Performability. In: Dependable Computing for Critical Applications, Dependable Computing and Fault-Tolerant Systems, vol. 4, pp. 215–237. Springer, Heidelberg (1991)
10. Sanders, W.H.: Integrated Frameworks for Multi-Level and Multi-Formalism Modeling. In: Proc. of the 8th Int'l. Workshop on Petri Nets and Performance Models (PNPM 1999), Zaragoza, Spain (1999)
11. SimGine Homepage, `http://pdel.iust.ac.ir/Projects/SimGine.html`
12. Vittorini, V., et al.: The OsMoSys approach to multi-formalism modeling of systems. Software and Systems Modeling 3(1), 68–81 (2004)
13. Zimmermann, A.: Stochastic Discrete-Event Systems: Modeling, Evaluation and Applications. Springer, Heidelberg (2008)

# Author Index