Workshop Proceedings

2nd Workshop on

Application of Region Theory (ART)

Newcastle upon Tyne, UK, June 21, 2011

Satellite event of the conferences

11th International Conference on

Application of Concurrency to System Design (ACSD 2011)

32nd International Conference on

Application and Theory of Petri Nets (PETRI NETS 2011)

Edited by Jörg Desel and Alex Yakovlev

Preface

Regions have been defined about 20 years ago by Andrzej Ehrenfeucht and Grzegorz Rozenberg as sets of nodes of a finite transition system that correspond to potential conditions that enable or disable transition occurrences in a corresponding elementary net system. Thus, regions have been the essential concept for synthesis of an elementary net systems from its anonymous state graph (states are unknown but transitions between states are known). Since that time, many generalizations and variants of the synthesis problem of Petri nets from behavioral descriptions have been studied, including synthesis of more general Petri net classes, synthesis from languages, synthesis from partially ordered runs and synthesis from incomplete behavioral descriptions. All this work has in common that the transition names are given more or less directly by the behavioral description. The places of the net to be synthesized always correspond to regions which are defined in many different ways, depending on the form of the behavioral description. A main issue in this research is the study of regions, whence we call the entire research direction region theory.

Region Theory was applied in many different areas such as

- hardware synthesis from precise specifications (synthesis from transition systems)
- visualization of concurrent hardware behavior (synthesis from logic circuit models, transition systems and partial orders)
- GALS synthesis and desynchronisation based on synthesis (synthesis from step transition systems and re-synthesis from Petri nets)
- synthesis of control and policies for discrete event systems (synthesis from both languages and transition systems)
- modelling biological (membrane) systems with localities (synthesis from step transition systems)
- generation of specifications from incomplete specifications (mining from transition systems)
- model generation from examples (specification from (partial) languages)
- mining of process descriptions (mining from languages)

The aim of the ART workshop series was to bring together people working in these or other application areas of region theory, to exchange ideas and concepts and to work on common workshop results.

This proceedings volume contains reviewed contributions submitted to and presented at the 2nd ART workshop.

Jörg Desel (Hagen, Germany)
Alex Yakovlev (Newcastle University, UK)
June 2011

Table of Content

Jetty Kleijn, Maciej Koutny, Marta Pietkiewicz-Koutny and Grzegorz Rozenberg

Classifying Boolean Nets for Region-based Synthesis.. 5

Josep Carmona

The Label Splitting Problem.. 22

Boudewijn van Dongen, Jörg Desel and Wil van der Aalst

Aggregating Causal Runs into Workflow Nets.. 36

Robin Bergenthum and Sebastian Mauser

Folding Partially Ordered Runs... 52

Philippe Darondeau and Laurie Ricker

Towards Distributed Control of Discrete-Event Systems.................................. 63

Robin Bergenthum and Sebastian Mauser

Mining with User Interaction... 79

Wil van der Aalst

Do Petri Nets Provide the Right Representational Bias for Process Mining? ... 85

Programme Committee

Josep Carmona, UPC Barcelona, Spain

Philippe Darondeau, INRIA Rennes, France

Jörg Desel, FernUniversität in Hagen, Germany (co-chair)

Boudewijn van Dongen, TU Eindhoven, The Netherlands

Luís Gomes, Universidade Nova de Lisboa, Portugal

Gabriel Juhás, Slovak University of Technology, Slovak Republic

Jetty Kleijn, Leiden University, The Netherlands

Alex Kondratyev, Cadence Design Systems Inc., Berkeley CA, USA

Luciano Lavagno, Politecnico di Torino, Italy

Robert Lorenz, Universität Augsburg, Germany

Marta Pietkiewicz-Koutny, Newcastle University, UK

Grzegorz Rozenberg, Leiden University, The Netherlands

Akex Yakovlev, Newcastle University, UK (co-chair)

Mengchu Zhou, New Jersey Institute of Technology, USA

Classifying Boolean Nets for Region-based Synthesis

Jetty Kleijn[1], Maciej Koutny[2],
Marta Pietkiewicz-Koutny[2], and Grzegorz Rozenberg[1,3]

[1] LIACS, Leiden University, The Netherlands
`{kleijn,rozenber}@liacs.nl`
[2] School of Computing Science, Newcastle University, UK
`{maciej.koutny,marta.koutny}@ncl.ac.uk`
[3] Department of Computer Science, University of Colorado at Boulder, U.S.A.

Abstract. A Petri net model is referred to as Boolean if the only possible markings are sets, i.e., places are marked or not without further quantification; moreover, also the enabling conditions and firing rule are based on this principle of set-based token arithmetic. Elementary Net systems are an example of a class of Boolean nets, and so are the recently introduced SET-nets. In our investigation of the synthesis problem for SET-nets, it would be useful to know how this new net model can be fitted into the general theory of net synthesis based on the generic concept of τ-nets. Here, we demonstrate how SET-nets and the idea of Boolean operations on tokens provide an opportunity to classify a wide variety of Boolean nets that are amenable to region-based net synthesis.
Keywords: Petri net, Boolean net, step semantics, concurrency, conflict, net synthesis, theory of regions, transition system, NET-type

1 Introduction

Recently, in [10], a new class of Petri nets, called SET-nets, has been introduced to provide a net based computational model matching very closely the computations exhibited by reaction systems [5, 7, 8], a framework for the investigation of processes carried out by biochemical reactions in living cells. The formalisation leading to reaction systems has been motivated by properties that are common to many biochemical reactions. This has resulted in a model based on principles that are different from most existing models of computation. Of particular importance for the net model inspired by reaction systems, are the non-counting features (motivated by the two main regulation mechanisms of facilitation and inhibition) implying that entities are either present or not present and enable reactions by their presence or absence. As a consequence, there is no conflict between reactions in the sense that the occurrence of one reaction might imply that another reaction which is also enabled at the current state, cannot occur.

The new class of Petri nets, SET-nets, provides a faithful computational model matching very closely that exhibited by reaction systems. The key difference between standard Petri nets like Place/Transition nets (PT-nets) and SET-nets is that the former support multiset-based token arithmetic, whereas the latter support set-based (or Boolean) operations on tokens.

Thus, the computational intuition embedded in bio-processes has led to a new class of nets with yet to discover properties. On the other hand, an important motivation behind the wish to establish the link with net theory is to establish whether Petri net based concepts (such as causal processes) and methods (such as synthesis of nets from a specification of their behaviour) could be used to provide analytical tools for reaction systems.

The research presented in this paper originates with the synthesis problem for SET-nets. We show that the new class of nets can be treated within the general theory of region-based net synthesis. More precisely, we show that SET-nets are an instance of τ-nets [1] which incorporate many Petri net classes, and for which the synthesis problem has been investigated and solved using regions of transition systems [6]. Moreover, τ-nets with maximally concurrent semantics (the semantics of SET-nets when used to model reaction systems) fall within the general framework of τ-nets with *policies* introduced in [2]. In this paper, we will actually concern ourselves with the wider task of dealing with a whole variety of net models similar to SET-nets and referred to as Boolean nets. It is our aim to classify such nets thus working towards automatic net synthesis algorithms. The key part of our investigation is a detailed study of connection monoids (CONN-monoids) for Boolean nets which allows us to capture not only the step semantics of nets but also structural conflicts between transitions in Boolean nets, thanks to a special 'blocking' connection which can be used to capture the essence of conflicts in (Boolean) τ-nets. In this way, CONN-monoids emerge as a single formalism which can be used to deal with conflicts, concurrency and net synthesis.

The paper is organised as follows. First, we present the basics of SET-nets and other types of nets. Section 3 recalls the general setup of [1, 2] in which Petri net classes are defined using CONN-monoids and τ-nets. Section 4 shows how to build CONN-monoids for EN-systems and SET-nets. The observations in this section are generalised in Section 5 to Boolean τ-nets.

2 SET-nets, EN-systems, and PT-nets

In this section we present SET-nets and relate them to elementary net systems (EN-systems) [13] and Place/Transition nets (PT-nets) [4].

The main idea underlying SET-nets is that there is no concept of token counting. Places are marked or not marked and arcs have no weights. In this way, SET-nets resemble EN-systems, a fundamental net model in the study of basic features of concurrent systems. However, the execution semantics of the two models differ significantly.

Both SET-nets and EN-systems have an (unweighted) net as their underlying structure. A *net* is a triple (P, T, F) such that P and T are disjoint finite sets of *places* and *transitions*, respectively, and $F \subseteq (T \times P) \cup (P \times T)$ is the *flow* of the net. We use the standard dot-notation: for any place or transition x, we let ${}^\bullet x = \{y \mid (y,x) \in F\}$ be its set of input elements and $x^\bullet = \{y \mid (x,y) \in F\}$ its output elements. This extends in the usual way to sets of places and/or transitions. For EN-systems, we have the additional structural assumption that the underlying net has no 'self-loops' i.e., ${}^\bullet t \cap t^\bullet = \varnothing$ for all $t \in T$.

A *marking* of a SET-net or EN-system is a subset of places of their underlying net. A place belonging to a given marking is said to be marked. In diagrams, places are drawn as circles and transitions as rectangles. If $(x,y) \in F$, then (x,y) is an *arc* leading from *node* x to *node* y. Markings are indicated by drawing in each place belonging to a given marking, a small black dot (a 'token').

A SET-*net* is a tuple $N = (P,T,F,M_0)$ such that (P,T,F) is a net and $M_0 \subseteq P$ is its *initial* marking. A EN-*system* is a tuple $N = (P,T,F,M_0)$ such that (P,T,F) is a net without self-loops and $M_0 \subseteq P$ is its *initial* marking.

The dynamics of SET-nets is defined as follows. Let $N = (P,T,F,M_0)$ be a SET-net and let $t \in T$. Then t is *enabled* at a marking M if ${}^\bullet t \subseteq M$. In such a case, t can occur (*fire*), leading to the marking $M' = (M \setminus {}^\bullet t) \cup t^\bullet$. A subset U of T, a *step*, is *enabled* at M if ${}^\bullet U \subseteq M$. If U is enabled, it can occur, leading to $M' = (M \setminus {}^\bullet U) \cup U^\bullet$.

Hence in a SET-net, a step U is enabled whenever each of its input places belongs to the current marking, in other words, each of its elements is enabled. When U occurs, its input places lose their token, while all output places will be marked. If a place is both input and output for U, it is marked before and after the occurrence of U. Furthermore, output places of U that were marked before its occurrence will remain marked. It is also worthwhile to observe that there may be distinct transitions t, $u \in U$ for which ${}^\bullet t \cap {}^\bullet u \neq \varnothing$ or $t^\bullet \cap u^\bullet \neq \varnothing$. This has no effect on their participation in the occurrence of U.

The dynamics of EN-systems is defined in a similar way, except that the enabling conditions are crucially different. Let $N = (P,T,F,M_0)$ be an EN-system and let $t \in T$. Then t is *enabled* at a marking M if ${}^\bullet t \subseteq M$ $\underline{\text{and}}$ $t^\bullet \cap M = \varnothing$. If t is enabled at M, it can fire which results in the marking $M' = (M \setminus {}^\bullet t) \cup t^\bullet$. A step U of T is *enabled* at M if each $t \in U$ is enabled at M $\underline{\text{and}}$ $({}^\bullet t \cup t^\bullet) \cap ({}^\bullet u \cap u^\bullet) = \varnothing$ for any two distinct transitions $t, u \in U$. Then U can occur leading to $M' = (M \setminus {}^\bullet U) \cup U^\bullet$.

Hence, in an EN-system, if a step U is enabled at marking M then each of its input places is marked $\underline{\text{and}}$ none of its output places is marked. Actually, a step can only ever be enabled if the input/output neighbourhood of the transitions in U do not overlap (i.e., if there is no *structural conflict* in U).

In SET-nets and EN-systems markings are sets and tokens are manipulated using set-based rather than multiset-based arithmetic. We will refer to such Petri net models as being Boolean. In contrast to both SET-nets and EN-systems, and

Boolean nets in general, Place/Transition nets (PT-nets) have a multiset-based arithmetic. [4]

A (*weighted*) PT-*net* is specified as a tuple $N = (P, T, W, M_0)$, where P and T are, as before, finite, disjoint sets of places and transitions; $W : (P \times T) \cup (T \times P) \to \mathbb{N}$ specifies the arcs of N by their weights; and $M_0 : P \to \mathbb{N}$ is the initial marking. In general, for PT-nets, markings are multisets, rather than sets. In diagrams, whenever $W(x, y) \geq 1$ for some $(x, y) \in (T \times P) \cup (P \times T)$, then (x, y) is an arc from x to y; it is annotated with $W(x, y)$ if this is 2 or more. Given a marking M of N and a place $p \in P$, we say that $M(p)$ is the number of tokens in p. Note that the dot-notations are now multisets, indicating the multiplicity of each input/output element.

A transition t of N is *enabled* at a marking M of N, if ${}^\bullet t \leq M$. If t is enabled at M, it can fire which leads to the new marking $M' = M - {}^\bullet t + t^\bullet$. Thus M' is obtained from M by deleting $W(p, t)$ tokens from each place p and adding $W(t, p)$ tokens to each place p. A step of a PT-net N is a multiset of transitions. Step U is *enabled* at a marking M of N if $\sum_{t \in T} U(t) \cdot {}^\bullet t \leq M$. Thus, in order for U to be enabled at M, for each place p, the number of tokens in p under M should at least be equal to the accumulated number of tokens needed as input to each of the transitions in U, respecting their multiplicities in U. If U is enabled at M, it may occur which leads to $M' = M - \sum_{t \in T} U(t) \cdot {}^\bullet t + \sum_{t \in T} U(t) \cdot t^\bullet$. Thus the effect of executing U is the accumulated effect of executing each of its transitions (taking into account their multiplicities in U). Note that there is no concept of structural conflict in the class of PT-nets: transitions may occur together in a step whenever a marking supplies enough tokens.

Inhibitor and activator arcs. To each of the above net models we can add *inhibitor arcs* and *activator arcs* connecting places to transitions, by adding relations *Inh* and *Act* to their specification. Given the set of places P and set of transitions T of a SET-net, or EN-system, or PT-net, $Inh, Act \subseteq P \times T$ define its set of *inhibitor* or *activator* arcs, respectively. For each transition $t \in T$, we denote ${}^\circ t = \{p \mid (p, t) \in Inh\}$ for the set of inhibitor places of t and ${}^\blacklozenge t = \{p \mid (p, t) \in Act\}$ for its activator places. (Both notions are extended to sets of transitions, and to multisets, disregarding multiplicities.)

The intuition behind these *context arcs* is that in order for a transition to be enabled at a marking, its activator places should be marked (have at least one token) and its inhibitor places should not be marked (have no token). Thus the dynamics of these extended net classes is adapted in the following way: a step U is *enabled* at a marking when it is enabled in the underlying SET-net, or EN-system, or PT-net, and ${}^\blacklozenge U \subseteq M$ and ${}^\circ U \cap M = \varnothing$. When U is enabled at M and it occurs, then the resulting marking is defined as before (here the activator and inhibitor arcs have no effect). Note that this semantics is an *a priori semantics* (see, e.g., [12])

[4] A multiset μ over a set X is a function $\mu : X \to \mathbb{N} = \{0, 1, 2, \ldots\}$; such multiset may be represented by listing its elements with repetitions. Sets can be considered as multisets without repetitions.

3 Connections and connection monoids

Now we are ready to recall the general setup of [1, 2] in which Petri net classes are defined on the basis of individual connections between places and transitions. Moreover, the effect of the simultaneous execution of a *step* (a set or multiset of transitions) on a given place is calculated using a dedicated commutative monoid which returns the composite connection between that place and the step. For Boolean nets, we will assume that each step is a set of transitions rather than a multiset as in [1, 2]. This simplifies the presentation and is harmless as Boolean Petri nets as we consider them here, would not allow true multiset steps anyway.

Connection monoids describe the relation between a place and a step. A *connection monoid* (or CONN-monoid) is a set $\mathbb{S}$ of *connections* with a commutative and associative binary composition operation $\oplus$, and a neutral element (identity) $\mathbf{0}$. We will use the same symbol $\mathbb{S}$ for a CONN-monoid and for its underlying set of connections. Moreover, for each $s \in \mathbb{S}$ we let $\bigoplus^0 s = \mathbf{0}$ and $\bigoplus^{n+1} s = (\bigoplus^n s) \bigoplus s$ for all $n \in \mathbb{N}$.

Let $\mathbb{S}$ be a CONN-monoid. Then, a NET-*type over* $\mathbb{S}$ is a transition system $\tau = (Q, \mathbb{S}, \Delta)$ where Q is a set of *states*, and $\Delta : Q \times \mathbb{S} \to Q$ is a partial function such that $\Delta(q, \mathbf{0}) = q$, for all $q \in Q$. For every state q, the set $\mathit{enbld}_\tau(q) = \{s \mid \Delta(q, s) \text{ is defined}\}$ consists of all connections from $\mathbb{S}$ that are *enabled* at q.

As an example let us consider the CONN-monoid $\mathbb{S}_{PT} = (\mathbb{N} \times \mathbb{N}, \oplus, \mathbf{0})$ with $\mathbf{0} = (0, 0)$ and point-wise arithmetic addition $\oplus$. Using this monoid, the connections between places and multisets of transitions in PT-nets can be expressed through the NET-type $\tau_{PT} = (\mathbb{N}, \mathbb{S}_{PT}, \Delta_{PT})$ over $\mathbb{S}_{PT}$, where $\Delta_{PT} = \{(n, (m, k)) \mapsto n - m + k \mid n \geq m \geq 0\}$. Intuitively, this states that a place containing n tokens enables steps which take no more than n tokens, and that the resulting number of tokens is $n-m+k$ where m and k are the numbers of tokens taken and produced, respectively, by all occurrences of transitions in that step together. A fragment of τ_{PT}, interpreted as a labelled directed graph, is shown in Figure 1(d).

In general, each NET-type $\tau = (Q, \mathbb{S}, \Delta)$ defines a class of nets, the so-called τ-*nets*. The NET-type specifies through Q the values that can be assigned to places; through the connections in $\mathbb{S}$ the effect of combining connections; and through Δ, the enabling conditions and newly generated values.

A τ-*net* is a tuple $N = (P, T, F, M_0)$, where P and T are, respectively, disjoint finite sets of places and transitions, $F : (P \times T) \to \mathbb{S}$ is a *connection mapping*, and M_0 is the *initial marking* of N (in general, a marking is a mapping from P to Q). For a place p of N and a step U of transitions, we define the composite connection between U and p by $F(p, U) = \bigoplus_{t \in T}(\bigoplus^{U(t)} (F(p, t))$. Thus, if U is a set, then $F(p, U) = \bigoplus_{t \in U} F(p, t)$ and $F(p, \varnothing) = \bigoplus_{t \in \varnothing} F(p, t) = \mathbf{0}$.

A step U is *(resource) enabled* at a marking M if $F(p, U) \in \mathit{enbld}_\tau(M(p))$ for every place $p \in P$. The *firing* of such a step produces the marking M' such that $M'(p) = \Delta(M(p), F(p, U))$, for every place $p \in P$. The *concurrent reachability graph* $\mathit{CRG}(N)$ of N is formed by firing inductively from M_0 all possible (resource) enabled steps of N.

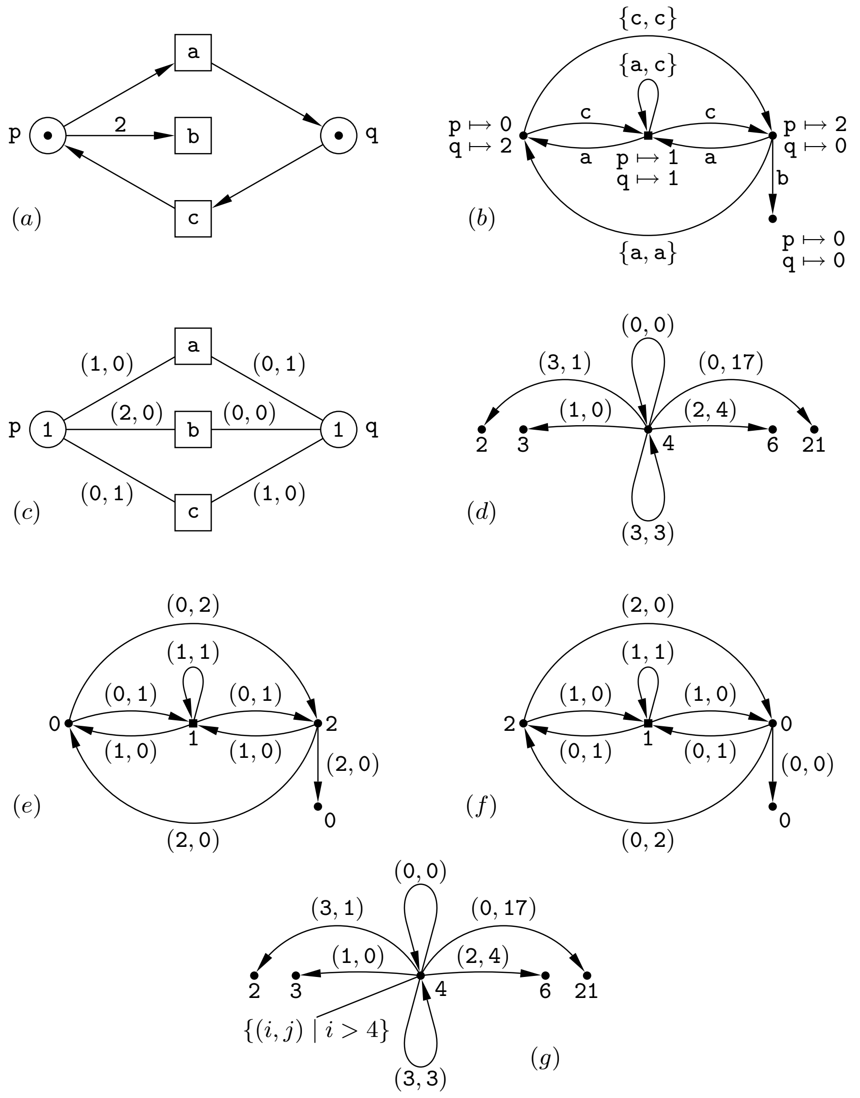


Fig. 1. A PT-net (a); its concurrent reachability graph (b) with the initial state represented by a small square; and its rendering as a τ_{PT}-net system (c). A fragment of the NET-type τ_{PT} is shown in (d). In (e) and (f) we re-trace in (b) the behaviour of places p and q, respectively, in terms of the NET-type τ_{PT}. An alternative graphical representation of the NET-type τ_{PT} is shown in (g).

The NET-type τ_{PT} defines τ_{PT}-nets. In order to view a PT-net as a τ_{PT}-net, all one needs to do is to associate integers, representing the number of tokens, with each place, and set $F(p,t) = (W(p,t), W(t,p))$, for all places p and transitions t. The CONN-monoid $\mathbb{S}_{PT}$ together with the transition system τ_{PT}

provides accurate information about the enabling and firing of steps U. Indeed, all one needs to do is to calculate $F(p, U) = (\mathit{inwgt}, \mathit{outwgt})$ using the monoid operation of point-wise addition (for all input and output weights of all multiple occurrences of transitions in U).

As an illustration, let us consider the PT-net depicted in Figure 1(a). This PT-net is represented by the τ_{PT}-net in Figure 1(c). Notice that, in particular, $F(\mathtt{q}, \mathtt{b}) = (\mathtt{0}, \mathtt{0})$ means that q and b in Figure 1(a) are disconnected. Also the markings are represented (by indicating the number of tokens by an appropriate integer 0, 1, 2, etc.). Figure 1(b) gives the concurrent reachability graph. We furthermore obtain $F(\mathtt{p}, \{\mathtt{a}, \mathtt{c}\}) = (\mathtt{1}, \mathtt{0}) \oplus (\mathtt{0}, \mathtt{1}) = (\mathtt{1}, \mathtt{1})$ and $F(\mathtt{q}, \{\mathtt{a}, \mathtt{c}\}) = (\mathtt{0}, \mathtt{1}) \oplus (\mathtt{1}, \mathtt{0}) = (\mathtt{1}, \mathtt{1})$ which, together with $\Delta_{PT}(\mathtt{1}, (\mathtt{1}, \mathtt{1})) = \mathtt{1}$, means that: (i) the net in Figure 1(a) enables the step $\{\mathtt{a}, \mathtt{c}\}$ at the initial marking at which both p and q have one token; and (ii) its firing results in the same marking. On the other hand the step $\{c, c\}$ is not enabled at the initial marking because $F(\mathtt{q}, \{\mathtt{c}, \mathtt{c}\}) = (\mathtt{1}, \mathtt{0}) \oplus (\mathtt{1}, \mathtt{0}) = (2, \mathtt{0})$ and $(2, \mathtt{0})$ is not enabled at 1. However, it is enabled at the marking M with $M(q) = 2$ and $M(p) = 0$ and then its firing results in the marking M' with $M'(q) = 0$ and $M'(p) = 2$. Now focus in the concurrent reachability graph in Figure 1(b) on the local markings of the place p in combination with the connections that lead to changes of those local markings. We can do this by labelling each state with the corresponding marking of p, and each arc with the cumulative arc weight w.r.t. p of the step U labelling that arc i.e., $F(\mathtt{p}, U)$. The result is shown in Figure 1(e). Repeating the same procedure for the place q, yields Figure 1(f). Note that both graphs in Figure 1(e) and (f) can also be seen in the graph of the NET-type τ_{PT}, see Figure 1(d).

4 Connection monoids for en-systems and set-nets

Starting from EN-systems, we will now present a number of specific classes of Boolean nets defined on basis of their place-transition connections. In what follows we describe the structure of the connection monoids by a Cayley table displaying the outcome of all possible combinations of connections.

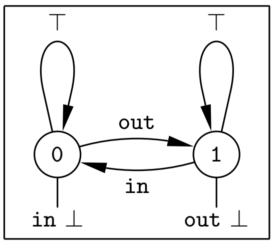
EN-systems
In EN-systems, there are three basic connections between places and transitions:

– $F(p, t) = \top$	p and t are disconnected (independent)	$\textcircled{p} \quad \boxed{t}$
– $F(p, t) = \mathtt{in}$	there is an arc from p to t	$\textcircled{p} \longrightarrow \boxed{t}$
– $F(p, t) = \mathtt{out}$	there is an arc from t to p	$\textcircled{p} \longleftarrow \boxed{t}$

Figure 2(a) depicts τ_{EN}, the NET-type showing how the connections between a place and a transition in an EN-system determine the enabledness of the transition w.r.t. that place and the resulting marking if it fires. In the diagram, 0 and 1 mean that the place is respectively *empty* (i.e., not marked) and *full* (marked). Thus, if the place p is marked and there is an arc from p to the transition t ($F(p, t) = \mathtt{in}$), then t may fire as far as p is concerned and the effect will be that p is empty after the occurrence of t. If p and t are not connected, t may always occur from the point of view of p and its occurrence has no effect on the marking of p. There is also an explicit reference to $F(p,t) = \texttt{in}$ for the case that p is empty and one to $F(p,t) = \texttt{out}$ when p is full. In these cases the marking of p prohibits the enabledness of t. In addition to the three standard types of connections, τ_{EN} has a special 'blocking' connection $\perp$ which does not label any arc (is never enabled), hence $\perp \notin \mathit{enbld}_{\tau_{EN}}(\texttt{0}) \cup \mathit{enbld}_{\tau_{EN}}(\texttt{1})$. The connection $\perp$ is also used to capture *structural conflict* between transitions. As such it is a convenient device to capture precisely those steps which are not allowed, because of the internal conflicting relations between their transitions w.r.t. the place.

The CONN-monoid $\mathbb{S}_{EN} = (\{\top, \texttt{out}, \texttt{in}, \perp\}, \oplus_{EN}, \top)$. is defined through its Cayley table in Figure 2(b). Here $\texttt{out} \oplus_{EN} \texttt{out} = \texttt{out} \oplus_{EN} \texttt{in} = \texttt{in} \oplus_{EN} \texttt{out} = \texttt{in} \oplus_{EN} \texttt{in} = \perp$ corresponds directly to the requirement that the neighbourhoods of transitions in a step must be disjoint for it to be enabled to occur.

For example, if we have two transitions, t and u, both removing tokens from place p, $\boxed{t} \leftarrow \textcircled{p} \rightarrow \boxed{u}$, thus both have p as an input place, then the connection of the step $\{t,u\}$ w.r.t. p is calculated as $F(p,\{t,u\}) = F(p,t) \oplus_{EN} F(p,u) = \texttt{in} \oplus_{EN} \texttt{in} = \perp$ implying that $\{t,u\}$ can never occur together (on account of p).



(a)

$\oplus_{EN}$	$\top$	out	in	$\perp$
$\top$	$\top$			
out	out	$\perp$		
in	in	$\perp$	$\perp$	
$\perp$	$\perp$	$\perp$	$\perp$	$\perp$

(b)

$\oplus_{EN}$	out	in
out	$\perp$	
in	$\perp$	$\perp$

(c)

Fig. 2. NET-type τ_{EN}, and the Cayley table of $\mathbb{S}_{EN}$.

In all CONN-monoids , $\top$ will be the identity element and — if present — $\perp$ is the absorbing element. The monoid $\mathbb{S}_{EN}$ is the most restrictive monoid over $\top$, in, out, and $\perp$, because its operation does not yield any non-$\perp$ results except when $\top$ is involved. This is clearly seen in Figure 2(c) which depicts the non-trivial part of the Cayley table from Figure 2(b), while omitting the values implicitly due to commutativity. In what follows we will present CONN-monoids using a minimal presentation of their Cayley table as in Figure 2(c).

A *basis* of a CONN-monoid is any irreducible subset of its non-$\perp$ connections such that any other non-$\perp$ connection can be derived from it.

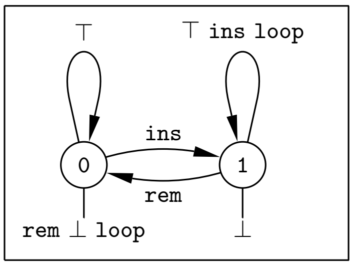
Proposition 1. $\{\top, \texttt{in}, \texttt{out}\}$ *is the only basis of* $\mathbb{S}_{EN}$.

Proof. Follows directly from the table in Figure 2(c). □

SET-nets
Now there are four basic connections between places and transitions:

- $F(p,t) = \top$ p and t are disconnected (independent) $\textcircled{p} \quad \boxed{t}$
- $F(p,t) = \texttt{rem}$ there is an arc from p to t $\textcircled{p} \dashrightarrow \boxed{t}$
- $F(p,t) = \texttt{ins}$ there is an arc from t to p $\textcircled{p} \dashleftarrow \boxed{t}$
- $F(p,t) = \texttt{loop}$ there is an arc from t to p, and from p to t $\textcircled{p} \rightleftarrows \boxed{t}$

Figure 3(a) depicts τ_{SN}. Comparing Figure 3(a) and Figure 2(a) brings to light the important difference between the meaning of an arc from a transition to a place in EN-systems (connection `out`) and the meaning of an arc from a transition to a place in SET-nets (connection `ins`).



(a)

$\oplus_{SN}$	ins	rem	loop
ins	ins		
rem	loop	rem	
loop	loop	loop	loop

(b)

Fig. 3. NET-type τ_{SN}, and the simplified table of $\mathbb{S}_{SN} = (\{\top, \texttt{ins}, \texttt{rem}, \texttt{loop}\}, \oplus_{SN}, \top)$.

Figure 4 shows the `out`-labelled arc in τ_{EN} and the `ins`-labelled arc in τ_{SN}.

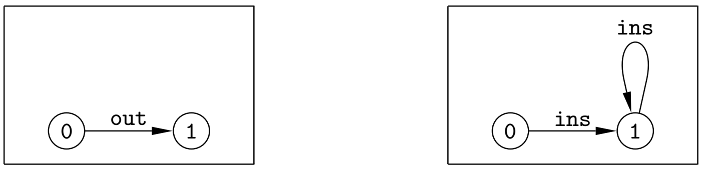


Fig. 4. Difference between arcs from transitions to places in EN-systems and SET-nets.

The simplified Cayley table of the CONN-monoid $\mathbb{S}_{SN}$ is shown in Figure 3(b). From the table we see, e.g., that if p is an output place of a transition t and input place to u, $\boxed{t} \dashrightarrow \textcircled{p} \dashrightarrow \boxed{u}$, then the connection of the step $\{t, u\}$ w.r.t. p is given by $F(p, \{t, u\}) = F(p, t) \oplus_{SN} F(p, u) = \texttt{ins} \oplus_{SN} \texttt{rem} = \texttt{loop}$ and so, as far as p is concerned, $\{t, u\}$ can occur if p contains a token; moreover, p will also have a token after the occurrence of $\{t, u\}$.

Another important property of the $\mathbb{S}_{SN}$ monoid is the idempotence of its operation (the diagonal of the Cayley table of $\mathbb{S}_{SN}$). This reflects one of the main features of SET-nets, namely that since resources are not quantified, they can be used by many transitions with the same connectivity in tandem, as though they were just one such transition. Note furthermore, that since SET-nets know no structural conflict, $\bot$ is not introduced through $\oplus_{SN}$. Consequently, $\bot$ is not necessary in the case of τ_{SN} and $\mathbb{S}_{SN}$. Actually, also $\mathbb{S}_{PN}$ did not need $\bot$, as PT-nets know no structural conflicts either. However, if we consider the class of k-bounded PT-nets then the situation is rather different as the corresponding CONN-monoid is defined as $\mathbb{S}_{BPT} = (\{\bot\} \cup \mathbb{N}_k \times \mathbb{N}_k, +_k, (\mathtt{0}, \mathtt{0}))$, where $\mathbb{N}_k = \{0, 1, \ldots, k\}$, $\bot$ is the absorbing element, and, for all $n, m, n', m' \in \mathbb{N}_k$,

$$(n,m) +_k (n',m') = \begin{cases} (n+n', m+m') & \text{if } n+n' \leq k \ \wedge \ m+m' \leq k \\ \bot & \text{otherwise .} \end{cases}$$

Proposition 2. $\{\top, \mathtt{ins}, \mathtt{rem}\}$ *is the only basis of* $\mathbb{S}_{SN}$.

Proof. Follows directly from the table in Figure 3(b). □

This insight forms a formal justification of the way in which τ_{SN}-nets are drawn: direct dashed arrows are used for `ins` and `rem`, but `loop` as a 'compound' connection can be depicted by the 'compound' representation for `ins` and `rem`. Note that in the $\bot$-less version of $\mathbb{S}_{SN}$, `loop` is the absorbing element, but this will change when we add inhibitor arcs. First however, we add inhibitor arcs to EN-systems.

EN-systems with inhibitor arcs

In comparison with EN-systems, we now have one more connection to take into account:

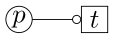
– $F(p,t) = \mathtt{inh}$ there is an inhibitor arc from p to t 

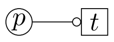
Figure 5 shows the NET-type τ_{ENI}, and the simplified Cayley table of the CONN-monoid $\mathbb{S}_{ENI} = (\{\top, \mathtt{out}, \mathtt{in}, \mathtt{inh}, \bot\}, \oplus_{ENI}, \top)$. From this we see that the monoid $\mathbb{S}_{ENI}$ captures an additional type of structural conflict: $\mathtt{in} \oplus_{ENI} \mathtt{inh} = \mathtt{inh} \oplus_{ENI} \mathtt{in} = \bot$. That $\mathtt{out} \oplus_{ENI} \mathtt{inh} = \mathtt{inh} \oplus_{ENI} \mathtt{out} = \mathtt{out}$ is a consequence of the a priori semantics.

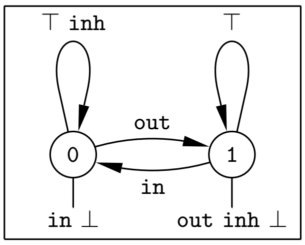
Proposition 3. $\{\top, \mathtt{in}, \mathtt{out}, \mathtt{inh}\}$ *is the only basis of* $\mathbb{S}_{ENI}$.

Proof. Follows directly from the table in Figure 5(b). □

SET-nets with inhibitor arcs

Again, we have to cater for one additional connection:

– $F(p,t) = \mathtt{inh}$ there is an inhibitor arc from p to t 



(*a*)

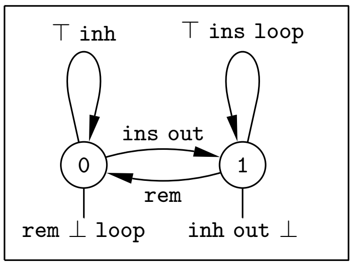
$\oplus_{ENI}$	out	in	inh
out	$\perp$		
in	$\perp$	$\perp$	
inh	out	$\perp$	inh

(*b*)

Fig. 5. NET-type τ_{ENI} (a), and the simplified Cayley table of $\mathbb{S}_{ENI}$ (b).

Figure 6 shows the NET-type τ_{SNI}, and the simplified Cayley table of the CONN-monoid $\mathbb{S}_{SNI} = (\{\top, \texttt{ins}, \texttt{rem}, \texttt{loop}, \texttt{inh}, \texttt{out}, \perp\}, \oplus_{SNI}, \top)$. In this case we do need $\perp$ as structural conflicts occur when inhibitors are combined with consumption (exactly as in EN-systems). Thus $\mathbb{S}_{ENI}$ captures a conflict: $\texttt{rem} \oplus_{SNI} \texttt{inh} = \texttt{inh} \oplus_{SNI} \texttt{rem} = \perp$.

Furthermore, the monoid must be closed w.r.t. its operation and so due to the a priori step semantics for SNI-nets, `out` had to be added as a new connection to describe $\texttt{ins} \oplus_{SNI} \texttt{inh}$. Notice that `out`, although on its own has the same meaning here as in EN-systems and ENI-systems, it is understood differently when combined with other connections. An example is $\texttt{out} \oplus_{SNI} \texttt{out} = \perp$ rather than $\texttt{out} \oplus_{ENI} \texttt{out} = \texttt{out}$ since the step semantics of SNI-nets is different from that of ENI-systems.



(*a*)

$\oplus_{SNI}$	ins	rem	inh	loop	out
ins	ins				
rem	loop	rem			
inh	out	$\perp$	inh		
loop	loop	loop	$\perp$	loop	
out	out	$\perp$	out	$\perp$	out

(*b*)

Fig. 6. NET-type τ_{SNI} (a), and the simplified Cayley table of $\mathbb{S}_{SNI}$ (b).

Proposition 4. $\{\top, \texttt{ins}, \texttt{rem}, \texttt{inh}\}$ *is the only basis of* $\mathbb{S}_{SNI}$.

Proof. Follows directly from the table in Figure 6(b). □

Like for $\mathbb{S}_{SNI}$, also the operation of $\mathbb{S}_{SNI}$ is idempotent. Even stronger:

Proposition 5. *Let $T \neq \varnothing$ be a set of transitions and $\mathbb{T} = \{F(p,t) \mid t \in T\}$.*

$$F(p,T) = \begin{cases} \top & \textit{if } \ \mathbb{T} = \{\top\} \\ \texttt{ins} & \textit{if } \ \mathbb{T} \subseteq \{\texttt{ins}, \top\} \quad \wedge\ \texttt{ins} \in \mathbb{T} \\ \texttt{rem} & \textit{if } \ \mathbb{T} \subseteq \{\texttt{rem}, \top\} \quad \wedge\ \texttt{rem} \in \mathbb{T} \\ \texttt{inh} & \textit{if } \ \mathbb{T} \subseteq \{\texttt{inh}, \top\} \quad \wedge\ \texttt{inh} \in \mathbb{T} \\ \texttt{out} & \textit{if } \ \mathbb{T} \subseteq \{\texttt{inh}, \texttt{ins}, \texttt{out}, \top\} \wedge (\texttt{out} \in \mathbb{T} \ \vee \{\texttt{inh}, \texttt{ins}\} \subseteq \mathbb{T}) \\ \texttt{loop} & \textit{if } \ \mathbb{T} \subseteq \{\texttt{ins}, \texttt{rem}, \texttt{loop}, \top\} \wedge (\texttt{loop} \in \mathbb{T} \vee \{\texttt{rem}, \texttt{ins}\} \subseteq \mathbb{T}) \\ \bot & \textit{otherwise}\ . \end{cases}$$

Proof. Follows directly from the table in Figure 6(b). The table shows that $\oplus_{SNI}$ is idempotent and that $\{\texttt{ins}, \texttt{rem}, \texttt{inh}, \top\}$ is $\mathbb{S}_{SNI}$'s basis. $\square$

EN-systems with inhibitor and activator arcs

The last connection we consider is:

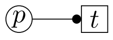
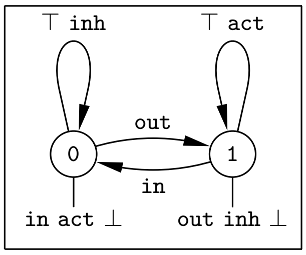
– $F(p,t) = \texttt{act}$ there is an activator arc from p to t 

Figure 7 shows the NET-type τ_{ENC}, and the simplified Cayley table of the CONN-monoid $\mathbb{S}_{ENC}$ for ENC-systems (with the a priori step semantics).



(a)

$\oplus_{ENC}$	out	in	inh	act
out	$\bot$			
in	$\bot$	$\bot$		
inh	out	$\bot$	inh	
act	$\bot$	in	$\bot$	act

(b)

Fig. 7. NET-type τ_{ENC} (a), and the simplified Cayley table of $\mathbb{S}_{ENC}$ (b).

In the simplified Cayley table of $\mathbb{S}_{ENC}$, we see that $\texttt{inh} \oplus_{ENC} \texttt{out} = \texttt{out}$ and $\texttt{act} \oplus_{ENC} \texttt{in} = \texttt{in}$. These pairs of connections reflect that while the transitions involved are enabled with respect to the given place (which should be empty for the `inh` and `out` connections, and marked for the `act` and `in` connections) they affect it in a different way. The connections that induce a state change (`out` and `in`) are 'stronger' , while the connections designated for testing (`inh` and `act`) are 'weaker'.

Proposition 6. $\{\top, \texttt{out}, \texttt{in}, \texttt{inh}, \texttt{act}\}$ *is the only basis of* $\mathbb{S}_{ENC}$.

Proof. Follows directly from the table in Figure 7(b). □

SET-nets with inhibitor and activator arcs
Again we add an activator connection:

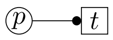
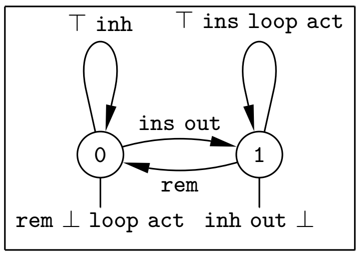
– $F(p,t) = \texttt{act}$ there is an activator arc from p to t 

Figure 8 shows the NET-type τ_{SNC}, and the simplified Cayley table of the CONN-monoid $\mathbb{S}_{SNC} = (\{\top, \texttt{ins}, \texttt{rem}, \texttt{inh}, \texttt{act}, \texttt{loop}, \texttt{out}, \bot\}, \oplus_{SNC}, \top)$ for SET-nets with inhibitor and activator arcs (under the a priori step semantics).



(a)

$\oplus_{SNC}$	ins	rem	inh	act	loop	out
ins	ins					
rem	loop	rem				
inh	out	$\bot$	inh			
act	loop	rem	$\bot$	act		
loop	loop	loop	$\bot$	loop	loop	
out	out	$\bot$	out	$\bot$	$\bot$	out

(b)

Fig. 8. NET-type τ_{SNC} (a), and the simplified Cayley table of $\mathbb{S}_{SNC}$ (b).

Proposition 7. $\{\top, \texttt{ins}, \texttt{rem}, \texttt{inh}, \texttt{act}\}$ *is the only basis of* $\mathbb{S}_{SNI}$.

Proof. Follows directly from the table in Figure 8(b). □

There are 9 different patterns for the various connections: from `0` (unmarked) and from `1` (marked), either to `0` or to `1`, or undefined; see Figure 9.

In each CONN-monoid considered before, different connections had different topological patterns in the associated NET-type. This changes now, as in τ_{SNC} both `loop` and `act` give rise to the same pattern. The effect of combining `act` and `loop` however is `loop` rather than `act`. This is because, according to the step semantics of SET-nets, adding tokens happens after removing or testing. So, in this combination, `loop` as a connection that induces a change of the state is 'stronger' than `act`. Another interesting pair in the table is formed by `act` and `rem`. The effect of composing them is `rem` which differs from `in` due to the different underlying step semantics even though the arc pattern of `rem` in τ_{SNC} and that of `in` in τ_{ENC} are the same.

Thus we arrive at the crucial point in our considerations where it becomes clear that the sophisticated (and sometimes surprising) nature of different connections necessarily involves algebraic properties in addition to topological ones.

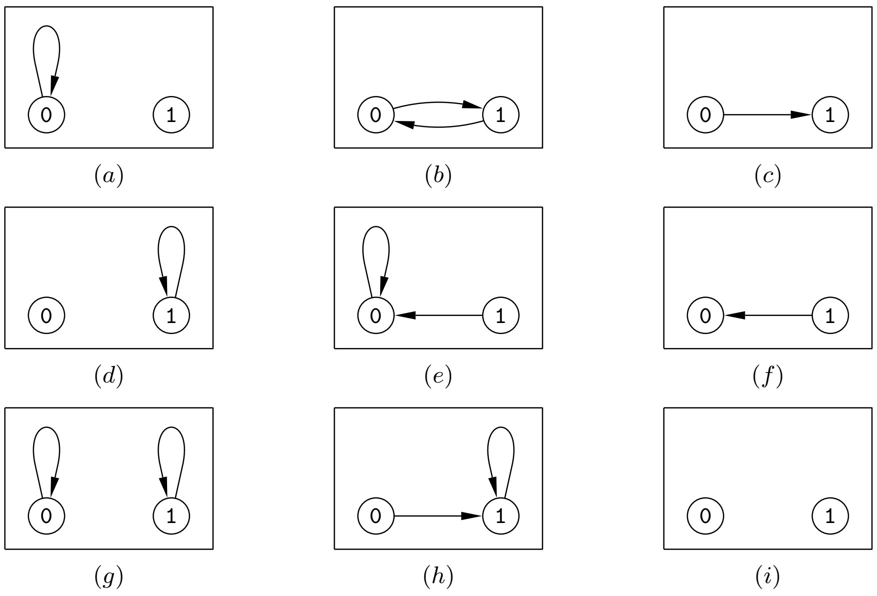


Fig. 9. Topological patterns of connections in Boolean τ-nets.

5 General Boolean τ-nets

We now propose a general classification of *all* possible connections in Boolean τ-nets. We take the general view that each connection defines enabling and effect, and has an associated strength (weak or strong).

Weak connections impose constraints only with respect to enabling, but unlike strong connections, they do not impose constraints on the state resulting from the transition firing. So, when combined with a transition with a stronger connection, it is the latter that dictates the final result. For example, `loop` is strong in $\mathbb{S}_{SN}$ as it 'finishes' by adding a token in operational sense, as this is supported by the algebraic property of absorbtion. `rem`, on the other hand, is weak in $\mathbb{S}_{SN}$ as with this connection the enabling conditions are important, but the state of the net place (connected in this way to some transition) after the transitions fired may be changed by another transition (removing tokens or testing precedes token insertion). This leads to 25 different connections ∂_{xy} where $x, y \in \{\mathtt{w}, \mathtt{s}, \overline{\mathtt{w}}, \overline{\mathtt{s}}, \mathtt{n}\}$. Here x refers to arrows outgoing from `0`, and y refers to arrows outgoing from `1`; `w` means a weak arrow, `s` a strong arrow, and `n` no arrow (non-enabledness); finally, $\overline{(.)}$ implies changing the state (from `0` to `1` or vice versa). In particular, we have the following encoding of the previously discussed connections:

$\perp$	$\top$	`in`	`out`	`ins`	`rem`	`loop`	`inh`	`act`
$\partial_{\mathtt{nn}}$	$\partial_{\mathtt{ww}}$	$\partial_{\mathtt{n}\overline{\mathtt{s}}}$	$\partial_{\overline{\mathtt{s}}\mathtt{n}}$	$\partial_{\overline{\mathtt{s}}\mathtt{s}}$	$\partial_{\mathtt{n}\overline{\mathtt{w}}}$	$\partial_{\mathtt{ns}}$	$\partial_{\mathtt{wn}}$	$\partial_{\mathtt{nw}}$

The corresponding topological patterns are shown in Figure 10.

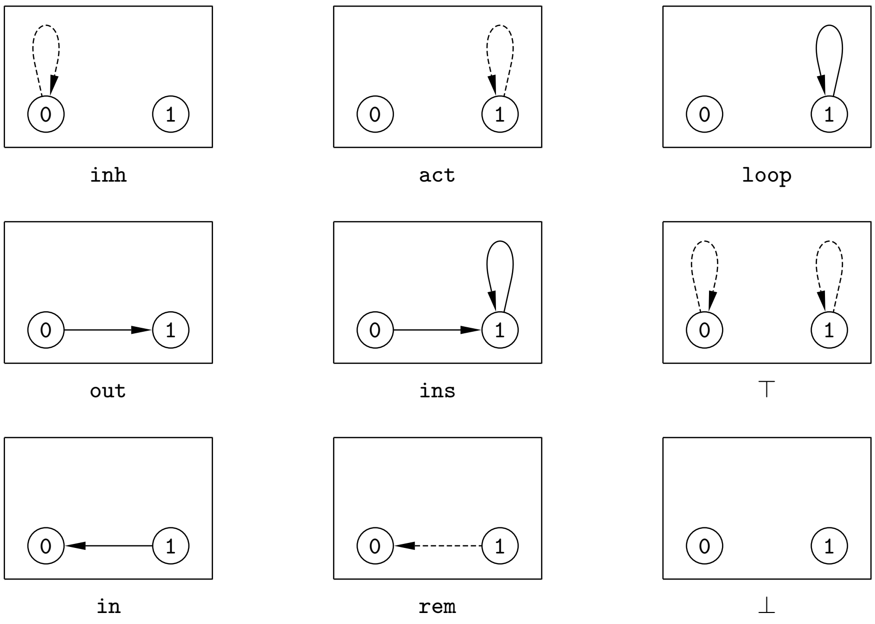


Fig. 10. Connections in Boolean τ-nets with weak arcs indicated by dashed lines.

We will now formalise in a general way algebraic operations on the 25 connections. Almost all will be motivated by enabledness, and the idea of weak and strong. In this way, associativity will be automatic in most cases. Let $\partial_{xy} \oplus \partial_{x'y'} = \partial_{x \odot x'\ y \odot y'}$. where $\odot$ is a commutative operation given by:

$\odot$	$\mathtt{w}$	$\overline{\mathtt{w}}$	$\mathtt{s}$	$\overline{\mathtt{s}}$	$\mathtt{n}$
$\mathtt{w}$	$\mathtt{w}$				
$\overline{\mathtt{w}}$	$\overline{\mathtt{w}}$	$\overline{\mathtt{w}}$			
$\mathtt{s}$	$\mathtt{s}$	$\mathtt{s}$	$\mathtt{s}$		
$\overline{\mathtt{s}}$	$\overline{\mathtt{s}}$	$\overline{\mathtt{s}}$	$\mathtt{n}$	$\overline{\mathtt{s}}$	
$\mathtt{n}$	$\mathtt{n}$	$\mathtt{n}$	$\mathtt{n}$	$\mathtt{n}$	$\mathtt{n}$

The intuition behind, for example, $\partial_{\mathtt{ss}}$ is that the transition is always enabled but its execution keeps the marking in the place unchanged.

Proposition 8. $\mathbb{S}_{conn} = (\{\mathtt{w}, \mathtt{s}, \overline{\mathtt{w}}, \overline{\mathtt{s}}, \mathtt{n}\}, \odot, \mathtt{w})$ *is a commutative monoid.*

Proof. We need to show that $(a \odot b) \odot c = a \odot (b \odot c)$ for all $a, b, c \in \{\mathtt{w}, \mathtt{s}, \overline{\mathtt{w}}, \overline{\mathtt{s}}, \mathtt{n}\}$. To start with, if $\mathtt{n} \in \{a, b, c\}$ then $(a \odot b) \odot c = \mathtt{n} = a \odot (b \odot c)$. Otherwise, we observe that the following hold:

– If $\mathtt{s} \in \{a, b, c\}$ and $\overline{\mathtt{s}} \notin \{a, b, c\}$ then $(a \odot b) \odot c = \mathtt{s} = a \odot (b \odot c)$.
– If $\overline{\mathtt{s}} \in \{a, b, c\}$ and $\mathtt{s} \notin \{a, b, c\}$ then $(a \odot b) \odot c = \overline{\mathtt{s}} = a \odot (b \odot c)$.

$\oplus_{ENC}$	`out`	`in`	`inh`	`act`
`out`	$\perp$			
`in`	$\perp$	$\perp$		
`inh`	`out`	$\perp$	`inh`	
`act`	$\perp$	`in`	$\perp$	`act`

$\oplus_{ENC}$	`out`	`in`	`inh`	`act`
`out`	$\perp$			
`in`	$\perp$	$\perp$		
`inh`	$\perp$	$\perp$	`inh`	
`act`	$\perp$	$\perp$	$\perp$	`act`

Fig. 11. a-priori (a) and a-posteriori (b) semantics for ENC-systems.

– If $\overline{\mathtt{s}} \in \{a,b,c\}$ and $\mathtt{s} \in \{a,b,c\}$ then $(a \odot b) \odot c = \mathtt{n} = a \odot (b \odot c)$.
– If $a,b,c \in \{\mathtt{w},\overline{\mathtt{w}}\}$ and $\overline{\mathtt{w}} \in \{a,b,c\}$ then $(a \odot b) \odot c = \overline{\mathtt{w}} = a \odot (b \odot c)$.
– If $a = b = c = \mathtt{w}$ then $(a \odot b) \odot c = \mathtt{w} = a \odot (b \odot c)$. □

Theorem 1. $\mathbb{S}_{bool} = (\{\partial_{xy} \mid x,y \in \{\mathtt{w},\mathtt{s},\overline{\mathtt{w}},\overline{\mathtt{s}},\mathtt{n}\}\}, \oplus, \partial_{\mathtt{ww}})$ *is a* CONN*-monoid.*

Proof. Follows from Proposition 8 and $\partial_{xy} \oplus \partial_{x'y'} = \partial_{x \odot x' \; y \odot y'}$. □

$\mathbb{S}_{SN}$, $\mathbb{S}_{SNI}$ and $\mathbb{S}_{SNC}$ are all sub-monoids of $\mathbb{S}_{bool}$. The $\mathbb{S}_{SN}$ sub-monoid is a special one; it is *non-blocking* as composing connections never yields $\perp$.

We can now formally describe Boolean net models as those classes of nets that are defined by a net type over (a submonoid of) $\mathbb{S}_{bool}$. From [2], it follows that thanks to the interpretation of the step semantics in term of monoids, Boolean nets are instances of τ-nets for which there exists a region-based solution to the synthesis problem. Moreover, τ-nets with maximally concurrent semantics (the semantics of SET-nets when used to model reaction systems) fall within the general framework of τ-nets with *policies* introduced in [2].

Finally, we should point out that in order to arrive at this general classification of Boolean nets, we have had to make some (slightly arbitrary) assumptions when the intended operational meaning of a combination was not clear. In particular, $\mathtt{w} \odot \overline{\mathtt{w}}$ has been defined in such a way as to give more priority to the change of state. This and perhaps other assumptions are not cast in stone. With differently motivated models in mind, one may freely modify them, study, appreciate the differences. Also, we decided to define $\mathtt{s} \odot \overline{\mathtt{s}}$ as $\mathtt{n}$ because $\mathtt{s}$ and $\overline{\mathtt{s}}$ are both strong and so 'uncompromising': one changes the state whereas the other insists on preserving the state. This contradiction cannot be reconciled.

A posteriori vs. a priori execution semantics Interestingly, CONN-monoids can distinguish between the 'a priori' semantics defined at the end of Section 2, and the 'a posteriori' execution semantics. In the setting of EN-systems, 'a posteriori' is exactly the same as 'a priori' with one extra condition for an enabled set of transitions: ${}^\bullet U \cap {}^\blacklozenge U = U^\bullet \cap {}^\circ U = \varnothing$. Figure 11 exhibits this difference.

6 Conclusions

The reader might wonder why we included in our presentation PT-nets which are clearly non-Boolean nets. Apart from certain didactic motivations, we thought that PT-nets come with a 'calculus of connections' based on a simpler monoid of natural numbers. To our initial surprise, a similar effect can be achieved in our symbolic setting where the monoid of ∂_{xy} connections is completely determined by a simpler monoid with the $\odot$ operation. This could, perhaps, suggest a general approach for constructing practical implementations of synthesis algorithms for SET-nets.

Note that there are variations of Petri nets, such as Boolean Petri nets, where adding a token to an already marked place does not add another token [3, 9]. Also, behaviour of this kind was mentioned in [1] in the context of net synthesis. Having said that, the semantics considered in prior works known to us was based on single transition firings, rather than steps as is the case for SET-nets.

Acknowledgement We would like to thank the anonymous reviewers for their suggestions and comments. This research was supported by the Pascal Chair award from Leiden University and the EPSRC VERDAD project.

References

1. Badouel, E., Darondeau, Ph.: Theory of Regions. In: Part I of [14] (1998) 529–586
2. Darondeau, P., Koutny, M., Pietkiewicz-Koutny, M., Yakovlev, A.: Synthesis of Nets with Step Firing Policies. Fundamenta Informaticae **94** (2009) 275–303
3. De Bra, P., Houben, G.J., Kornatzky, Y.: A Formal Approach to Analyzing the Browsing Semantics of Hypertext. Proc. CSN-94 Conference (1994) 78–89
4. Desel, J., Reisig, W.: Place/Transition Petri Nets. In: Part I of [14] (1998) 122–173
5. Ehrenfeucht, A., Main, M., Rozenberg, G.: Combinatorics of Life and Death for Reaction Systems. International Journal of Foundations of Computer Science **22** (2009) 345–356
6. Ehrenfeucht, A., Rozenberg, G.: Partial 2-structures; Part I: Basic Notions and Representation Problem, and Part II: State Spaces of Concurrent Systems. Acta Informatica **27** (1990) 315–368
7. Ehrenfeucht, A., Rozenberg, G.: Reaction Systems. Fundamenta Informaticae **76** (2006) 1–18
8. Ehrenfeucht, A., Rozenberg, G.: Events and Modules in Reaction Systems. Theoretical Computer Science **376** (2007) 3–16
9. Heiner, M., Gilbert, D., Donaldson, R.: Petri Nets for Systems and Synthetic Biology. Lecture Notes in Computer Science **5016** (2008) 215–264
10. Kleijn, J., Koutny, M., Rozenberg, G.: Modelling Reaction Systems with Petri Nets. Technical Report CS-1244. Newcastle University (2011)
11. Koutny, M., Pietkiewicz-Koutny, M.: Synthesis of Elementary Net Systems with Context Arcs and Localities. Fundamenta Informaticae **88** (2008) 307–328
12. Pietkiewicz-Koutny, M.: The Synthesis Problem for Elementary Net Systems with Inhibitor Arcs. Fundamenta Informaticae **40** (1999) 251–283
13. Rozenberg, G., Engelfriet, J.: Elementary Net Systems. In: Part I of [14] (1998) 12–122
14. Reisig, W., Rozenberg, G. (eds.): Lectures on Petri Nets. Lecture Notes in Computer Science **1491, 1492** (1998)

The Label Splitting Problem

J. Carmona

Universitat Politècnica de Catalunya, Spain
jcarmona@lsi.upc.edu

Abstract. The theory of regions was introduced by Ehrenfeucht and Rozenberg in the early nineties to explain how to derive (synthesize) an event-based model from an automaton. To be applicable, the theory relies on stringent conditions on the input automaton. Although some relaxation on these restrictions has been done in the last decade, in general not every automaton can be synthesized while preserving its behavior. A crucial step for a non-synthesizable automaton is to transform it in order to satisfy the synthesis conditions. This paper revisits label splitting, a technique to satisfy the synthesis conditions through renaming of problematic labels. For the first time, the problem is formally characterized and its optimality addressed.

1 Introduction

The *synthesis problem* [4] consists in building a Petri net [8] that has a behavior equivalent to a given automaton (*transition system*). The problem was first addressed by Ehrenfeucht and Rozenberg [5] introducing *regions* to model the sets of states that characterize marked places in the Petri net. The theory is applicable to *elementary transition systems*, a proper subclass where additional conditions are required, and for which the synthesis produces a Petri net with isomorphic behavior. These restrictions were significantly relaxed in [3], introducing the subclass of *excitation-closed transition systems*, where not isomorphism but *bisimilarity* is guaranteed. The theory of this paper is within the subclass of excitation-closed transition systems.

When synthesis conditions do not hold, the Petri net derived might represent a proper superset of the initial behavior [1], and therefore any faithful use of such Petri net is rather limited. To overcome this problem, one might force the synthesis conditions by transforming the initial transition system. The work in [3] was the first in addressing this problem, introducing label splitting as a technique that can be applied when excitation-closure is not satisfied. The technique is based on relabeling the transitions of a particular event in the transition system with new copies of the same event, thus preserving the event name but considering each new copy as a new event with respect to the synthesis conditions. However, [3] only presented the technique as a heuristic to progress into excitation-closure.

The new copies produced by the label splitting technique increase the complexity of the Petri net derived: each new copy will be transformed into a transition, and hence the *label splitting problem* is to find an optimal sequence of splittings that induces the minimal number of transitions in the derived Petri net.

The label splitting technique presented in this paper is a particular one: it is defined on the sets of states computed when searching for regions in state-based synthesis methods [1, 3]. These sets, called *essential*, are the building blocks used in this paper to decide the labels to split. The methods for label splitting in the aforementioned work also use the essential sets for label splitting, but as described previously, only in a heuristic manner.

In summary, this paper presents a novel view on the label splitting technique. First, we show how label splitting for excitation closure is nothing else than computing the optimal coloring of a graph, i.e., the *chromatic number*. Second, we characterize the conditions under which an optimal label splitting can be derived to accomplish excitation closure. Finally, we present an algorithm that can be used when excitation closure can not been attained by a single application of the label splitting technique presented in this paper. This algorithm is based on a relaxation of the label splitting problem that can be mapped into the *weighted set cover* problem.

For the sake of clarity, the theory of this paper will be presented for the class of safe (1-bounded) Petri nets. The contribution can be extended with no substantial change for the class of general (k-bounded) Petri nets.

2 Preliminaries

2.1 Finite transition systems and Petri nets

Definition 1 (Transition system). *A* transition system *is a tuple* (S, E, A, s_{in})*, where* S *is a set of* states*,* E *is an alphabet of* actions*, such that* $S \cap E = \emptyset$*,* $A \subseteq S \times E \times S$ *is a set of* (labeled) transitions*, and* s_{in} *is the* initial state.

We use $s \xrightarrow{e} s'$ as a shortcut for $(s, e, s') \in A$, and we denote its transitive closure as $\xrightarrow{*}$. A state s' is said to be *reachable from state* s if $s \xrightarrow{*} s'$. Let $\mathsf{TS} = (S, E, A, s_{in})$ be a transition system. We consider connected transition systems that satisfy the following axioms: i) S and E are finite sets, ii) every event has an occurrence: $\forall e \in E : \exists (s, e, s') \in A$, and iii) every state is reachable from the initial state: $\forall s \in S : s_{in} \xrightarrow{*} s$.

Definition 2 (Petri net [8]). *A Petri net is a tuple* $\mathsf{PN} = (P, T, F, M_0)$ *where* P *and* T *represent finite disjoint sets of places and transitions, respectively, and* $F \subseteq (P \times T) \cup (T \times P)$ *is the flow relation. The initial marking* $M_0 \subseteq P$ *defines the initial state of the system.*

For a node n (place or transition) of a Petri net, $\bullet n$ ($n\bullet$) is the predecessor (successor) set of n in F. A transition $t \in T$ is *enabled* in a marking M iff $\bullet t \subseteq M$. The *firing* of t results in a new marking M', with one less token in $\bullet t$ and one more token in $t\bullet$. A marking M' is *reachable* from M if there is a sequence of firings

$\sigma = t_1 t_2 \dots t_n$ that transforms M into M', denoted by $M[\sigma\rangle M'$. A sequence of transitions $\sigma = t_1 t_2 \dots t_n$ is a *feasible sequence* if $M_0[\sigma\rangle M$, for some M. The set of all markings reachable from the initial marking M_0 is called its Reachability Set. The *Reachability Graph* of a Petri net PN ($\mathsf{RG}(\mathsf{PN})$) is a transition system in which the set of states is the Reachability Set, the events are the transitions of the net and a transition (M_1, t, M_2) exists if and only if $M_1 \xrightarrow{t} M_2$.

2.2 Regions and synthesis

The theory of regions provides a way to go from transition systems to Petri nets. We now review this theory (the interested reader can refer to [1,3–5,7] for a complete overview). Let S' be a subset of the states of a transition system, $S' \subseteq S$. If $s \notin S'$ and $s' \in S'$, then we say that transition (s, a, s') *enters* S'. If $s \in S'$ and $s' \notin S'$, then transition (s, a, s') *exits* S'. Otherwise, transition (s, a, s') *does not cross* S'.

Definition 3. *Let* $\mathsf{TS} = (S, E, A, s_{in})$ *be a transition system. Let* $S' \subseteq S$ *be a subset of states and* $e \in E$ *be an event. The following conditions (in the form of predicates) are defined for* S' *and* e*:*

$$\begin{aligned}
\mathtt{in}(e, S') &\equiv \exists (s, e, s') \in A : s, s' \in S' \\
\mathtt{out}(e, S') &\equiv \exists (s, e, s') \in A : s, s' \notin S' \\
\mathtt{nocross}(e, S') &\equiv \exists (s_1, e, s_2) \in A : s_1 \in S' \Leftrightarrow s_2 \in S' \\
\mathtt{enter}(e, S') &\equiv \exists (s_1, e, s_2) \in A : s_1 \notin S' \wedge s_2 \in S' \\
\mathtt{exit}(e, S') &\equiv \exists (s_1, e, s_2) \in A : s_1 \in S' \wedge s_2 \notin S'
\end{aligned}$$

Note that $\mathtt{nocross}(e, S') = \mathtt{in}(e, S') \ \vee \ \mathtt{out}(e, S')$. We will abuse the notation and will use $\mathtt{nocross}(e, S', (s_1, e, s_2))$ to denote a transition (s_1, e, s_2) that makes the predicate $\mathtt{nocross}(e, S')$ to hold, and the same for the rest of predicates.

The notion of a *region* is central for the synthesis of Petri nets. Intuitively, each region is a set of states that corresponds to a place in the synthesized Petri net, so that every state in the region models the marking of the place.

Definition 4 (Region). *A set of states* $r \subseteq S$ *in transition system* $\mathsf{TS} = (S, E, A, s_{in})$ *is called a* region *if the following two conditions are satisfied for each event* $e \in E$*:*

– *(i)* $\mathtt{enter}(e, r) \ \Rightarrow \ \neg\mathtt{nocross}(e, r) \ \wedge \ \neg\mathtt{exit}(e, r)$
– *(ii)* $\mathtt{exit}(e, r) \ \Rightarrow \ \neg\mathtt{nocross}(e, r) \ \wedge \ \neg\mathtt{enter}(e, r)$

A region is a subset of states in which *all* transitions labeled with the same event e have exactly the same "entry/exit" relation. This relation will become the successor/predecessor relation in the Petri net. The event may always be either an *enter* event for the region (case (i) in the previous definition), or always be an *exit* event (case (ii)), or never "cross" the region's boundaries, i.e. each transition labeled with e is *internal* or *external* to the region, where

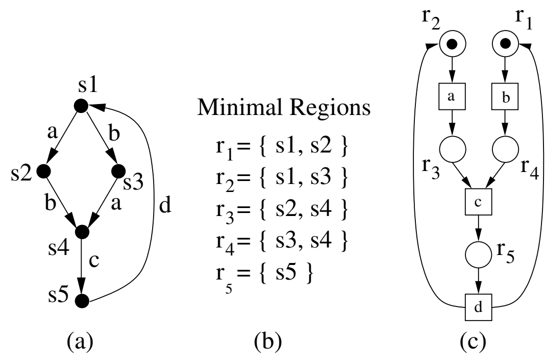


Fig. 1. (a) transition system, (b) minimal regions, (c) synthesis applying Algorithm of Figure 3.

the antecedents of neither (i) nor (ii) hold. The transition corresponding to the event will be successor, predecessor or unrelated with the corresponding place respectively. Examples of regions are reported in Figure 1: from the transition system of Figure 1(a), some regions are enumerated in Figure 1(b). For instance, for region r_2, event a is an exit event, event d is an entry event while the rest of events do not cross the region. Let r and r' be regions of a transition system. A region r' is said to be a *subregion* of r if $r' \subset r$. A region r is a *minimal* region if there is no other region r' which is a subregion of r. Going back to the example of Figure 1, in Figure 1(b) we report the set of minimal regions. Each transition system $\mathsf{TS} = (S, E, A, s_{in})$ has two *trivial regions*: the set of all states, S, and the empty set. The set of non-trivial regions of TS will be denoted by R_{TS}.

A region r is a *pre-region* of event e if there is a transition labeled with e which exits r. A region r is a *post-region* of event e if there is a transition labeled with e which enters r. The sets of all pre-regions and post-regions of e are denoted with ${}^\circ e$ and e°, respectively. By definition it follows that if $r \in {}^\circ e$, then all transitions labeled with e exit r. Similarly, if $r \in e^\circ$, then all transitions labeled with e enter r.

The computation of the minimal regions is crucial for the synthesis methods in [1, 3]. It is based on the notion of *excitation region* [6].

Definition 5 (Excitation region). *The excitation region of an event e, $\mathsf{ER}(e)$, is the set of states in which e is enabled, i.e.*

$$\mathsf{ER}(e) = \{s \mid \exists s' : (s, e, s') \in A\}$$

In Fig. 1(a), the set $\mathsf{ER}(c) = \{s4\}$ is an example of an excitation region[1]. The set of minimal regions sufficient for synthesis can be generated from the ERs of the events in a transition system in the following way: starting from the ER of

[1] Excitation regions are not regions in the terms of Definition 4. The term is used for historical reasons. For instance, $\mathsf{ER}(c)$ is not a region.

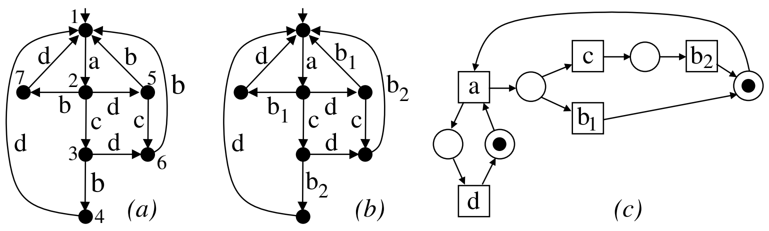


Fig. 2. (a) transition system (only numbers of states are shown), (b) ECTS by label-splitting, (c) synthesized Petri net.

each event, set expansion is performed on those events that violate the region condition (a pseudocode of the expansion algorithm is given in Figure 10 in [3]). The following lemma characterizes the states to be added in the expansion of ERs:

Lemma 1 (Essential states to become a region [3]). *Let* TS $= (S, E, A, s_{in})$ *be a transition system. Let* $r \subset S$ *be a set of states such that* r *is not a region. Let* $r' \subseteq S$ *be a region such that* $r \subset r'$. *Let* $e \in E$ *be an event that violates some of the conditions for* r *to be a region. The following predicates hold:*

1. $\texttt{in}(e, r) \wedge [\texttt{enter}(e, r) \vee \texttt{exit}(e, r)] \implies$
   $\{s|\exists s' \in r : (s, e, s') \in A \vee (s', e, s) \in A\} \subseteq r'$
2. $\texttt{enter}(e, r) \wedge \texttt{exit}(e, r) \implies$
   $\{s|\exists s' \in r : (s, e, s') \in A \vee (s', e, s) \in A\} \subseteq r'$
3. $\texttt{out}(e, r) \wedge \texttt{enter}(e, r) \implies$
   $[\{s|\exists s' \in r : (s, e, s') \in A\} \subseteq r'] \vee [\{s|\exists s' \notin r : (s', e, s) \in A\} \subseteq r']$
4. $\texttt{out}(e, r) \wedge \texttt{exit}(e, r) \implies$
   $[\{s|\exists s' \in r : (s', e, s) \in A\} \subseteq r'] \vee [\{s|\exists s' \notin r : (s, e, s') \in A\} \subseteq r']$

Hence, in cases 1 and 2 above, the violating event e is converted into a `nocross` event, where only one way of expanding r is possible. On the contrary, in case 3 (4) there are two possibilities for expansion, depending on whether the violating event will be converted into an `enter` or `nocross` (`exit` or `nocross`) event. In summary, set expansion to legalize violating events in a set of states generates a binary exploration tree. An example of such tree can be found in [3], Fig.12(b). The following definition formalizes the notion of essential set:

Definition 6 (Essential set of an event). *Let* TS $= (S, E, A, s_{in})$ *be a transition system, and event* $e \in E$. Essential(e) *is the set of sets of states found by inductive application of Lem. 1, with initial set* ER*(e).*

Notice that $Essential(e) \subseteq \mathcal{P}(S)$, and $\forall S_i \in Essential(e)$: $\mathsf{ER}(e) \subseteq S_i$. For instance, in Fig. 2(a) $Essential(c) = \{\mathsf{ER}(c) = \{s2, s5\}, \{s1, s2, s5, s7\}, \{s2, s3, s5, s6\}\}$. The two states different from $\mathsf{ER}(c)$ in $Essential(c)$ are computed according to case 4 in Lem. 1 on event b.

Algorithm: Petri net synthesis

- For each event $e \in E$ generate a transition labeled with e in the Petri net;
- For each minimal region $r_i \in R_{\mathsf{TS}}$ generate a place r_i;
- Place r_i contains a token in the initial marking iff the corresponding region r_i contains the initial state of TS s_{in};
- The flow relation is as follows: $e \in r_i\bullet$ iff r_i is a pre-region of e and $e \in \bullet r_i$ iff r_i is a post-region of e, i.e.,

$$F_{\mathsf{TS}} \overset{def}{=} \{(r,e) | r \in R_{\mathsf{TS}} \ \wedge \ e \in E \ \wedge \ r \in {}^{\circ}e\} \\ \cup \{(e,r) | r \in R_{\mathsf{TS}} \ \wedge \ e \in E \ \wedge \ r \in e^{\circ}\}$$

Fig. 3. Algorithm for Petri net synthesis from [7].

The procedure given by [7] to synthesize a Petri net, $N_{\mathsf{TS}} = (R_{\mathsf{TS}}, E, F_{\mathsf{TS}}, R_{s_{in}})$, from an *elementary transition system*[2], $\mathsf{TS} = (S, E, A, s_{in})$, is illustrated in Figure 3. Notice that only minimal regions are required in the algorithm [4]. An example of the application of the algorithm is shown in Figure 1. The initial transition system and the set of minimal regions is reported in Figures 1(a) and (b), respectively. The synthesized Petri net is shown in Figure 1(c).

2.3 Excitation-closed transition systems

Definition 7 (Excitation-closed transition systems). *A transition system* $\mathsf{TS} = (S, E, A, s_{in})$ *is* excitation-closed *(*ECTS*) if it satisfies the following two axioms:*

- *Excitation closure (*EC*): For each event a:* $\bigcap_{r \in {}^{\circ}a} r = \mathsf{ER}(a)$
- *Event effectiveness: For each event a:* ${}^{\circ}a \neq \emptyset$

The synthesis algorithm in Figure 3 applied to an ECTS derives a Petri net with reachability graph *bisimilar* to the initial transition system [3]. When the transition system is not excitation closed, then it must be transformed to enforce that property. One possible strategy is to represent every event by multiple transitions with the same label. This technique is called *label splitting.* Figure 2 illustrates the technique. The initial transition system, shown in Figure 2(a), is not an ECTS: the event c is not EC (the only minimal region that contains $\mathsf{ER}(c)$ is $\{s2, s3, s5, s6\}$, which does not makes the EC axiom of Def. 7 to hold). The transition system is transformed by splitting the event b into the events b_1 and b_2, as shown in Figure 2(b), resulting in an ECTS. The synthesized Petri net, with two transitions for event b is shown in Figure 2(c). The reachability

[2] Elementary transition systems are a proper subclass of the transition systems considered in this paper, were additional conditions to the ones presented in Section 2.1 are required.

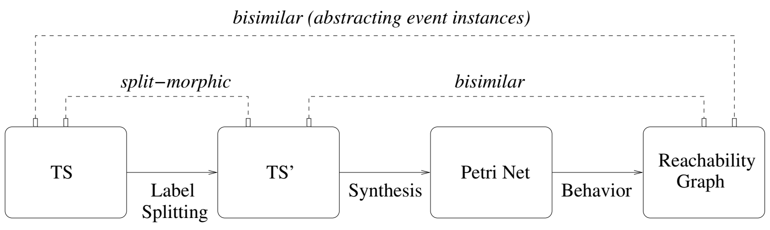


Fig. 4. Relationship between the different objects if label splitting is applied.

graph of the Petri net of Fig. 2(c) is *split-morphic* [3] to the transition system of Fig. 2(a): there exist a surjective mapping between the sets of events, where different instances of the same event (a_1, a_2, ...) are mapped to the only one event a. If we abstract away the label indexes, the equivalence relation defined is bisimilarity. Fig. 4 shows the relationships between the original transition system, the transformed one obtained through label splitting, and the reachability graph of the synthesized Petri net.

Hence in Petri net synthesis label splitting might be crucial for the existence of a Petri net with bisimilar behavior. The following definition describes the general application of label splitting:

Definition 8 (Label splitting). *Let* $\mathsf{TS} = (S, E, A, s_{in})$ *be a transition system. The splitting of event* $e \in E$ *derives a transition system* $\mathsf{TS}' = (S, E', A', s_{in})$*, with* $E' = E - \{e\} \cup \{e_1, \ldots, e_n\}$*, and such that every transition* $(s_1, e, s_2) \in A$ *corresponds to exactly one transition* (s_1, e_i, s_2)*, and the rest of transitions for events different from* e *in* A *are preserved in* A'.

Label splitting is a powerful transformation which always guarantees excitation closure: any TS can be converted into one where every transition has a different label. By definition, the obtained TS is ECTS but the size of the derived Petri net is equal to the size of the obtained ECTS. In this paper we aim at reducing the number of splittings, thus reducing the size of the Petri net derived.

The work presented in this paper considers a particular application of the label splitting technique which is based on converting a set into a region, described in the next section. By restricting the transformation, we are able to determine these situations where a minimal set of labels is enough to guarantee excitation closure.

3 Optimal label splitting to attain a region

In this section the following problem is addressed: given a transition system $\mathsf{TS} = (S, E, A, s_{in})$ and a set of states $S' \subseteq S$ which is not a region, determine the minimal number of label splittings to be applied in order to S' become a region. This is a crucial step for the technique presented in the following section to satisfy the ECTS condition. The main contribution of this section is

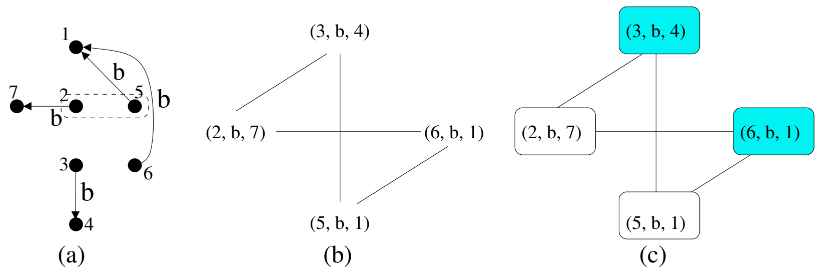


Fig. 5. (a) Projection of the transition system of Fig. 2(a) for the transitions on event b and set of states $\{s_2, s_5\}$, (b) $\mathcal{GG}(b, \{s_2, s_5\})$, (c) coloring.

to show that the problem might be reduced to compute the *chromatic number* of a graph [9].

First we introduce the concept of *gradient graph*:

Definition 9 (Gradient Graph). *Given a transition system* $\mathsf{TS} = (S, E, A, s_{in})$*, a set* $S' \subseteq S$ *and an event* $e \in E$*, the* gradient graph *of* e *with respect to* S' *in* TS*, denoted as* $\mathcal{GG}(e, S') = (A_e, M)$ *is an undirected graph defined as:*

– $A_e = \{(s, x, s') | (s, x, s') \in A \ \wedge \ x = e\}$, *is the set of nodes*
– $M = \{(v, v') | v, v' \in A_e \ \wedge$
$[(\mathtt{enter}(e, S', v) \ \wedge \ (\mathtt{nocross}(e, S', v') \ \vee \ \mathtt{exit}(e, S', v'))) \ \vee$
$(\mathtt{exit}(e, S', v) \wedge (\mathtt{nocross}(e, S', v') \vee \mathtt{enter}(e, S', v')))]\}$, *is the set of edges.*

Informally, the gradient graph contains as nodes the transitions of an event e, and an edge exist between two nodes if they satisfy different predicates on set S'. For instance, the gradient graph on event b and set of states $S' = \{s_2, s_5\}$ in the transition system of Fig. 2(a) is shown in Fig. 5(b) (for the sake of clarity we show in Fig. 5(a) the transitions on event b from Fig. 2(a)).

A graph $G = (V, E)$ is *k-colourable* if there exists an assignment $\alpha : V \to \{1, 2, \ldots k\}$ for which any pair of nodes $v, v' \in V$ such that $(v, v') \in E$ satisfy $\alpha(v) \neq \alpha(v')$. The *chromatic number*, $\chi(G)$, of a graph G is the minimum k for which G is k-colourable [9]. The rest of the section shows the relation between the chromatic number and the optimal label splitting to attain a region.

Definition 10 (Label splitting as gradient graph coloring). *Given a transition system* $\mathsf{TS} = (S, E, A, s_{in})$*, the label splitting of event* e *according to a coloring* α *of the gradient graph* $\mathcal{GG}(e, S')$ *derives the transition system* $\mathsf{TS}' = (S, E', A', s_{in})$ *where* $E' = E - \{e\} \cup \{e_1, \ldots, e_n\}$*, with* $\{e_1, \ldots, e_n\}$ *being the colors defined by the coloring of* $\mathcal{GG}(e, S')$*. Every transition* (s, e, s') *of event* e *is transformed into* $(s, e_{\alpha((s,e,s'))}, s')$*, whilst the rest of transitions of events in* $E - \{e\}$ *are preserved in* A'*.*

For instance, the label splitting of the transition system of Fig. 2(a) according to the coloring shown in Fig. 5(c) of $\mathcal{GG}(b, \{s_2, s_5\})$ is shown on Fig. 2(b).

Proposition 1. *Given a transition system* $\mathsf{TS} = (S, E, A, s_{in})$ *and the gradient graph* $\mathcal{GG}(e, S')$*. If event* e *is split in accordance with a* $\chi(\mathcal{GG}(e, S'))$*-coloring of* $\mathcal{GG}(e, S')$ *then the new events* $\{e_1, \ldots, e_{\chi(\mathcal{GG}(e,S'))}\}$ *inserted satisfy the region condition on* S' *(c.f., Def. 4)).*

Proof. By contradiction: assume there exists $e_i \in \{e_1, \ldots, e_{\chi(\mathcal{GG}(e,S'))}\}$ such that conditions of Def. 4 do not hold. Without loss of generality, we assume that there exist (s_1, e_i, s_2) and (s'_1, e_i, s'_2) for which predicates $\texttt{enter}(e_i, S', (s_1, e_i, s_2))$ and $\texttt{nocross}(e_i, S', (s_1, e_i, s_2))$ hold (the other cases can be proven similarly). But then the nodes (s_1, e_i, s_2) and (s'_1, e_i, s'_2) are connected by an edge, but they are assigned the same color e_i. Contradiction. $\square$

Corollary 1. *Given a transition system* $\mathsf{TS} = (S, E, A, s_{in})$ *and a set* $S' \subseteq S$*. If every event* e *is split according to the colors required for achieving* $\chi(\mathcal{GG}(e, S'))$*, then* S' *is a region in the resulting transition system.*

Proof. It follows from iterative application of Prop. 1. $\square$

In the following theorem we abuse the notation and extend the χ operator to sets of states:

Theorem 1. *Given a transition system* $\mathsf{TS} = (S, E, A, s_{in})$ *and a set* $S' \subseteq S$*, let* $\chi(S') = \chi(\mathcal{GG}(e_1, S')) + \ldots + \chi(\mathcal{GG}(e_n, S'))$*, with* $E = \{e_1, \ldots, e_n\}$*. Then* $\chi(S')$ *is a lower bound to the number of labels needed to make* S' *to be a region.*

Proof. By contradiction: if there is a $k < \chi(S')$ such that only k labels are needed to convert S' into region, then there is an event $e \in E$ for which less than $\chi(\mathcal{GG}(e, S'))$ labels are used to split e for satisfying the conditions of Def. 4. This leads to a contradiction to the chromatic number of the graph $\mathcal{GG}(e, S')$. $\square$

The theory of this section represents the core idea for the label splitting technique of this paper. The next section shows how apply it to attain ECTSs.

4 Optimal label splitting on essential sets for synthesis

Given a non-ECTS, the following question arises: is there an algorithm to transform it into an ECTS with a minimal number of label splittings? This section addresses this problem, deriving sufficient conditions under which a positive answer can be given. As was done in previous work [3], in this paper we will restrict the theory to a particular application of the label splitting: instead of an arbitrary instantiation of Def. 8 which may split an event of a transition system in peculiar way, we will only consider the splittings used to convert essential sets into regions (Def. 10), technique which has been shown in the previous section.

We will tackle the problem in two phases: first, we will show how the EC for a given event can be achieved by using the essential sets found in the expansion of the ER of an event, defined in Lem. 1. Then we will show the conditions under which the strategy can be applied in the general case, i.e. considering the whole set of non-EC events. We start by defining the set of states that both are included in the intersection of pre-regions of an event but are not included in the excitation region:

$$Remainder(e) = (\bigcap_{q \in {}^\circ e} q) \setminus \mathsf{ER}(e)$$

Clearly, event e is EC if and only if $Remainder(e) = \emptyset$ (cf., Def. 7).

For the definitions and theorems below, we assume a context transition system $\mathsf{TS} = (S, E, A, s_{in})$, and use $Witness(e)$ to denote the sets of essential sets of an event e such that, if they are converted into regions, $Remainder(e)$ becomes empty. Formally:

Definition 11 (Witness set of an event). *Let* $\mathsf{TS} = (S, E, A, s_{in})$ *be a transition system, and event* $e \in E$. Witness(e) *is defined as follows:*

$$\mathcal{C} = \{S_1, \ldots, S_k\} \in Witness(e) \iff \forall S_i \in \mathcal{C} : S_i \in Essential(e) \wedge \\ (\textstyle\bigcap_{q \in ({}^\circ e \,\cup\, \mathcal{C})} q) \setminus \mathsf{ER}(e) = \emptyset$$

Finally, if $\mathcal{C} = \{S_1, \ldots, S_k\}$, we abuse the notation and use $\chi(\mathcal{C})$ to denote $\chi(\mathcal{GG}(e_1, S_1) \cup \ldots \cup \mathcal{GG}(e_1, S_k)) + \ldots + \chi(\mathcal{GG}(e_n, S_1) \cup \ldots \cup \mathcal{GG}(e_n, S_k))$, with $E = \{e_1, \ldots, e_n\}$. The union operator on gradient graphs is defined as $\mathcal{GG}(e, S_1) \cup \ldots \cup \mathcal{GG}(e, S_k) = (A_e, M_1 \cup \ldots \cup M_k)$, with A_e and M_i being the nodes and edges of the graph $\mathcal{GG}(e, S_i)$, respectively c.f., Def. 9[3].

First, we start by describing the minimal strategy to make an event to satisfy its excitation closure:

Proposition 2. *Let* $\mathcal{C} = \{S_1, \ldots, S_k\} \in Witness(e)$ *such that* $\chi(\mathcal{C})$ *is minimal, i.e.,* $\forall \mathcal{C}' \in Witness(e) : \chi(\mathcal{C}) \leq \chi(\mathcal{C}')$. *Then, if only label splitting on essential sets (Def. 10) is considered,* $\chi(\mathcal{C})$ *is the minimal number of labels needed to make* $Remainder(e) = \emptyset$.

Proof. The minimality of $\chi(\mathcal{C})$ guarantees that, by using label splitting on essential sets, the minimal number of labels has been used in the coloring of the gradient graphs of each event for each set in $\mathcal{C}$. □

Given a non-ECTS, the optimal label splitting problem on essential sets is to determine the sets to convert into regions in order to reduce the set *Remainder* to the empty set for each non-EC event, using minimal number of labels.

Definition 12 (Optimal label splitting on essential sets). *Given the non-*EC *events* $e_1 \ldots e_k \in E$, *define the universe* $\mathcal{U}$ *as* $Witness(e_1) \cup \ldots \cup Witness(e_k)$. *The optimal label splitting problem is to determine sets* $S_1, \ldots, S_n \subseteq \mathcal{U}$ *such that* $\forall 1 \leq i \leq k : \exists \mathcal{C} \in Witness(e_i) : \mathcal{C} \subseteq \{S_1, \ldots, S_n\}$ *and where* $\chi(S_1, \ldots, S_n)$ *is minimal.*

[3] We only consider the union of gradient graphs of the same event.

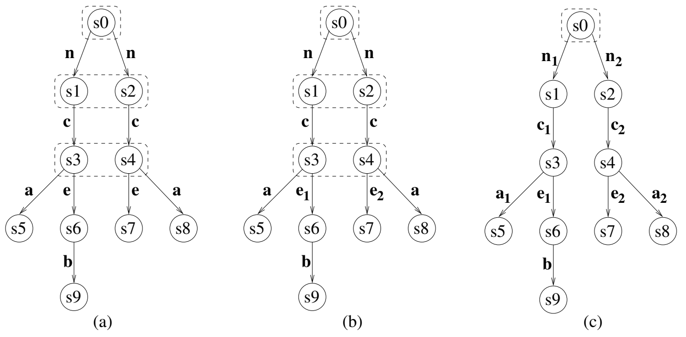


Fig. 6. (a) Initial transition system: all the events but b are EC, (b) the splitting of e leads to new events e_1 and e_2 not satisfying the EC property, (c) the final ECTS, where all events have been split.

Notice that Def. 12 is defined on the set of all essential sets of the non-EC events, searching for a set of essential sets which both has minimal number of labels and ensures EC for these events. An interesting result guarantees EC preservation for those events that both satisfy initially the EC and were not split:

Proposition 3. *Let $e \in E$ be such that $Remainder(e) = \emptyset$ and it has not been selected for splitting. Then $Remainder(e) = \emptyset$ in the new transition system obtained after label splitting.*

Proof. Label splitting preserves the set of regions: the predicates of Def. 4 that hold on each region will also hold if some event is split. Hence, the witness set of regions that ensures $Remainder(e) = \emptyset$ is still valid after label splitting. $\square$

However, label splitting may break events for which the excitation closure was satisfied or attained. Unfortunately, the new events appearing might not satisfy the excitation closure property as the following example demonstrates.

Example 1. In the transition system of Figure 6(a) events n, c, a and e are EC, as demonstrated by their witness: $Witness(n) = \{\{s0\}\}$, $Witness(c) = \{\{s1, s2\}\}$, $Witness(a) = Witness(e) = \{\{s3, s4\}\}$. However, b is not EC, and the splitting required for b to be EC is done on the essential set $\{s6\}$, which requires to split the EC event e, resulting in the transition system of Fig. 6(b). The new events e_1 and e_2 arising from the splitting of e are not EC. This requires further splittings, which as in the case of b, force the splitting of EC events resulting in new events that are not EC. Four iterations are required to attain the ECTS shown in Fig. 6(c).

This example invalidates any label splitting strategy that aims at reaching excitation closure in just one iteration of the synthesis process: in general, when the splitting of some event is applied, its ER is divided into several ERs for which there might be no witness which guarantee the EC of these new events arising.

Importantly, the label splitting technique preserves the regions, but new regions might be necessary for the new events arising from a splitting. Therefore, any label splitting technique must be an iterative method (see next section for such a method). However, if the new labels inserted do not incur excitation closure problems, the presented technique guarantees the optimal label splitting:

Theorem 2. *Let* $\mathsf{TS}' = (S, E', A', s_{in})$ *be the transition system reached after splitting labels on non-*EC *events* $e_1, \dots e_k$ *in transition system* $\mathsf{TS} = (S, E, A, s_{in})$*, using the strategy from Def. 12. Then, if the new events appearing are* EC*, the number of splittings performed is minimal and* TS' *is* ECTS*.*

Proof. The minimality of the strategy used in Def. 12 ensures that no less splittings are possible to make $e_1, \dots e_k$ EC. Moreover, the assumption implies that the new events arising from the splitting are EC. Finally, Prop. 3 guarantees that EC events that were not split are still EC. The set of events E' is partitioned into these three sets, and therefore TS' is ECTS. □

5 A greedy algorithm for iterative label splitting

The optimization problem for optimal label splitting presented in the previous section is hard to tackle: considering the global optimal label splitting through local achievement of EC for every non-EC event. However, if some relaxations are done, the problem can be seen as an instance of the *weighted set cover* (WSC) problem [2]. This section shows how to cast the problem into an WSC setting and depicts an algorithm for label splitting that iterates until excitation closure is attained. Unfortunately, for a set of witness elements $S_1, \dots, S_n$, the equality

$$\chi(S_1, \dots, S_n) = \chi(S_1) + \dots + \chi(S_n) \tag{1}$$

does not hold in general, e.g., imagine that some S_i, S_j satisfy $S_i \cap S_j \neq \emptyset$: in that situation one may be counting twice the labels (colors) needed in the right part of the equality of (1). That prevents an exact mapping of the label splitting optimization problem into the WSC setting. However, one may allow this imprecision for the sake of having an efficient manner to derive good candidates. The following model maps the optimization problem of Def. 12 into an ILP problem, whose solution provides candidates that, although potentially suboptimal, might represent good candidates for label splitting in practice:

$$\begin{array}{rl} & min \sum_{\mathcal{C} \in \mathcal{W}} \chi(\mathcal{C}) \cdot X_{\mathcal{C}} \\ s.t. & \\ & \forall e \in e_1 \dots e_k : \sum_{\mathcal{C} \in Witness(e)} X_{\mathcal{C}} \geq 1 \\ & X_{\mathcal{C}} \in \{0, 1\} \end{array} \tag{2}$$

where $e_1 \dots e_k \in E$ are the set of non-EC events and $\mathcal{W} = Witness(e_1) \cup \dots \cup Witness(e_k)$. A solution to the ILP model (2) will then minimize the sum of the cost of the witness selected for each non-EC event.

Algorithm 1: IterativeSplittingAlgorithm

```
   Input: Transition system $\mathsf{TS} = (S, E, A, s_{in})$
   Output: Excitation-closed transition system $\mathsf{TS}' = (S, E', A', s_{in})$ bisimilar to
           $\mathsf{TS}$
1  begin
2      $\mathsf{TS}' = \mathsf{TS}$
3      $\mathcal{R}$ = GenerateMinimalRegions($\mathsf{TS}$)
4      while not (ExcitationClosed($\mathsf{TS}'$,$\mathcal{R}$)) do
5          $\mathcal{W}$ = CollectWitnesses($\mathsf{TS}'$)
6          $(S_1, \dots, S_n)$ = Solution ILP model (2)
7          $\mathsf{TS}'$ = SplitLabels($\mathsf{TS}'$, $S_1, \dots, S_n$)
8          $\mathcal{R} = \mathcal{R} \cup \{S_1, \dots, S_n\}$
9      end
10 end
```

Algorithm 1 presents the iterative strategy to derive an ECTS. It first computes the minimal regions of the initial transition system. Then the main loop of the technique starts by collecting the witnesses for non-EC events of the current transition system, which are provided in the set $\mathcal{W}$ (line 5). Then the cover for the non-EC events is computed by solving the ILP model (2) in line 6. Notice that for the sake of clarity of the algorithm, we provide in line 6 the sets that form the cover instead of providing the particular witness selected for each event (i.e., given a solution $\mathcal{C}_1, \dots, \mathcal{C}_k$ of model (2), $\{S_1, \dots, S_n\} = \bigcup_{1 \le i \le k} \mathcal{C}_i$). In line 7 the splitting of labels corresponding to $\chi(S_1, \dots, S_n)$ is performed, ensuring that sets $S_1, \dots, S_n$ become regions. The new regions are appended to the regions found so far (which are still regions, see proof of Prop. 3), and the excitation closure is re-evaluated to check convergence.

Although the presented iterative technique is not guaranteed to provide the optimal, the improvements with respect to the previous (also non-optimal) approaches [1, 3] are:

– the whole set of non-EC events are considered in every iteration: in previous work only one is considered, and
– for every non-EC event, the necessary splittings are applied to attain excitation closure: on the previous work, only one of the splittings was applied.

Hence the *macro* technique presented in this section is meant to speed-up the achievement of the excitation closure, when compared to the *micro* techniques presented in the literature.

6 Discussion and conclusions

This paper has presented a fresh look at the problem of label splitting, by relating it to some of the well-known NP-complete problems like chromatic number or set covering. In a restricted application of label splitting based on essential sets, optimality is guaranteed if certain conditions hold.

For the sake of clarity, we have restricted the theory to safe Petri nets. The extension to k-bounded Petri nets can be done by adapting the notion of gradient graph to use *gradients* (see [1] for the formal definition of gradient) instead of the predicates required in Def. 4. Given a *multiset* r which is not a k-bounded region, the essential sets will be those multisets $r' \geq r$ such that the number of gradients for some event has decreased, i.e., r' is a sound step towards deriving a region from multiset r. Finally, the excitation closure definition for the general case should be the one described in [1], and the witness of an event defined accordingly.

As a future work, addressing the general problem with unrestricted application of splitting (i.e., not using essential sets but instead arbitrary selections of labels to split) might be an interesting direction to follow. Also, incorporating the presented techniques into our synthesis tool [1] will be considered.

References

1. J. Carmona, J. Cortadella, and M. Kishinevsky. New region-based algorithms for deriving bounded Petri nets. *IEEE Transactions on Computers*, 59(3):371–384, 2009.
2. T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001.
3. J. Cortadella, M. Kishinevsky, L. Lavagno, and A. Yakovlev. Deriving Petri nets from finite transition systems. *IEEE Transactions on Computers*, 47(8):859–882, Aug. 1998.
4. J. Desel and W. Reisig. The synthesis problem of Petri nets. *Acta Inf.*, 33(4):297–315, 1996.
5. A. Ehrenfeucht and G. Rozenberg. Partial (Set) 2-Structures. Part I, II. *Acta Informatica*, 27:315–368, 1990.
6. M. Kishinevsky, A. Kondratyev, A. Taubin, and V. Varshavsky. *Concurrent Hardware: The Theory and Practice of Self-Timed Design*. John Wiley and Sons, London, 1993.
7. M. Nielsen, G. Rozenberg, and P. Thiagarajan. Elementary transition systems. *Theoretical Computer Science*, 96:3–33, 1992.
8. C. A. Petri. *Kommunikation mit Automaten*. PhD thesis, Bonn, Institut für Instrumentelle Mathematik, 1962. (technical report Schriften des IIM Nr. 3).
9. D. B. West. *Introduction to Graph Theory*. Prentice-Hall, 1996.

Aggregating Causal Runs into Workflow Nets

B.F. van Dongen, J. Desel, and W.M.P. van der Aalst

b.f.v.dongen@tm.tue.nl, joerg.desel@fernuni-hagen.de, w.m.p.v.d.aalst@tm.tue.nl

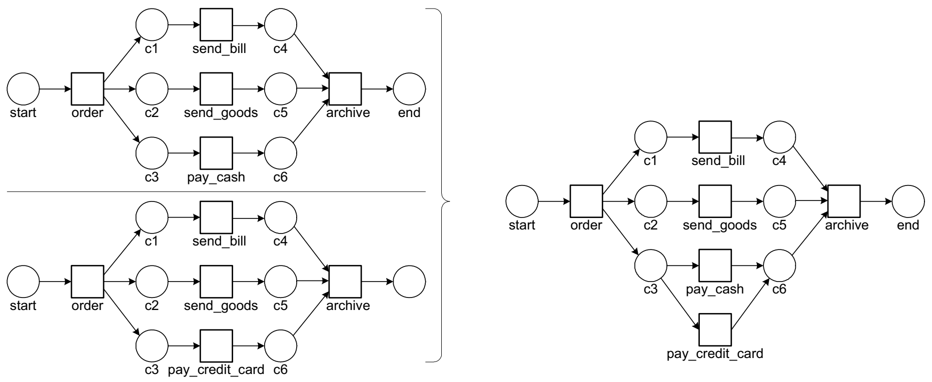
Abstract. This paper provides three algorithms for constructing system nets from sets of partially-ordered causal runs. The three aggregation algorithms differ with respect to the assumptions about the information contained in the causal runs. Specifically, we look at the situations where labels of conditions (i.e. references to places) or events (i.e. references to transitions) are unknown. Since the paper focusses on aggregation in the context of process mining, we solely look at workflow nets, i.e. the class of Petri nets with unique start and end places. The difference of the work presented here and most work on process mining is the assumption that events are logged as partial orders instead of linear traces. Although the work is inspired by applications in the process mining and workflow domains, the results are generic and can be applied in other application domains.

1 Introduction

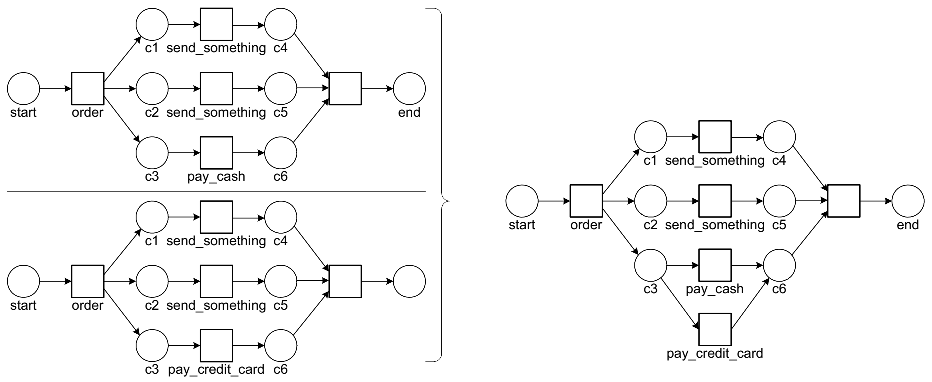
This paper proposes different approaches to "discover" process models from observing runs, i.e., runs (also known as causal nets or occurrence nets, see e.g. [1]) are aggregated into a single Petri net that captures the observed behaviour. Runs provide information about events together with pre- and post-conditions which constitute a (partial) order between these events.

There are many techniques to discover sequential process models based on event logs (also known as transaction logs, audit trails, etc). People working on process mining techniques [2] also tackle situations where processes may be concurrent and the set of observations is incomplete. The set of possible sequences is typically larger than the number of process instances thus making it unrealistic to assume that all possible combinations of behaviour have been observed.

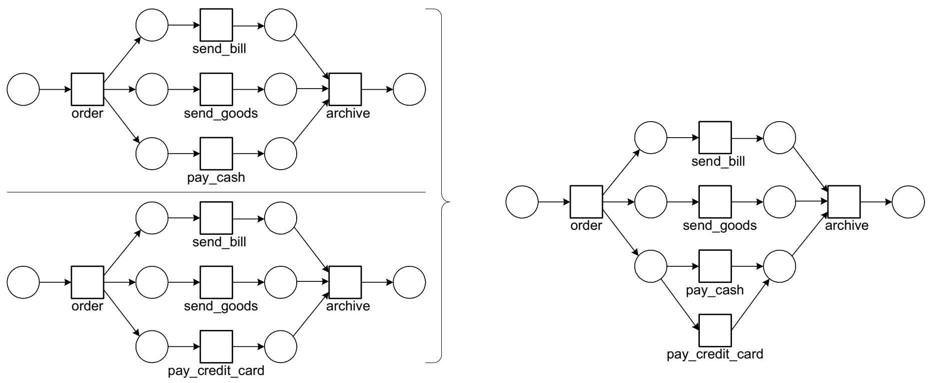
In many applications, the event log is linear, e.g., sorted based on timestamps, and an approach based on runs is not applicable. However, there are processes where it is possible to monitor causal dependencies (e.g., by analyzing the dataflows). We also encountered some systems that actually log behaviour using a representation similar to runs. The ad-hoc workflow management system InConcert of Tibco (formerly Xerox) allows end users to define and modify process instances (e.g., customer orders) while capturing the causal dependencies between the various activities. The representation used directly corresponds to runs. The analysis tool ARIS PPM (Process Performance Monitor) of IDS Scheer can extract runs represented as so-called *instance EPCs* (Event-driven Process Chains) from systems such as SAP R/3 and Staffware. These examples show that in real-life systems and processes runs can be recorded or already are being recorded, thus motivating the work presented in this contribution.



(a) Known event and condition labels.



(b) Known condition labels and unknown or non-unique event labels.



(c) Known event labels and unknown condition labels.

Fig. 1. Example describing the three problems tackled in this paper.

To introduce the main topic of this paper we use the examples shown in Figure 1. The left-hand side of this figure shows several runs. These are the behaviours that have been observed by extracting information from e.g. some enterprise information system. The right-hand side shows the models that we would like to discover by aggregating the runs shown on the left-hand side. Runs are represented by acyclic Petri nets without any choices. Figure 1 (a) shows the easiest situation. Here we assume that in the run all event and condition labels have been recorded in some event log. There are two runs and it is easy to see that the aggregated model can indeed reproduce these two runs, and no other runs are possible. However, in general not all possible runs need to be present. In most application domains the number of possible runs is larger than the actual number of process instances. Figure 1 (b) describes a more complex problem where not all event labels are recorded or where the same label may refer to different transitions. For example, in Figure 1 (b) the archive event is no longer visible and the two send events (`send_goods` and `send_bill`) cannot be distinguished, since both of them are recorded as `send_something`. Figure 1 (c) illustrates the most challenging problem, i.e., the event labels are given but the condition labels are not recorded at all. Nevertheless, the Petri net on the right is the most likely candidate process to exhibit such behaviour. In this paper we will tackle the problem of aggregating runs into a Petri net that can generate these runs. We will show that it is possible to do this for the three situations depicted in Figure 1.

The generation of system nets from their causal runs has been investigated before. The first publication on this topic is [3]. Here the basis is assumed to be the set of all runs. These runs are folded, i.e., events representing the occurrence of the same transition are identified, and so are conditions representing a token on the same place. In [4] a similar folding approach is taken, but there the authors start with a set of causal runs, as we do in the present paper. [4] does not present algorithms in details for the aggregation of runs but rather concentrates on correctness criteria for the derived system net. [5] extracts Petri nets from models which are based on Message Sequence Charts (MSCs), a concept quite similar to causal runs. Less related is the work presented in [6], where a special variant of MSCs is used to generate a system implementation.

In more recent papers [7], so-called regions are defined for partial orders of events representing runs. These regions correspond to anonymous places of a synthesized Place/Transition net, which can generate these partial orders. In contrast to our work, the considered partial orders are any linearizations of causal orders, i.e., two ordered events can either occur in a sequence (then there is a causal run with a condition ”between” the events) or they can occur concurrently. Consequently, conditions representing tokens on places are not considered in these partial orders whereas our approach heavily depends on these conditions.

The goal of process mining is to extract information about processes from event logs. One of its aspects focusses on finding a process specification, based on a set of executions of that process, i.e. a process log is taken as a starting point. A variety of algorithms have been proposed to generate a process model based on this log. Typically, such a log is considered to consist of cases (i.e. process instances, for example one insurance claim in a process dealing with insurance claims) and all tasks in each case are totally ordered (typically based on the timestamps). In this paper, we take a different approach. We start by looking at so-called runs. These runs are a *partial* ordering on the tasks within each case. However, in addition to the partial ordering of tasks, we may have information about the local states of the system from which the logs originated, i.e. for each event the pre- and post-conditions are known. This closely relates to the process mining algorithms presented in [8] and [9]. However, also in these papers only causal dependencies between events and no state information is assumed to be known.

In this paper, we provide three algorithms for the aggregation of runs. First, we assume we indeed have full knowledge of each event, its preconditions and its postconditions. Then, we assume that we cannot uniquely identify events, i.e. the label of an event may refer to multiple transitions. Finally, we provide an algorithm that assumes less knowledge about pre- and post-conditions. Before we elaborate on our results, we first provide some preliminary definitions that we use throughout the paper.

2 Preliminaries

Let $G = (N, E)$ be a directed graph, i.e. N is the set of nodes and $E \subseteq N \times N$ is the set of edges. If $N' \subseteq N$, we say that $G' = (N', E \cap (N' \times N'))$ is a subgraph of G. $N' \subseteq N$ generates a *maximal connected subgraph* if it is a maximal set of vertices generating a connected subgraph. For $n \in N$, we define $\overset{G}{\bullet} n = \{m \in N \mid (m, n) \in E\}$ as the pre-set and $n \overset{G}{\bullet} = \{m \in N \mid (n, m) \in E\}$ as the post-set of n with respect to the graph G. If the context is clear, the superscript G is omitted.

Let $G = (N, E)$ be a graph. Let μ be a set of colors. A function $f : N \to \mu$ is a coloring function if, for all $(n_1, n_2) \in E$, either $n_1 = n_2$ or $f(n_1) \neq f(n_2)$.

As stated in the introduction, our starting point is not only a partial order of events within a case, but also information about the state of a case. Since we want to be able to represent both events and states, Petri nets provide a natural basis for our approach. In this paper, we use the standard definition of finite marked Place/Transition (P/T-nets) nets $N = (P, T, F, M_0)$. We restrict ourselves to P/T nets where for all transitions t holds that $\bullet t \neq \emptyset$ and $t \bullet \neq \emptyset$.

We use square brackets for the enumeration of the elements of a bag representing a marking of a P/T-net, e.g. $[2a, b, 3c]$ denotes the bag with two a's, one b, and three c's. The sum of two bags ($X \uplus Y$), the presence of an element in a bag ($a \in X$), and the notion of subbags ($X \leq Y$) are defined in a straightforward way, and they can handle a mixture of sets and bags. Furthermore, $\biguplus_{a \in A} \big(f(a)\big)$ denotes the sum over the bags that are results of function f applied to the elements a of a bag A.

Petri nets specify processes. The behaviour of a Petri net is given in terms of causal nets, representing process instances (i.e. cases). Therefore, we introduce some concepts (notation taken from [1]). First, we introduce the notion of a causal net, this is a specification of one process instance of some process specification.

Definition 2.1. (Causal net)
A P/T net (C, E, K, S_0) is called a causal net if:

- for every place $c \in C$ holds that $| \bullet c| \leq 1$ and $|c \bullet | \leq 1$,
- the transitive closure of K is irreflexive, i.e. it is a partial order on $C \cup E$,
- for each place $c \in C$ holds that $S_0(c) = 1$ if $\bullet c = \emptyset$ and $S_0(c) = 0$ if $\bullet c \neq \emptyset$.

In causal nets, we refer to places as *conditions* and to transitions as *events*.

Each event of a causal net should refer to a transition of a corresponding P/T-net and each condition should refer to a token on some place of the P/T-net. These references are made by mapping the conditions and the events of a causal net onto places and transitions, respectively, of a Petri net. We call the combination of a causal net and such a mapping a *run*.

Definition 2.2. (Run)
A run (N, α, β) of a P/T-net (P, T, F, M_0) is a causal net $N = (C, E, K, S_0)$, together with two mappings $\alpha : C \to P$ and $\beta : E \to T$, such that:

- For each event (transition) $e \in E$, the mapping α induces a bijection from $\bullet e$ to $\bullet \beta(e)$ and a bijection from $e\bullet$ to $\beta(e)\bullet$,
- $\alpha(S_0) = M_0$ where α is generalized to markings by $\alpha : (C \to \mathbb{N}) \to (P \to \mathbb{N})$, such that $\alpha(S_0)(p) = \sum_{c|\alpha(c)=p} S_0(c)$.

The causal behaviour of the P/T-net (P, T, F, M_0) is defined as its set of runs. To avoid confusion, the P/T-net (P, T, F, M_0) is called *system net* in the sequel.

3 Aggregation of Runs

In this section, we will introduce an approach that takes a set of runs as a starting point. From this set of runs, a system net is constructed. Moreover, we need to find a mapping from all the events and conditions in the causal nets to the transitions and places in the system net. From Definition 2.2, we know that there should exist a bijection between all places in the pre- or post-set of an event in some causal net and the pre- or post-set of a transition in a system net. *Therefore, two conditions belonging to the pre- or post-set of an event should not be mapped onto the same label.* This restriction is in fact merely another way to express the fact that our P/T-nets do not allow for more than one edge between a place and a transition or vice versa. More generally, we define a labelling function on the nodes of a graph as a function that does not give the same label to two nodes that have a common element in their pre-sets or a common element in their post-sets.

Definition 3.1. (Labelling function)
Let μ be a set of labels. Let $G = (N, E)$ be a graph. Let $R = \{(n_1, n_2) \subseteq N \times N \mid n_1 \overset{G}{\bullet} \cap n_2 \overset{G}{\bullet} \neq \emptyset \vee \overset{G}{\bullet} n_1 \cap \overset{G}{\bullet} n_2 \neq \emptyset\}$. We define $f : N \to \mu$ to be a *labelling function* if f is a coloring function on the graph (N, R).

We focus on the aggregation of runs that originate from a Petri net with clearly defined starting state and completion state, i.e. processes that describe a lifespan of some case. This assumption is very natural in the context of workflow management systems. However, it applies to many other domains where processes are instantiated for specific cases. Hence, we will limit ourselves to a special class of Petri nets, namely workflow nets.

Definition 3.2. (Workflow nets)
A P/T-net $N = (P, T, F, M_0)$ is a *workflow net* (WF-net) if:

1. *object creation*: P contains an input place p_{ini} such that $\bullet p_{ini} = \emptyset$,
2. *object completion*: P contains an output place p_{out} such that $p_{out}\bullet = \emptyset$,
3. *connectedness*: there is a path from p_{ini} to every node and from every node to p_{out},
4. *initial marking*: $M_0 = [p_{ini}]$, i.e. the initial marking marks only p_{ini}.

As a consequence, a WF-net has exactly one one input place. When looking at a run of a WF-net, we can therefore conclude that there is exactly one condition containing a token initially and all other conditions do not contain tokens. A set of causal nets fulfilling this condition and some structural consequences is called a *causal set*.

Definition 3.3. (Causal set)
Let $n \in \mathbb{N}$ and let $\Phi = \{(C_i, E_i, K_i, S_i) \mid 0 \leq i < n\}$ be a set of causal nets. We call this set a *causal set* if and only if, for $0 \leq i < n$ holds:

- all sets C_i, E_i, K_i are disjoint,
- for $0 \leq i \leq n$, $\sum_{c \in C_i} S_i(c) = 1$, i.e. exactly one condition has an empty pre-set,
- for $e \in E_i$, if $S_i(c) = 1$ for some $c \in \bullet e$ then $\{c\} = \bullet e$,
- for $e \in E_i$, if $c\bullet = \emptyset$ for some $c \in e\bullet$, then $\{c\} = e\bullet$.

The concept of constructing a system net from a causal set is called *aggregation*. This concept can be applied if we assume that each causal net in the given set can be called a run of some system net. From Definition 2.2 we know that we need two mappings α and β satisfying the two properties mentioned. Using the definition of a system net and the relation between system nets and runs, we can conclude that any aggregation algorithm should have the following functionality:

- it should provide the set of places P of the system net,
- it should provide the set of transitions T of the system net,
- it should provide the flow relation F of the system net,
- it should provide the initial marking M_0 of the system net,
- for each causal net in the causal set, it should provide the mappings $\alpha_i : C_i \to P$ and $\beta_i : E_i \to T$, in such a way that for all causal nets, $\alpha_i(S_i)$ is the same (i.e. they have the same initial marking) and they induce bijections between pre- and post-sets of events and their corresponding transitions.

Each event that appears in a causal net has a corresponding transition in the original system net. Moreover, bijections exist between the pre- and post-sets of this event and the corresponding transitions. In order to express this in terms of labelling functions of causal nets, we formalize this concept using the notion of *transition equivalence*.

Definition 3.4. (Transition equivalence)
Let μ, ν be two disjoint sets of labels. Let $\Phi = \{N_i = (C_i, E_i, K_i, S_i) \mid 0 \leq i < n\}$ be a causal set, and let $\Psi = \{(\alpha_i : C_i \to \mu, \beta_i : E_i \to \nu) \mid 0 \leq i < n\}$ be a set of labelling functions of (C_i, E_i, K_i, S_i). We define (Φ, Ψ) to *respect transition equivalence* if and only if for each $e_i \in E_i$ and $e_j \in E_j$ with $\beta_i(e_i) = \beta_j(e_j)$ the following holds:

- for each $(c_i, e_i) \in K_i$ we have a $(c_j, e_j) \in K_j$ such that $\alpha_i(c_i) = \alpha_j(c_j)$,
- for each $(e_i, c_i) \in K_i$ we have a $(e_j, c_j) \in K_j$ such that $\alpha_i(c_i) = \alpha_j(c_j)$.

4 Aggregation with Known Labels

In this section, we present an aggregation algorithm that assumes that we know all mapping functions, and that these mapping functions adhere to the definition of a run. To illustrate the aggregation process, we make use of a running example. Consider Figure 2 where four part of runs are shown. We assume that the events A,B,C,D,E,F and G do not appear in any other part of each run. Our first aggregation algorithm is

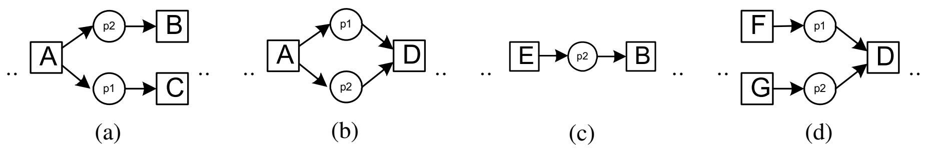


Fig. 2. Four examples of parts of runs.

called the ALK aggregation algorithm (short for "All Labels Known"). This algorithm assumes all information to be present, such as in Figure 2, i.e. it assumes known labels for events and known labels for conditions. These labels refer to concrete transitions and places in the aggregated system net.

Definition 4.1. (ALK aggregation algorithm)
Let μ, ν be two disjoint sets of labels. Let Φ be a causal set of size n with causal nets (C_i, E_i, K_i, S_i) $(0 \leq i < n)$.

Furthermore, let $\{(\alpha_i : C_i \to \mu, \beta_i : E_i \to \nu) \mid 0 \leq i < n\}$ be a set of labelling functions respecting transition equivalence, such that for all causal nets $\alpha_i(S_i)$ is the same. We construct the system net (P, T, F, M_0) belonging to these runs as follows:

- $P = \bigcup_{0\leq i<n} \text{rng}(\alpha_i)$ is the set of places (note that $P \subseteq \mu$)[1],
- $T = \bigcup_{0\leq i<n} \text{rng}(\beta_i)$ is the set of transitions (note that $T \subseteq \nu$),
- $F = \bigcup_{0\leq i<n}\{(\alpha_i(c), \beta_i(e)) \in P \times T \mid (c, e) \in K_i \cap (C_i \times E_i)\} \cup$
  $\bigcup_{0\leq i<n}\{(\beta_i(e), \alpha_i(c)) \in T \times P \mid (e, c) \in K_i \cap (E_i \times C_i)\}$
  is the flow relation,
- $M_0 = \alpha_0(S_0)$ is the initial marking.

The result of the ALK aggregation algorithm from Definition 4.1 for the parts presented in Figure 2 is shown in Figure 3. Another example is given in Figure 1(a).

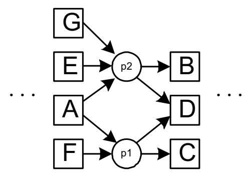


Fig. 3. The aggregated Petri net.

The aggregated net shown in Figure 3 can actually generate the runs of Figure 2. This is always the case after applying the ALK aggregation algorithm:

[1] With rng we denote the range of a function, i.e. rng$(f) = \{f(x) \mid x \in \text{dom}(f)\}$

Property 4.2. (The ALK algorithm is correct)
For $0 \leq i < n, N_i = (C_i, E_i, K_i, S_i), (N_i, \alpha_i, \beta_i)$ is indeed a run of $\sigma = (P, T, F, M_0)$ (i.e., the requirements stated in Definition 2.2 are fulfilled).

The ALK algorithm is a rather trivial aggregation over a set of runs. However, it is assumed that the mapping functions α_i and β_i are known for each causal net. Furthermore, we assume two sets of labels μ and ν to be known. However, when applying these techniques in the context of process mining, it is often not realistic to assume that all of these are present. Therefore, in the remainder of this paper, we relax some of these assumptions to obtain more usable process mining algorithms.

5 Aggregation with Duplicate or Missing Transition Labels

In this section, we will assume that the causal set used to generate the system net and the labelling functions do not respect transition equivalence. We introduce an algorithm to change the labelling function for events in such a way that this property holds again. In the domain of process mining, the problem of so-called "duplicate transitions" (i.e. several transitions with the same label) is well-known (cf. [10–12]). Therefore, there is a need for algorithms to find out which events actually belong to which transition. We assume that we have causal nets with labelling functions, where some events have the same label, even though they may refer to different transitions (see Figure 4).

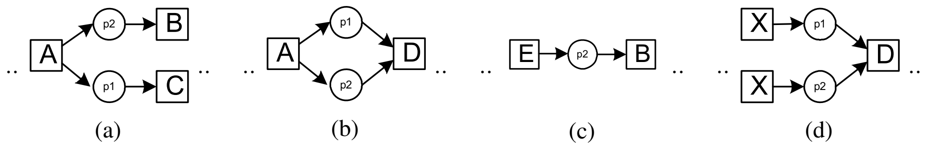


Fig. 4. Four examples of parts of runs.

In terms of an aggregation algorithm, the problem of duplicate labels translates to the situation where the property of transition equivalence is not satisfied. Since the aggregation algorithm presented in the previous section only works if this property holds, we provide an algorithm to redefine the labelling functions for events.

Definition 5.1. (Relabelling algorithm)
Let μ, ν be two disjoint sets of labels. Let $\Phi = \{N_i \mid 0 \leq i < n \wedge N_i = (C_i, E_i, K_i, S_i)\}$ be a causal set and let $\Psi = \{(\alpha_i : C_i \to \mu, \beta_i : E_i \to \nu) \mid 0 \leq i < n\}$ be a set of labelling functions in (C_i, E_i, K_i, S_i) such that $\alpha_i(S_i)$ is the same for all causal nets. Furthermore, assume that μ and ν are minimal, i.e. $\bigcup_{0 \leq i < n} \text{rng}(\alpha_i) = \mu$ and $\bigcup_{0 \leq i < n} \text{rng}(\beta_i) = \nu$. Let $E^\star = \bigcup_{0 \leq i < n} E_i$ be the set of all events in the causal set.

We define the relabelling algorithm as follows:

1. Define $\bowtie \subseteq E^\star \times E^\star$ as an equivalence relation on the elements of $E^\star$ in such a way that $e_i \bowtie e_j$ with $e_i \in E_i$ and $e_j \in E_j$ if and only if $\beta_i(e_i) = \beta_j(e_j)$, $\alpha_i(\overset{N_i}{\bullet} e_i) = \alpha_j(\overset{N_i}{\bullet} e_j)$, and $\alpha_i(e_i \overset{N_i}{\bullet}) = \alpha_j(e_j \overset{N_i}{\bullet})$.
2. For each $e \in E^\star$, we say $\text{eqvl}(e) = \{e' \in E^\star \mid e \bowtie e'\}$.
3. Let ν' be the set of equivalence classes of $\bowtie$, i.e. $\nu' = \{\text{eqvl}(e) \mid e \in E^\star\}$.

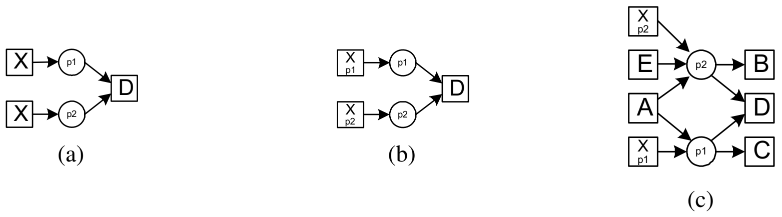


Fig. 5. The original and relabelled part of Figure 4(d) and a part of the aggregated net.

4. For all causal nets (C_i, E_i, K_i, S_i) and labelling functions α_i, define a labelling function $\beta'_i : E_i \to \nu'$ such that for an event e_i, $\beta'_i(e_i) = \text{eqvl}(e_i)$, i.e. it returns the equivalence class of $\bowtie$ containing e_i.

After re-labelling the events, the part of the run shown in Figure 4(d) is relabelled to include the pre- and post-conditions. Figure 5(a) shows the fragment before relabelling, whereas Figure 5(b) shows the fragment after relabelling. (We only show the relabelling with respect to the post-conditions.) Applying the ALK algorithm of Definition 4.1 to the relabelled runs yields the result as shown in Figure 5(c). Note that we do not show the ν' labels explicitly, i.e. B refers to the equivalence class of events labelled B.

What remains to be shown is that our algorithm does not only work for our small running example, but also in the general case. The only difference between the assumptions in Definition 4.1 and Definition 5.1 is the requirement with respect to transition equivalence. Therefore, if suffices to show that after applying the relabelling algorithm on a causal set, we can establish transition equivalence.

Property 5.2. (Transition equivalence holds after relabelling)
Let μ, ν be two disjoint sets of labels. Let $\Phi = \{(C_i, E_i, K_i, S_i) \mid 0 \leq i < n\}$ be a causal set, and let $\Psi = \{(\alpha_i : C_i \to \mu, \beta_i : E_i \to \nu) \mid 0 \leq i < n\}$ be a set of labelling functions in (C_i, E_i, K_i, S_i), such that $\alpha_i(S_i)$ is the same for all causal nets. After applying the relabelling algorithm, the property of transition equivalence holds for (Φ, Ψ'), with $\Psi' = \{(\alpha_i : C_i \to \mu, \beta'_i : E_i \to \nu') \mid 0 \leq i < n\}$, and β'_i as defined in Definition 5.1.

The algorithm presented above is capable of finding events that have the same label, but correspond to different transitions in the system net. When *no transition labels are known at all*, it can be applied to *find all transition labels*, by using an initial $\nu = \{\tau\}$ and initial mapping functions β_i, mapping everything onto τ. However, in that case, no distinction can be made between events that have the same pre- and post-set, but should have different labels. After applying this relabelling algorithm, the ALK algorithm of Section 4 can be used to find the system net belonging to the given causal nets.

6 Aggregation with Unknown Place Labels

In Section 5, we have shown a way to identify the transitions in a system net, based on the labels of events in causal nets. However, what if condition labels are not known? Notice that the difference to other approaches based on partial orders is that here we do

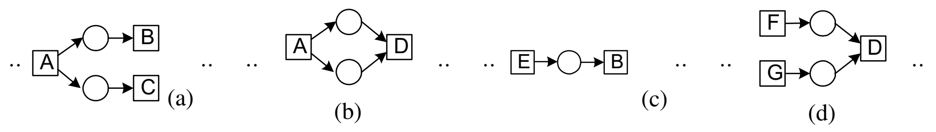


Fig. 6. Four examples of parts of runs.

know the conditions constituting the order between events but do *not* know which two conditions refer to a token in the same place of the P/T-net representing the process.

So, in this section, we take one step back. We assume all events to refer to the correct transition and try to identify the labels of conditions. We introduce an algorithm to aggregate causal nets to a system net, such that the original causal nets are indeed runs of that system net. In Figure 6, we again show our small example of the aggregation problem, only this time there are no labels for conditions $p1$ and $p2$, which we did have in Figures 2 and 4.

Consider the four runs of Figure 6. Remember that they are *parts* of causal nets, in such a way that the tasks A, B, C, D, E, F and G do not appear in any other way in another causal net. In contrast to the algorithms presented in previous sections, we cannot always derive a unique aggregated system net for causal nets if we do not have labels for the conditions. Instead, we define an *aggregation class*, describing a class of WF-nets that could have generated these causal nets. The following table shows some requirements all WF-nets in the aggregation class of our example should satisfy.

Fragment	Conclusions
Fig. 6 (a)	$A\bullet = \bullet B \uplus \bullet C$
Fig. 6 (b)	$A\bullet = \bullet D$
Fig. 6 (c)	$E\bullet = \bullet B$
Fig. 6 (d)	$F\bullet \uplus G\bullet = \bullet D$

This information is derived using the concept of a segment, which can be considered to be the context of a condition in a causal net.

Definition 6.1. (Segment)
Let $N = ((C, E, K), S_0)$ be a causal net and let $N' = (C', E_{in}, E_{out})$ be such that $C' \subseteq C$, $E_{in} \cup E_{out} \subseteq E$ and $E_{in} \neq \emptyset$ and $E_{out} \neq \emptyset$. We call N' a *segment* if:

- for all $c \in C'$ holds that $\bullet c \subseteq E_{in}$ and $c\bullet \subseteq E_{out}$, and
- for all $e \in E_{in}$ holds that $e\bullet \subseteq C'$, and
- for all $e \in E_{out}$ holds that $\bullet e \subseteq C'$, and
- the subgraph of N made up by $C' \cup E_{in} \cup E_{out}$ is connected.

We call the events in E_{in} the input events and the events in E_{out} the output events. Furthermore, a segment is called *minimal* if C' is minimal, i.e. if there does not exist a segment $N'' = (C'', E'_{in}, E'_{out})$ with $C'' \subset C'$ and $C'' \neq \emptyset$.

For the fragments of Figure 6, it is easy to see that each of them contains only one minimal segment, where the input events are the events on the left hand side and the output events are the events on the right hand side.

The meaning of a segment is as follows. If we have a run and a segment in that run, then we know that after each of the events in the input set of the segment occurred,

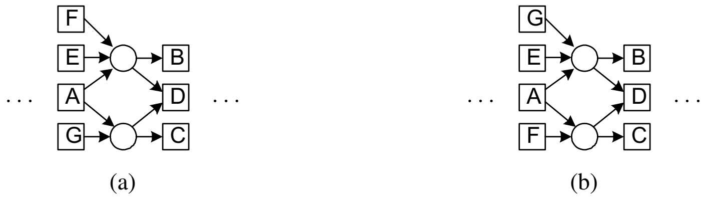


Fig. 7. Two aggregated nets.

all the events in the output set occurred in the execution represented by this run. This translates directly to a marking in a system net, since the occurrence of a set of transitions would lead to some marking (i.e. a bag over places), which enables another set of transitions. Furthermore, each transition only produces one token in each output place. Combining this leads to the fact that for each minimal segment in a causal net the bag of places following the transitions corresponding to the input events of the segment should be the same as the bag of places preceding the transitions corresponding to the output set of events.

Clearly, when looking only at these fragments, what we are looking for are the places that should be put between tasks A, E, F and G on the one hand, and B, C and D on the other hand. Therefore, we only focus on this part of the causal nets. For this specific example, there are two possibilities, both of which are equally correct, namely the two WF-net fragments shown in Figure 7.

From the small example, we have seen that it is possible to take a set of causal nets without labels for any of the conditions (but with labels for all the events) and to define a class of WF-nets that could be system nets of the causal nets. In the remainder of this section, we show that this is indeed possible for all causal sets. For this, we first introduce the NCL algorithm.

6.1 NCL Algorithm

Before presenting the NCL algorithm (which stands for "No Condition Labels"), we first take a look at a more intuitive example. Consider Figure 8, where we present three causal nets, each of which corresponds to a paper review process. In the first causal net, three reviewers are invited to review the paper and after the three reviews are received, the paper is accepted. In the second causal net, only two reviews are received (the third one is not received on time), but the paper is rejected nonetheless (apparently the two reviewers that replied rejected the paper). In the third example only one review is received in time, and therefore an additional reviewer is invited, which hands in his review in time, but does not accept the paper.

As we stated before, we define an aggregation class of a causal set that contains all WF-nets that are capable of generating the causal nets in the causal set. The information needed for this aggregation class comes directly from the causal nets, using minimal segments. In Table 1, we present the conclusions we can draw based on the three causal nets. In this table we consider *bags* of pre- and post-sets of transitions in the aggregation class. The information in this table is obtained from the causal nets in the following way. Consider for example Figure 8(a), where `invite reviewers` is followed by

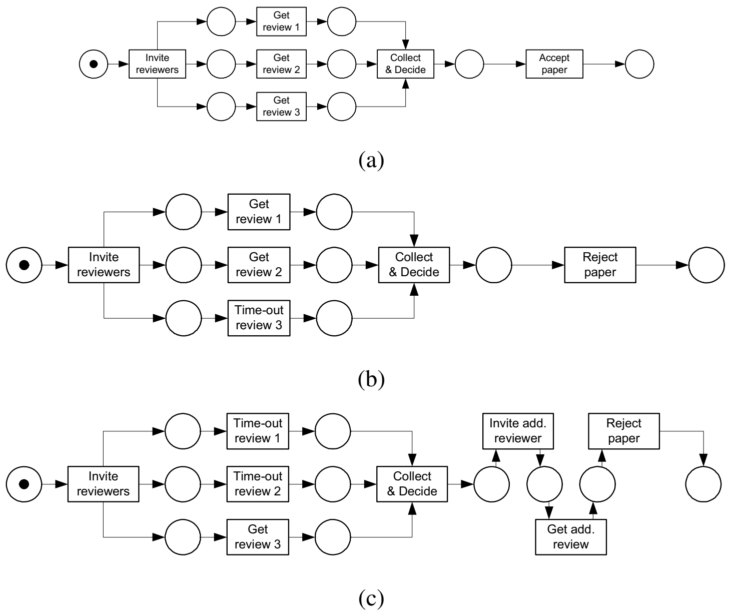


Fig. 8. Three causal nets of a review process of a paper.

`Get review 1`, `Get review 2` and `Get review 3`. This implies that the bag of output places of `invite reviewers`, should be the same as the sum over the bags of the input places of `Get review 1`, `Get review 2` and `Get review 3`.

Definition 6.2. (Aggregation class)
Let $\Phi = \{(C_i, E_i, K_i, S_i) \mid 0 \leq i < n\}$ be a causal set, and let $\sigma = (P, T, F, M_0)$ be a marked WF-net. For each causal net, let $\beta_i : E_i \rightarrow T$ be a mapping from the events of that causal net to T, such that β_i is a labelling function for (C_i, E_i, K_i, S_i). We define A_Φ, the *aggregation class* of Φ, as the set of all pairs $(\sigma, \mathcal{B})$ such that the following conditions are satisfied:

1. $T = \bigcup_{0 \leq i < n} \text{rng}(\beta_i)$ is the set of transitions, i.e. each transition appears as an event at least once in some causal net,
2. $\mathcal{B}$ is the set of all labelling functions, i.e. $\mathcal{B} = \{\beta_i \mid 0 \leq i < n\}$. We use $\beta_i \in \mathcal{B}$ to denote the labelling function for events belonging to $(C_i, E_i, K_i, S_i) \in \Phi$,
3. For all $p \in P$ holds that $\overset{\sigma}{\bullet} p \cup p \overset{\sigma}{\bullet} \neq \emptyset$,
4. $M_0 = [p_{ini}]$ and $\overset{\sigma}{\bullet} p_{ini} = \emptyset$,
5. For each causal net $\gamma = (C_i, E_i, K_i, S_i)$, with $e \in E_i$ and $\beta_i(e) = t$ holds that if $S_i(\overset{\gamma}{\bullet} e) = 1$ then $p_{ini} \in \overset{\sigma}{\bullet} t$,
6. For each causal net $\gamma = (C_i, E_i, K_i, S_i)$, with $e \in E_i$ and $\beta_i(e) = t$ holds that $|t \overset{\sigma}{\bullet}| = |e \overset{\gamma}{\bullet}|$ and $|\overset{\sigma}{\bullet} t| = |\overset{\gamma}{\bullet} e|$,
7. For each causal net $\gamma = (C_i, E_i, K_i, S_i)$, with $e \in E_i$, $\beta_i(e) = t$ and $T' \subseteq T$ holds that $|t \overset{\sigma}{\bullet} \cap \bigcup_{t' \in T'} (\overset{\sigma}{\bullet} t')| \geq \sum_{e' \in E_i, \beta(e') \in T'} |e \overset{\gamma}{\bullet} \cap \overset{\gamma}{\bullet} e'|$,

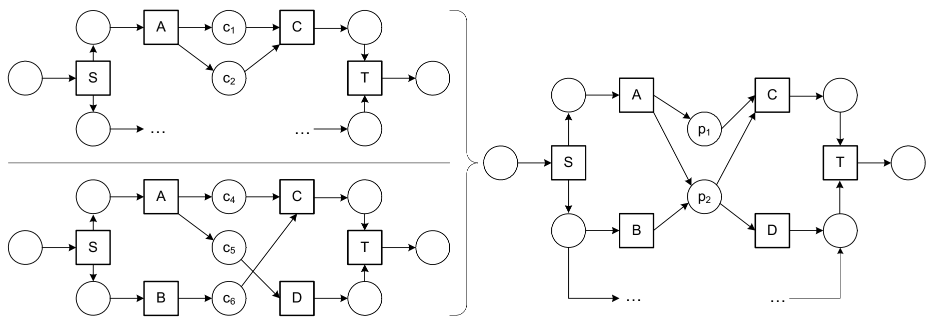


Fig. 9. Example explaining the use of bags.

8. For each causal net $\gamma = (C_i, E_i, K_i, S_i)$, with $e \in E_i$, $\beta_i(e) = t$ and $T' \subseteq T$ holds that $|\bigcup_{t' \in T'} (t' \overset{\gamma}{\bullet}) \cap \overset{\gamma}{\bullet} t| \geq \sum_{e' \in E_i, \beta(e') \in T'} |e' \overset{\gamma}{\bullet} \cap \overset{\gamma}{\bullet} e|$,
9. For each causal net $\gamma = (C_i, E_i, K_i, S_i)$ and any minimal segment (C'_i, E_{in}, E_{out}) of γ, holds that $\biguplus_{e \in E_{in}} \big(\beta_i(e) \overset{\gamma}{\bullet}\big) = \biguplus_{e \in E_{out}} \big(\overset{\gamma}{\bullet} \beta_i(e)\big)$.

Figure 9 is used to gain more insight into part 9 of Definition 6.2. In the lower causal net of that figure, there is a token travelling from A to D and from B to C. The upper causal net only connects A and C. Assuming that these are the only causal nets in which these transitions appear, we know that the conditions between A and D and between B and C should represent a token in the same place, since there is a minimal segment $(\{c_4, c_5, c_6\}, \{A, B\}, \{C, D\})$ in the lower causal net and therefore, $A\bullet \uplus B\bullet = \bullet C \uplus \bullet D = [p_1, 2p_2]$.

The NLC algorithm takes a set of runs without condition labels as a starting point. From these runs, an aggregation class of WF-nets is defined. If the runs were generated from some sound WF-net, then the WF-net itself is in that aggregation class. We conclude this section with an elaborate example of the application of the NCL algorithm.

Consider the four causal nets presented in Figure 10. These causal nets originate from a workflow system in which two activities need to be performed. These activities are labelled `L` and `R`. However, in the workflow design, there are several options. First, the system initializes the two activities trough event `Init LR`. Then, a person can decide to perform both activities at once, which is represented by the event `Do LR`. When both activities have been performed, the workflow can be finished through event `exit LR`. However, in a typical workflow environment, people can make mistakes and therefore, in Figure 10(b), both activities have been undone, thus generating events `Undo L` and `Undo R`. Finally, in Figure 10 (c) and (d) it is shown that the workflow system allows for the two activities to be executed separately, through `Do L` and `Do R`. To keep things interesting, the last causal net belongs to a case in the workflow system that is not finished yet. However, this set of causal nets still conforms to the definition of a causal set (i.e. Definition 3.3).

Using the NCL algorithm given by Definition 6.2, we can generate the aggregation class of the causal set of Figure 10. In fact, this aggregation class only contains one workflow net, namely the workflow net shown in Figure 11.

The workflow net in Figure 11 actually allows for more behaviour than is shown in the four causal nets of Figure 10. It is for example possible to execute a long sequence

Table 1. Information derived from review example.

Causal net	Conclusions on transitions in the aggregation class
Fig. 8 (a)	$\bullet$“Invite reviewers” $= [p_{ini}]$
	“Invite reviewers”$\bullet$ $= \bullet$“Get review 1” $\uplus$ $\bullet$“Get review 2” $\uplus$ $\bullet$“Get review 3”
	“Get review 1”$\bullet$ $\uplus$ “Get review 2”$\bullet$ $\uplus$ “Get review 3”$\bullet$ $= \bullet$“Collect & Decide”
	“Collect & Decide”$\bullet$ $= \bullet$“Accept paper”
	$\lvert$“Accept paper”$\bullet\rvert$ $= 1$
Fig. 8 (b)	$\bullet$“Invite reviewers” $= [p_{ini}]$
	“Invite reviewers”$\bullet$ $= \bullet$“Get review 1” $\uplus$ $\bullet$“Get review 2” $\uplus$ $\bullet$“Time-out review 3”
	“Get review 1”$\bullet$ $\uplus$ “Get review 2”$\bullet$ $\uplus$ “Time-out review 3”$\bullet$ $= \bullet$“Collect & Decide”
	“Collect & Decide”$\bullet$ $= \bullet$“Reject paper”
	$\lvert$“Reject paper”$\bullet\rvert$ $= 1$
Fig. 8 (c)	$\bullet$“Invite reviewers” $= [p_{ini}]$
	“Invite reviewers”$\bullet$ $= \bullet$“Time-out review 1” $\uplus$ $\bullet$“Time-out review 2” $\uplus$ $\bullet$“Get review 3”
	“Time-out review 1”$\bullet$ $\uplus$ “Time-out review 2”$\bullet$ $\uplus$ “Get review 3”$\bullet$ $= \bullet$“Collect & Decide”
	“Collect & Decide”$\bullet$ $= \bullet$“Invite add. reviewer”
	“Invite add. reviewer”$\bullet$ $= \bullet$“Get add. review”
	“Get add. review”$\bullet$ $= \bullet$“Reject paper”
	$\lvert$“Reject paper”$\bullet\rvert$ $= 1$

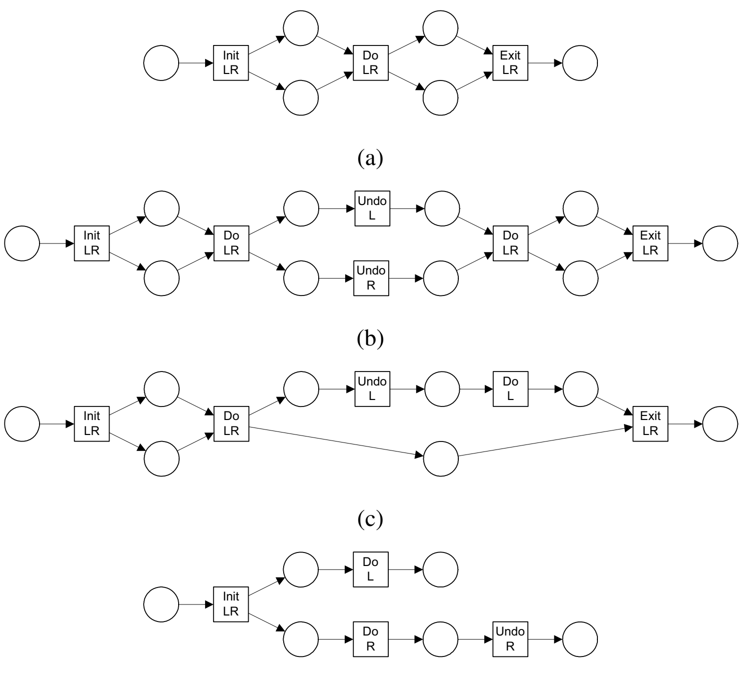


Fig. 10. Four causal nets without condition labels.

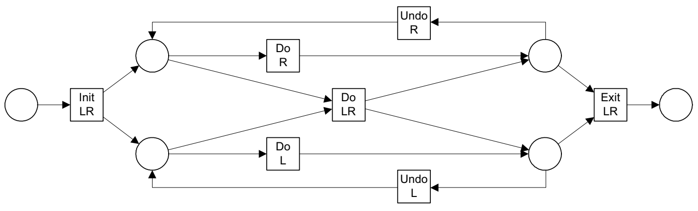


Fig. 11. The only element of the aggregation class of the four nets of Figure 10.

of `Do L` and `Undo L`, which is not shown in the causal nets. Therefore, this example once more shows that each net in the aggregation class can actually generate the runs of the causal set it was constructed from, but it might be able to generate more runs.

7 Conclusion and Future Work

In this paper, we looked at process mining from a new perspective. Instead of starting with a set of traces, we started with runs, which constitute partial orders on events. We presented three algorithms to generate a Petri net from these runs. The first algorithm assumes that, for each run, all labels of both conditions and events are known. The second algorithm relaxes this by assuming that some transitions can have the same label (i.e. duplicate labels are allowed in the system net). This algorithm can also be used if only condition/place-labels were recorded. Finally, we provided an algorithm that does not require condition labels, i.e. the event/transition labels are known, the condition/place labels are unknown and duplicate transition labels are not allowed.

The results presented in this paper hold for a subclass of Petri nets, so-called WF-nets. However, the first two algorithms presented here can easily be generalized to be applicable to any Petri net. For the third algorithm this can also be done, however, explicit knowledge about the initial marking would be required. When taking a set of runs as a starting point, this knowledge is not present in the general case.

References

1. Desel, J.: Validation of Process Models by Construction of Process Nets. In Aalst, W., Desel, J., Oberweis, A., eds.: Business Process Management: Models, Techniques, and Empirical Studies. Volume 1806 of Lecture Notes in Computer Science, Springer-Verlag, Berlin (2000) 110–128
2. Aalst, W., van Dongen, B., Herbst, J., Maruster, L., Schimm, G., Weijters, A.: Workflow Mining: A Survey of Issues and Approaches. Data and Knowledge Engineering **47**(2) (2003) 237–267
3. Smith, E.: On net systems generated by process foldings. In Rozenberg, G., ed.: Advances in Petri Nets. Volume 524 of Lecture Notes in Computer Science, Springer-Verlag, Berlin (1991) 253–276
4. Desel, J., Erwin, T.: Hyprid specifications: looking at workflows from a run-time perspective. Computer Systems Science and Engineering **5** (2000) 291–302
5. Roychoudhury, A., Thiagarajan, P.: Communicating transaction processes. In Lilius, J., Balarin, F., Machado, R., eds.: Proceedings of Third International Conference on Application of Concurrency to System Design (ACSD2003), IEEE Computer Society (2003) 157–166
6. Harel, D., Kugler, H., Pnueli, A.: Synthesis revisited: Generating statechart models from scenario-based requirements. In Kreowski, H.J., Montanari, U., Orejas, F., Rozenberg, G., Taentzer, G., eds.: Formal Methods in Software and Systems Modeling. Volume 3393 of Lecture Notes in Computer Science, Springer (2005) 309–324
7. Lorenz, R., Juhás, G.: Towards synthesis of petri nets from scenarios. In Donatelli, S., Thiagarajan, P.S., eds.: ICATPN. Volume 4024 of Lecture Notes in Computer Science, Springer (2006) 302–321
8. Dongen, B., Aalst, W.: Multi-phase Process mining: Building Instance Graphs. In Atzeni, P., Chu, W., Lu, H., Zhou, S., Ling, T., eds.: Conceptual Modeling - ER 2004. Volume 3288 of Lecture Notes in Computer Science, Springer-Verlag, Berlin (2004) 362–376
9. Dongen, B., Aalst, W.: Multi-phase Process mining: Aggregating Instance Graphs into EPCs and Petri Nets. In: PNCWB 2005 workshop. (2005) 35–58
10. Aalst, W., de Medeiros, A.A., Weijters, A.: Genetic Process Mining. In: Proceedings of the 26th International Conference on Applications and Theory of Petri Nets. Volume 3536 of Lecture Notes in Computer Science (2005)
11. de Medeiros, A.A., Weijters, A., van der Aalst, W.: Genetic Process Mining: A Basic Approach and its Challenges. In: BPM - BPI workshop. (2005)
12. Herbst, J., Karagiannis, D.: Workflow mining with InWoLvE. Comput. Ind. **53**(3) (2004) 245–264

Folding Partially Ordered Runs

Robin Bergenthum, Sebastian Mauser

Department of Software Engineering, FernUniversität in Hagen
{robin.bergenthum,sebastian.mauser}@fernuni-hagen.de

Abstract. In this paper we present a folding algorithm to construct a process model from a specification given by a set of scenarios. Each scenario is formalized as a partially ordered set of activities of the process. As a process model we consider Event-driven Process Chains which are well established in the domain of business process modeling. Assuming that a specification describes valid behavior of the process to be modeled, the crucial requirement is to get a process model which in any case is able to execute all input scenarios.

1 Introduction

Business process modeling and management has attracted increasing attention in recent years. The usual approach to process model construction is that a domain expert edits a formal process model. Simulation tools generate single scenarios of that process model which can also be viewed as formal objects. Then, the expert checks whether the scenarios of the model correspond to possible scenarios of the intended process. In the negative case, he changes the process model and iteratively repeats the simulation.

In the papers [1, 2] a proceeding in the opposite direction has been suggested. It is assumed that the domain experts know some or all scenarios of a process to be modeled better than the process itself. Actually, experts might also know parts of the process model including parts of its branching structure, but in this case scenarios can be derived from this partially known process model. Experience shows that in various application areas processes are specified in terms of example scenarios (an evidence is the commonly used sequence diagrams in UML to specify scenarios).

In a first step of the approach from [1, 2], the scenarios of a process are specified by domain experts. Then, these scenarios have to be formalized yielding formal runs. Formalization on the instance level of single scenarios is an intuitive task. In our setting, the scenarios are formalized in terms of labeled partial orders (LPOs) representing occurrences of process activities and their mutual order relation. Under the name instance Event-driven Process Chains (iEPCs), a similar concept is also applied in the industrial ARIS PPM tool [3, 4]. As in the case of iEPCs, we do not allow auto-concurrency, i.e. we assume that the scenario descriptions of the domain experts do not contain two concurrent occurrences of the same activity.

In a second step, a process model is automatically generated from the given formal runs. For this step, in previous work [2] it was suggested to build on synthesis algorithms generating Petri nets exactly having the behavior given by a set of LPOs. Although using an exact synthesis algorithm as the starting point for this step has several advantages such as reliable results, there are also several problems w.r.t. practical applicability. Namely, a precise reproduction is mostly not desired in practice due to incomplete information, the performance of synthesis is low and the resulting nets are sometimes difficult to understand.

In the related field of process mining [5], simplified construction algorithms are successful. Against this background, in this paper we develop a simplified construction algorithm for the automatic generation of a process model from a set of example scenarios given by LPOs. We here use Event-driven Process Chains (EPCs) to model processes. The idea of our approach is to fold the scenarios to one EPC model similar as in [4] (the approach has also similarities to [6, 5]). The resulting EPC represents all the direct dependencies given by the LPOs. The folding algorithm is efficient and generates intuitive models. Still, it has the important property that all the explicitly specified example LPOs are executable scenarios of the generated EPC.

To formally define the folding algorithm and to prove the latter important result, we first provide formal definitions in the next section, in particular we introduce the notion of an LPO executable w.r.t. an EPC. Then, in Section 3 we introduce and discuss the folding algorithm generating an EPC from a set of LPOs. Finally, Section 4 concludes the paper.

2 Partially Ordered Runs of EPCs

EPCs are an intuitive modeling language for business processes [7, 8]. Since the language was initially not intended for formal specifications, the formal definitions of EPCs in literature show some differences. We here give a short and simple definition of an EPC.

Definition 1. *An EPC-structure is a five-tuple $epc = (\mathcal{A}, \mathcal{E}, \mathcal{C}_{xor}, \mathcal{C}_{and}, \mathcal{D})$ where $\mathcal{A}$ is a finite set of activities (also called functions), $\mathcal{E}$ is a finite set of events, $\mathcal{C}_{xor}$ resp. $\mathcal{C}_{and}$ are finite sets of XOR- resp. AND-connectors and $\mathcal{D} \subseteq (\mathcal{A} \cup \mathcal{E} \cup \mathcal{C}_{xor} \cup \mathcal{C}_{and}) \times (\mathcal{A} \cup \mathcal{E} \cup \mathcal{C}_{xor} \cup \mathcal{C}_{and})$ is a set of directed edges connecting activities, events and connectors. An EPC-structure is an Event-driven Process Chain (EPC) if:*

(i) Events have at most one incoming and one outgoing edge.
(ii) Activities have precisely one incoming and one outgoing edge.
(iii) Connectors have either one incoming edge and multiple outgoing edges, or multiple incoming edges and one outgoing edge.

Activities, events and connectors are also called nodes of an EPC. Given a node n, the set of edges $\bullet n := \{(n', n) \in \mathcal{D} \mid n' \in \mathcal{A} \cup \mathcal{E} \cup \mathcal{C}_{xor} \cup \mathcal{C}_{and}\}$ is called preset of n and the set of edges $n\bullet := \{(n, n') \in \mathcal{D} \mid n' \in \mathcal{A} \cup \mathcal{E} \cup \mathcal{C}_{xor} \cup \mathcal{C}_{and}\}$ is called postset of n. Furthermore, an event having an empty preset is called initial event.

An example of an EPC is shown in Figure 1. Activities are illustrated by rounded rectangles, events by hexagons and connectors by circles.

For simplicity, this definition of an EPC does not regard OR-connectors, since they are not necessary for our considerations later on. Moreover, syntactically it is not as restrictive as other definitions found in the literature. The main differences are described in the following remark. However, these differences are not relevant for semantical issues.

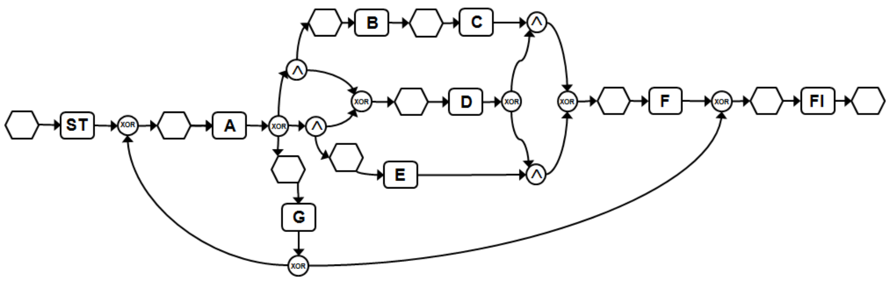


Fig. 1. EPC.

Remark 1. We omit the stylistic requirements for EPCs that in every path activities and events alternate, that an XOR-split must not succeed an event and that there should be no cycle of control flow that consists of connectors only.

In the following, we define a partial order semantics for EPCs. We introduce such semantics analogously as in the case of Petri nets, see e.g. [9].

First, we define an occurrence rule for nodes of an EPC. Thereby, we consider the notion of a marking representing a state of an EPC analogously as in [8] by assigning tokens resp. process folders to the edges of the EPC. A problem is that the semantics of XOR-joins (and OR-joins) of EPCs is not clear [8]. In the literature there are different occurrence rules for XOR-joins. In this paper, we consider the simple "Petri net semantics" as given in [7] where an XOR-join can fire if one of its incoming edges contains a process folder. In this way, the exclusiveness of an XOR-operator is not regarded. However, an appropriate approach for considering the exclusiveness meaning of XOR requires difficult non-local behavioral definitions [8] and there is not yet a standard approach for solving this problem.

Based on our occurrence rule for single nodes we then define a step occurrence rule for sets of nodes. Since each edge has exactly one successor node, there are no conflicts between nodes regarding the consumption of tokens. Consequently, a set of nodes is concurrently executable if each node of the set is executable. Using this step occurrence rule we further define the notion of an executable sequence of sets of nodes which we then restrict to activities by applying an appropriate projection.

Finally, we define the executability of an LPO as in the case of Petri nets (see [9]) by requiring that each step sequence allowed by the LPO, i.e. being a sequentialization of the LPO, is executable w.r.t. the given EPC.

A marking of an EPC is formally defined as follows.

Definition 2. *Given an EPC* $epc = (\mathcal{A}, \mathcal{E}, \mathcal{C}_{xor}, \mathcal{C}_{and}, \mathcal{D})$*, a marking of* epc *is a function* $m : \mathcal{D} \to \mathbb{N} = \{0, 1, 2, \ldots\}$*. A pair* (epc, m) *where* epc *is an EPC and* m *is a marking is called marked EPC.*

A directed edge $d \in \mathcal{D}$ *is called marked if* $m(d) > 0$*, marked by one if* $m(d) = 1$ *and unmarked if* $m(d) = 0$*.*

The initial marking m_0 *of an EPC is defined as follows:*

(i) The postsets of all initial events are marked by one and
(ii) all other edges are unmarked.

The occurrence rule for nodes of EPCs is given by the following definition.

Definition 3. *Given a marked EPC $epc = (\mathcal{A}, \mathcal{E}, \mathcal{C}_{xor}, \mathcal{C}_{and}, \mathcal{D}, m)$, a node $n \in \mathcal{A} \cup \mathcal{E} \cup \mathcal{C}_{xor} \cup \mathcal{C}_{and}$ is executable if one of the following statements holds.*

(i) $| \bullet n| = 1$ and the unique edge $d \in \bullet n$ is marked or
(ii) $| \bullet n| > 1$, $n \in \mathcal{C}_{and}$ and each edge $d \in \bullet n$ is marked or
(iii) $| \bullet n| > 1$, $n \in \mathcal{C}_{xor}$ and at least one edge $d \in \bullet n$ is marked.

If a node is executable, it can be fired changing the marking of the EPC. Firing a node $n \notin \mathcal{C}_{xor}$ leads to the marking m' defined by:

$$m'(d) = \begin{cases} m(d) - 1, \, d \in \bullet n \\ m(d) + 1, \, d \in n \bullet \\ m(d) \qquad else \end{cases}$$

When firing a node $n \in \mathcal{C}_{xor}$, a marked edge $e_{in} \in \bullet n$ and an edge $e_{out} \in n \bullet$ have to be fixed. Then, firing n w.r.t. e_{in} and e_{out} leads to the marking m' defined by:

$$m'(d) = \begin{cases} m(d) - 1, \, d = e_{in} \\ m(d) + 1, \, d = e_{out} \\ m(d) \qquad else \end{cases}$$

Each different choice of e_{in} and e_{out} yields another marking m'.

If a node n is executable in a marking m and firing n leads to a marking m', we shortly write $m \overset{n}{\to} m'$.

Next, we further extend the occurrence rule to sets of nodes.

Definition 4. *A set of nodes $\mathcal{N}$ is concurrently executable in a marking m if each node $n \in \mathcal{N}$ is executable in m.*

In this case, a follower marking m' is given by firing the set of nodes in any order. If a set of nodes $\mathcal{N}$ is executable in a marking m and firing $\mathcal{N}$ leads to a marking m', we write $m \overset{\mathcal{N}}{\to} m'$.

A sequence of sets of nodes $\sigma = \mathcal{N}_1\mathcal{N}_2 \dots \mathcal{N}_n$ is executable in a marking m, if there are markings $m_1, m_2, \dots m_n$ such that $m \overset{\mathcal{N}_1}{\to} m_1 \overset{\mathcal{N}_2}{\to} \dots \overset{\mathcal{N}_n}{\to} m_n$.

Given an executable sequence of sets of nodes $\sigma = \mathcal{N}_1\mathcal{N}_2 \dots \mathcal{N}_n$ and its projection onto activities $\sigma_{\mathcal{A}}^{\emptyset} = \mathcal{N}_1 \cap \mathcal{A} \dots \mathcal{N}_n \cap \mathcal{A}$, the sequence of sets of activities which arises from $\sigma_{\mathcal{A}}^{\emptyset}$ when omitting all empty sets is called activity step sequence $\sigma_{\mathcal{A}}$ of σ.

A sequence of sets of activities σ is executable in a marking m if there exists an executable sequence of sets of nodes σ' such that $\sigma = \sigma'_{\mathcal{A}}$.

For instance, in the initial marking of the EPC in Figure 1 the sequence of sets of nodes $\{ST\}$, $\{Event\}$, $\{A\}$, $\{XOR\}$, $\{AND\}$, $\{Event, XOR\}$, $\{Event\}$, $\{B, D\}$, $\{Event\}$, $\{C, XOR\}$ is executable (for simplicity we omit names for connectors and events). Therefore, the corresponding activity step sequence $\{ST\}, \{A\}, \{B, D\}, \{C\}$ is executable. We use the following notions for partially ordered runs.

Definition 5. *Given a set of activities $\mathcal{A}$, a (finite) labeled partial order (LPO) is a triple $lpo = (V, <, l)$, where V is a finite set of vertices, $<$ is an irreflexive and transitive binary relation over V and $l : V \to \mathcal{A}$ is a labeling function. The Hasse diagram underlying an LPO $lpo = (V, <, l)$ is defined by the labeled directed graph $h_{lpo} = (V, <_h, l)$ where $<_h = \{(v, v') \mid v < v' \wedge \nexists v'' : v < v'' < v'\}$ is the set of skeleton edges.*

We here only consider LPOs without autoconcurrency i.e. $v, v' \in V, v \neq v', v \nless v', v' \nless v \Rightarrow l(v) \neq l(v')$.

Given an LPO $lpo = (V, <, l)$, an LPO $lpo' = (V, <', l)$ with $< \subseteq <'$ is called sequentialization of lpo.

Given an LPO $lpo = (V, <, l)$ and a sequentialization $lpo' = (V, <', l)$ of lpo with $V = V_1 \dot{\cup} \ldots \dot{\cup} V_n$ and $<' = \bigcup_{i<j} V_i \times V_j$, the sequence of sets of activities $l(V_1) \ldots l(V_n)$ is called step sequence of lpo.

In Figure 2, Hasse diagrams of three LPOs are shown. An example of a step sequence of the first LPO is $\{ST\}, \{A\}, \{B, D\}, \{C\}, \{F\}, \{FI\}$. Finally, we define the executability of an LPO.

Definition 6. *Given an EPC epc, an LPO lpo is executable w.r.t. epc if all step sequences of lpo are executable in the initial marking of epc.*

Note that, as in the case of Petri nets, we can also define the executability of an LPO w.r.t. an EPC by using concepts similar to occurrence nets and process nets of Petri nets.

3 Folding Algorithm

In this section, we explain a folding algorithm generating an EPC model $epc = (\mathcal{A}, \mathcal{E}, \mathcal{C}_{xor}, \mathcal{C}_{and}, \mathcal{D})$ of a business process from a set of LPOs L representing scenarios of the process. Each LPO of L models one possible run of the process, i.e. different LPOs of L represent alternative scenarios. A vertex of an LPO represents an activity occurrence where the label refers to the respective activity of the process. The edges of an LPO describe the precedence relation of the activity occurrences within the respective scenario. Unordered vertices represent concurrent activity occurrences. For formal purposes, we assume that each LPO includes a vertex labeled with ST (Start) which is ordered before all the other vertices and by a vertex labeled with FI (Final) which is ordered behind all the other vertices (such vertices can always be added to an LPO).

We now present the steps of the folding algorithm generating an EPC epc from a set of LPOs L. Directly after the following formal definition of the algorithm which is rather technical, we provide an illustrative example.

- The algorithm generates an EPC with only two events, a start and a final event, i.e. $\mathcal{E} = \{Start, Final\}$. The set of activities of the EPC is given by the labels of the LPOs, i.e. $\mathcal{A} = \{l(v) \mid v \in V, (V, <, l) \in L\}$.
- For each vertex $v \in V$, $(V, <, l) \in L$ we consider the sets of direct predecessor and successor activities of the vertex, called predecessor-set and successor-set of the vertex. The predecessor-set is defined as $pred(v) = \{l(v') \mid v' <_h v\}$, the successor-set is defined as $succ(v) = \{l(v') \mid v <_h v'\}$.

– Then, for each activity $a \in \mathcal{A}$ we consider all vertices labeled by this activity and collect the predecessor-sets of these vertices in one set, called pre-activity-set of the activity, and the successor-sets of these vertices in another set, called post-activity-set of the activity. The pre-activity-set is defined as $prea(a) = \{pred(v) \neq \emptyset \mid v \in V, (V, <, l) \in L, l(v) = a\}$, the post-activity-set is defined as $posta(a) = \{succ(v) \neq \emptyset \mid v \in V, (V, <, l) \in L, l(v) = a\}$.
– For each activity $a \in \mathcal{A}$ we define an EPC-structure $epc^a = (\{a\}, \emptyset, \mathcal{C}^a_{xor}, \mathcal{C}^a_{and}, \mathcal{D}^a)$ containing the activity a and connectors determined by the dependencies given by the pre-activity-set and post-activity-set of a. The EPC-structure epc^a is called building block of a.
  - $\mathcal{C}^a_{xor} = \{xor^a_{pre}, xor^a_{post}\} \cup \{xor^a_{pre,a'} \mid a' \in x, x \in prea(a)\} \cup \{xor^a_{post,a'} \mid a' \in x, x \in posta(a)\}$
  - $\mathcal{C}^a_{and} = \{and^a_{pre,x} \mid x \in prea(a)\} \cup \{and^a_{post,x} \mid x \in posta(a)\}$
  - $\mathcal{D}^a = \{(xor^a_{pre}, a), (a, xor^a_{post}\} \cup \{(and^a_{pre,x}, xor^a_{pre}) \mid x \in prea(a)\} \cup \{(xor^a_{post}, and^a_{post,x}) \mid x \in posta(a)\} \cup \{(xor^a_{pre,a'}, and^a_{pre,x}) \mid a' \in x, x \in prea(a)\} \cup \{(and^a_{post,x}, xor^a_{post,a'}) \mid a' \in x, x \in posta(a)\}$
– By connecting the appropriate interface-connectors ($xor^a_{post,a'}$ with $xor^{a'}_{pre,a}$) of the building blocks we get the EPC-structure $epc' = (\mathcal{A}, \emptyset, \mathcal{C}'_{xor}, \mathcal{C}'_{and}, \mathcal{D}')$ defined by $\mathcal{C}'_{xor} = \bigcup_{a \in \mathcal{A}} \mathcal{C}^a_{xor}$, $\mathcal{C}'_{and} = \bigcup_{a \in \mathcal{A}} \mathcal{C}^a_{and}$ and $\mathcal{D}' = \bigcup_{a \in \mathcal{A}} \mathcal{D}^a \cup \{(xor^a_{post,a'}, xor^{a'}_{pre,a}) \mid a, a' \in \mathcal{A}, xor^a_{post,a'} \in \mathcal{C}^a_{xor}, xor^{a'}_{pre,a} \in \mathcal{C}^{a'}_{xor}\}$.
– Finally, the EPC $epc = (\mathcal{A}, E, \mathcal{C}_{xor}, \mathcal{C}_{and}, \mathcal{D})$ results from removing all connectors c having exactly one incoming edge (n, c) and exactly one outgoing edge (c, n') from $\mathcal{C}'_{xor}$. The two edges (n, c) and (c, n') are also removed from $\mathcal{D}'$ and an edge (n, n') is added to $\mathcal{D}'$. Moreover, the connectors xor^{ST}_{pre} and xor^{FI}_{post} are removed from $\mathcal{C}'_{xor}$. Also their related edges (xor^{ST}_{pre}, ST) and (FI, xor^{FI}_{post}) are removed from $\mathcal{D}'$ and edges $(Start, ST)$ and $(FI, Final)$ are added to $\mathcal{D}'$.

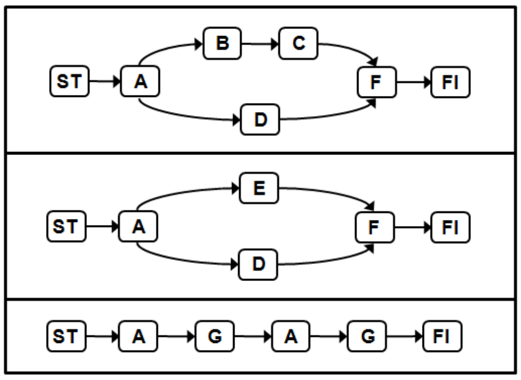


Fig. 2. Set of LPOs.

As an example, we consider the set of LPOs L illustrated in Figure 2 by their Hasse-diagrams. In this example, we have $\mathcal{A} = \{ST, A, B, C, D, E, F, G, FI\}$. To illustrate the notion of predecessor- and successor-sets consider the vertex v labeled by A in the first example LPO. There holds $pred(v) = \{ST\}$ and $succ(v) = \{B, D\}$. As the pre- and post-activity-set of the activity A we altogether get $prea(A) = \{\{ST\}, \{G\}\}$ and $posta(A) = \{\{B, D\}, \{E, D\}, \{G\}\}$. The construction of the building block for the activity A is shown in Figure 3.

- We first add an XOR-split-connector having an incoming edge coming from A and $|posta(A)|$ outgoing edges (Figure 3 (a)).
- Each outgoing edge of the XOR-split represents one set $x \in posta(A)$ (i.e. one possible set of successor-activities of A). Such edge is connected with a new AND-split connector having $|x|$ outgoing edges (Figure 3 (b)).
- Each outgoing edge of such AND-split represents a connection to one activity $a' \in x$. For each such a', the respective edges have to be merged to a unique interface w.r.t. a'. Therefore, these edges are connected with a new XOR-join-connector representing the unique interface. This connector has one outgoing edge for the connection with the activity a' (Figure 3 (c)).
- By proceeding analogously for $prea(A)$ (the role of incoming and outgoing edges as well as the role of splits and joins switches, see Figure 3 (d)-(f)) we get a building block for the activity A having a unique outgoing-interface to each activity $a \in x$, $x \in posta(A)$ and a unique incoming-interface to each activity $a \in x$, $x \in prea(A)$.

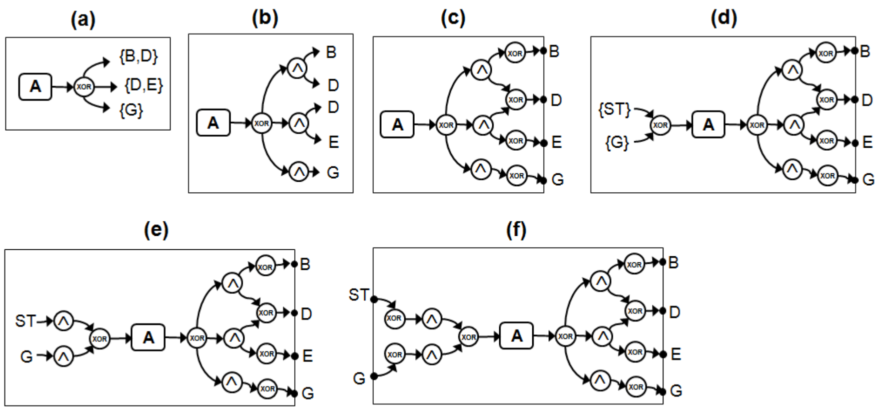


Fig. 3. Construction of a building block.

Figure 4 shows the building blocks for all the activities of our example, where connectors having exactly one incoming edge and exactly one outgoing edge and the connectors xor^{ST}_{pre} and xor^{FI}_{post} are already removed and replaced as defined by the last step of the algorithm. The interfaces of the building blocks exactly fit together, i.e. if the block of activity a has an outgoing-interface to a', then a' has an incoming-interface to

a. Therefore, the building blocks are connected along these interfaces yielding the final EPC shown in Figure 5.

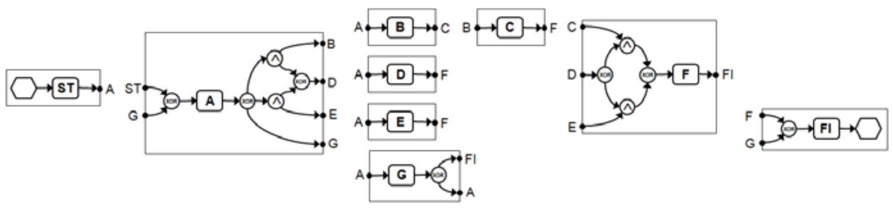


Fig. 4. Building blocks (inadequate connectors are already removed).

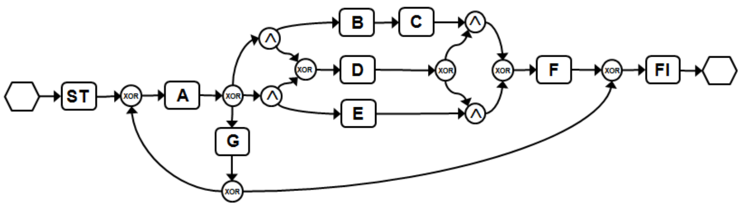


Fig. 5. Resulting EPC.

It remains to formally show that the presented folding algorithm in fact generates an EPC from a set of LPOs. For this purpose, we first prove that the interfaces of the building blocks in the step before last of the algorithm fit together.

Lemma 1. *In the previous algorithm, the interface connectors of the building blocks are matching, i.e. for two building blocks $epc^a, epc^{a'}$ there holds $xor^a_{post,a'} \in \mathcal{C}^a_{xor}$ if and only if $xor^{a'}_{pre,a} \in \mathcal{C}^{a'}_{xor}$.*

Proof. There holds $xor^a_{post,a'} \in \mathcal{C}^a_{xor}$ if and only if there exists $x \in posta(a)$ fulfilling $a' \in x$ if and only if there exists $v \in V$, $(V, <, l) \in L$, $l(v) = a$ fulfilling $a' \in succ(v)$ if and only if there exist $v, v' \in V$, $(V, <, l) \in L$, $v <_h v'$ fulfilling $l(v) = a, l(v') = a'$ if and only if there exists $v' \in V$, $(V, <, l) \in L$, $l(v') = a'$ fulfilling $a \in pred(v')$ if and only if there exists $x' \in prea(a')$ fulfilling $a \in x'$ if and only if there holds $xor^{a'}_{pre,a} \in \mathcal{C}^{a'}_{xor}$.

Lemma 2. *The EPC-structure $epc = (\mathcal{A}, \mathcal{E}, \mathcal{C}_{xor}, \mathcal{C}_{and}, \mathcal{D})$ generated by the previous algorithm is an EPC according to Definition 1.*

Proof. We have to check the requirements (i) - (iii).

(i): $Start$ and $Final$ are the only events of the EPC-structure. By construction, $Start$ has no incoming edge and only one outgoing edge connected with ST. $Final$ has no outgoing edge and only one incoming edge connected with FI.

(ii): By construction in epc' each activity $a \in \mathcal{A}$ has exactly one incoming edge connected with xor^a_{pre} and one outgoing edge connected with xor^a_{post}. In the last step of the algorithm, if such edge is removed, it is replaced by another edge, such that (ii) is preserved.

(iii): In the last step of the algorithm, property (iii) is ensured by removing from epc' the connectors xor^{ST}_{pre} and xor^{FI}_{post} and all connectors having exactly one incoming edge and exactly one outgoing edge. It remains to show that each connector of epc' except for xor^{ST}_{pre} and xor^{FI}_{post} either fulfills (iii) or has exactly one incoming edge and exactly one outgoing edge:

- xor^a_{pre} ($a \in \mathcal{A}$) has exactly one outgoing edge connected with a and it has an incoming edge connected with $and^a_{pre,x}$ for each $x \in prea(a)$ where $prea(a) \neq \emptyset$ for $a \neq ST$.
- xor^a_{post} ($a \in \mathcal{A}$) has exactly one incoming edge connected with a and it has an outgoing edge connected with $and^a_{post,x}$ for each $x \in posta(a)$ where $posta(a) \neq \emptyset$ for $a \neq FI$.
- $and^a_{pre,x}$ ($a \in \mathcal{A}$, $x \in prea(a)$) has exactly one outgoing edge connected with xor^a_{pre} and it has an incoming edge connected with $xor^a_{pre,a'}$ for each $a' \in x$ where $x \neq \emptyset$.
- $and^a_{post,x}$ ($a \in \mathcal{A}$, $x \in posta(a)$) has exactly one incoming edge connected with xor^a_{post} and it has an outgoing edge connected with $xor^a_{post,a'}$ for each $a' \in x$ where $x \neq \emptyset$.
- $xor^a_{pre,a'}$ ($a \in \mathcal{A}$, $x \in prea(a)$, $a' \in x$) by Lemma 2 has exactly one incoming edge connected with $xor^{a'}_{post,a}$ and it has an outgoing edge connected with $and^a_{pre,x'}$ for each $x' \in \{x \in prea(a) \mid a' \in x\}$ where $\{x \in prea(a) \mid a' \in x\} \neq \emptyset$.
- $xor^a_{post,a'}$ ($a \in \mathcal{A}$, $x \in posta(a)$, $a' \in x$) by Lemma 2 has exactly one outgoing edge connected with $xor^{a'}_{pre,a}$ and it has an incoming edge connected with $and^a_{post,x'}$ for each $x' \in \{x \in posta(a) \mid a' \in x\}$ where $\{x \in posta(a) \mid a' \in x\} \neq \emptyset$.

As it can be seen in the example, the EPC generated by the folding algorithm represents all the direct dependencies and respects all the independencies given by the specified LPOs. Therefore, it nicely represents the behavior given by the LPOs. In particular, it allows the execution of the specified LPOs according to Definition 6. We formally prove this interesting result in the following lemma. The result means that scenarios which are explicitly specified by a user are allowed by the generated EPC. This is an important and reasonable feature, e.g. in our example all three LPOs from Figure 2 are executable w.r.t. the constructed EPC from Figure 5. Nevertheless, many simplified algorithms for model construction, e.g. from the area of process mining [5], do not satisfy such property.

Lemma 3. *Each LPO $lpo \in L$ is executable w.r.t. the EPC epc generated by the previous folding algorithm (according to Definition 6).*

Proof. Given a step sequence $\sigma = \mathcal{A}_1 \dots \mathcal{A}_n$ of $lpo = (V, <, l)$ and the corresponding sequentialisation $lpo' = (V, <', l)$ of lpo, we consider the sequence σ' of sets of nodes of epc which is defined as follows:

- Given a set $\mathcal{A}_i = \{a_1 \dots a_m\}$ and its corresponding set of vertices $V_i = \{v_1 \dots v_m\} \subseteq V$ with $l(v_j) = a_j$, we construct a sequence σ_i of sets of nodes of *epc* having the form $\sigma_i = \sigma_i^{a_1,pre}\sigma_i^{a_2,pre} \dots \sigma_i^{a_m,pre}\mathcal{A}_i\sigma_i^{a_1,post}\sigma_i^{a_2,post} \dots \sigma_i^{a_m,post}$.
- For $pred(v_j) = \{a'_1 \dots a'_p\}$ we define $\sigma_i^{a_j,pre} = \{xor^{a_j}_{pre,a'_k} \mid 1 \leq k \leq p, xor^{a_j}_{pre,a'_k} \in \mathcal{C}_{xor}\}\{and^{a^j}_{pre,pred(v_j)} \mid and^{a^j}_{pre,pred(v_j)} \in \mathcal{C}_{and}\}\{xor^{a_j}_{pre} \mid xor^{a_j}_{pre} \in \mathcal{C}_{xor}\}$.
- For $succ(v_j) = \{a'_1 \dots a'_s\}$ we define $\sigma_i^{a_j,post} = \{xor^{a_j}_{post} \mid xor^{a_j}_{post} \in \mathcal{C}_{xor}\}$ $\{and^{a^j}_{post,succ(v_j)} \mid and^{a^j}_{post,succ(v_j)} \in \mathcal{C}_{and}\}\{xor^{a_j}_{post,a'_k} \mid 1 \leq k \leq s, xor^{a_j}_{post,a'_k} \in \mathcal{C}_{xor}\}$.
- Finally $\sigma' = \sigma_1 \dots \sigma_n$.

By construction $\sigma = \sigma'_{\mathcal{A}}$. To prove the executability of $\sigma = \mathcal{A}_1 \dots \mathcal{A}_n$, it remains to show that $\sigma' = \sigma_1 \dots \sigma_n$ is executable in the initial marking of *epc*. We sketch the proof for this statement in the following.

- First, we show that σ_1 is executable in the initial marking. We have $\mathcal{A}_1 = \{ST\}$ and $\sigma_1^{ST,pre} = \emptyset$. By construction ST is executable in the initial marking. Then, we have to check $\sigma_1^{ST,post}$. By construction, xor^{ST}_{post} is executable after ST. Then, if $|succ(v_j)| > 1$, we can fire $and^{ST}_{post,succ(v_j)}$. Afterwards, each xor^{ST}_{post,a'_k} fulfilling $a'_k \in succ(v_j)$ and $|\{x \in posta(ST) \mid a'_k \in x\}| > 1$ is executable.
- Now, we show that, if $\sigma_1 \dots \sigma_{i-1}$ is executable in the initial marking, then σ_i is executable in the follower marking. Given $\mathcal{A}_i = \{a_1 \dots a_m\}$ and $V_i = \{v_1 \dots v_m\}$ as before, we consider the executability of $\sigma_i^{a_j,pre}$. First, each $xor^{a_j}_{pre,a'_k}$ fulfilling $a'_k \in pred(v_j)$ and $|\{x \in prea(a_j) \mid a'_k \in x\}| > 1$ is executable. The reason is that we have fired the activity a'_k corresponding to the vertex $v <_h v_j$ with $l(v) = a'_k$ and, when they exist, the connectors $xor^{a'_k}_{post}$, $and^{a'_k}_{post,succ(v)}$ and $xor^{a'_k}_{post,a_j}$ within the sequence $\sigma_1 \dots \sigma_{i-1}$. Moreover, the process folder generated by $xor^{a'_k}_{post,a_j}$ is not consumed within the sequence $\sigma_1 \dots \sigma_{i-1}$ by our construction. Next, if $|pred(v_j)| > 1$, we can fire $and^{a_j}_{pre,pred(v_j)}$. Then, $xor^{a_j}_{pre}$ is executable. After each $\sigma_i^{a_j,pre}$, the set $\mathcal{A}_i$ is executable. Finally, the executability of $\sigma_i^{a_1,post}\sigma_i^{a_2,post} \dots \sigma_i^{a_m,post}$ can be shown analogously as in the last paragraph.

We have proven that each step sequence σ of *lpo* is executable in the initial marking of *epc* and thus *lpo* is executable.

The only problem of the folding algorithm is that indirect dependencies within the LPOs are not represented such that the generated EPC might also allow the execution of additional LPOs, i.e. additional behavior which has not been specified is possible. In our example from Figure 5, first the sequence $A \to G$ cannot only be repeated twice as specified by the third LPO, but it can be repeated arbitrarily often. Second, it is possible to finish the process after any number of repetitions of $A \to G$, in particular FI can be executed after just one execution of $A \to G$. Third, also the first two scenarios can still be executed after any number of repetitions of $A \to G$.[1] That means, due to the

[1] Moreover, to avoid a deadlock, the routing of the XOR-split behind D has to be chosen in accordance to the routing of the XOR-split behind A.

disregard of indirect dependencies the different specified behaviors can be combined in a certain way. However, this is not only a problem but can also be seen as a positive aspect. Typical specifications are incomplete. Thus, a real process often allows for more behavior than given by a specification. Abstracting from indirect dependencies, as it is done by the folding algorithm, results in a reasonable completion of the specified example behavior.

Finally, it remains to mention that the three stylistic requirements for EPCs mentioned in Remark 1 can easily be ensured by the folding algorithm. First of all, by construction the EPC generated by the algorithm contains no cycles of connectors only. Second, the generated EPC has only one starting and one final event. However, we can easily add (artificial) events in between the activities to guarantee alternating events and activities. For instance, we can add an event directly before each activity except of ST, i.e. events are inserted to the edges incoming to activities. In this way, not only alternation is ensured but the property that XOR-splits must not proceed events is also preserved. In our example, this approach yields the EPC shown in Figure 1 which now fulfills all the typical syntactic requirements for EPCs.

4 Conclusion

We have presented a folding algorithm to generate a process model in the form of an EPC from example scenarios given by a set of LPOs. The presented algorithm is very efficient, more precisely it runs in linear time. It generates an intuitive model by representing all the direct dependencies specified by the LPOs. We have formally proven that the generated EPC allows the execution of all the specified LPOs. Moreover, it usually overapproximates the specification, i.e. additional scenarios which are “similar” to the specified scenarios are possible which is reasonable due to incomplete specifications.

References

1. Desel, J.: From Human Knowledge to Process Models. In: UNISCON 2008, LNBIP 5, Springer (2008) 84–95
2. Bergenthum, R., Desel, J., Mauser, S., Lorenz, R.: Construction of Process Models from Example Runs. In: ToPNoC II, LNCS 5460, Springer (2009) 243–259
3. Scheer: IDS Scheer: ARIS Process Performance Manager. http://www.ids-scheer.com.
4. Dongen, B., Aalst, W.: Multi-Phase Process Mining: Aggregating Instance Graphs into EPC's and Petri Nets. In: 2nd Workshop on Applications of Petri Nets to Coordination, Workflow and Business Process Management, Petri Nets 2005, Miami (2005) 35–58
5. van der Aalst, W.M.P., Weijters, T., Maruster, L.: Workflow Mining: Discovering Process Models from Event Logs. IEEE Trans. Knowl. Data Eng. **16**(9) (2004) 1128–1142
6. Smith, E.: Zur Bedeutung der Concurrency-Theorie für den Aufbau hochverteilter Systeme. PhD thesis, Universität Hamburg (1989)
7. van der Aalst, W.M.P.: Formalization and Verification of Event-driven Process Chains. Information & Software Technology **41**(10) (1999) 639–650
8. Kindler, E.: On the Semantics of EPCs: A Framework for Resolving the Vicious Circle. In: BPM 2004, LNCS 3080, Springer (2004) 82–97
9. Kiehn, A.: On the Interrelation Between Synchronized and Non-Synchronized Behaviour of Petri Nets. Elektronische Informationsverarbeitung und Kybernetik **24**(1/2) (1988) 3–18

Towards Distributed Control of Discrete-Event Systems[⋆]

Philippe Darondeau[1] and S.L. Ricker[2]

[1] INRIA Rennes-Bretagne Atlantique, Campus de Beaulieu, Rennes, France
darondeau@irisa.fr
[2] Mathematics & Computer Science, Mount Allison University, Sackville NB, Canada
lricker@mta.ca

Abstract. To initiate a discussion on the modeling requirements for distributed control of discrete-event systems, a partially-automated region-based methodology is presented. The methodology is illustrated via a well-known example from distributed computing: the dining philosophers.

1 Decentralized, Asynchronous, and Distributed DES Control

In this section, we explain our understanding of the gradual evolution of control for discrete-event systems (DES) from centralized control to more advanced forms, including distributed control, an area still in its infancy but one that may soon become vital for practical applications of the theory.

Ramadge and Wonham's theory of non-blocking supervisory control for DES was introduced in [23] and developed soon after in [18, 24] to cover situations in which only some of the events generated by the DES are observed, for instance "in situations involving decentralized control" [24]. The basic supervisory control problem consists of deciding, for a DES with observable/unobservable and controllable/uncontrollable events, whether a specified behavior may be enforced on the DES by some admissible control law. This basic supervisory control problem is decidable. Moreover, if every controllable event is also observable, then one can effectively compute the largest under-approximation of the specified behavior that can be enforced by the supervisory control.

After the work done in [28] and extended in [35], *decentralized control* refers to a form of control in which several local supervisors, with different subsets of observable events and controllable events, cooperate in rounds. In each round, the local supervisors compute a set of authorized events and the DES performs authorized events until one is observed by some local supervisor, which puts an end to the

⋆ Work partially supported by the EC-FP7 under project DISC (Grant Agreement INFSO-ICT-224498) and NSERC.

round. In each round, the cooperation between the supervisors may result from an implicit synchronization or arbitration, e.g., in conjunctive coordination, or it may be obtained by applying distributed algorithms, e.g., to reach a consensus on the set of authorized events. In any case, the controlled DES is governed by a global clock, whose ticks are the events whose occurrences start the rounds. This global clock may not be a problem for embedded system controllers, especially when they are designed using synchronous programming languages like Lustre [12]. Yet global clocks are impractical for widely distributed systems or architectures, not to mention for distributed workflows or web services. For this reason Globally Asynchronous Locally Synchronous (GALS) architectures have been developed [22]. The existence of non-blocking decentralized supervisory control for the basic supervisory control problem is decidable in the special case where the closed behavior of a DES is equal to the closure of its marked behavior, but it is undecidable in the general case without communication [16, 30].

Decentralized control can be extended to the case where supervisors communicate [3]. When the correct control decision cannot be taken by any of the local supervisors at the end of one of the cooperative rounds of decision making, communication between supervisors can be introduced. The basic idea of this class of problems is to ensure that supervisors receive relevant information about system behavior that they cannot directly observe so that (at least) one local supervisor can make the correct control decision. Further, it is often desirable to synthesize an optimal communication protocol, either from a set-theoretic (e.g., [27]) or quantitative (e.g., [26]) perspective. Thus, supervisors make control decisions based on not only their partial observations, but also on information received from other supervisors.

For the most part, work in this area has been devoted to the study of synchronous communication protocols. There are several different approaches: build a protocol using a bottom-up approach [3, 25]; given a correct communication protocol, use a top-down approach to reduce it to one that is optimal from a set-theoretic perspective [27, 32]; and distribute an optimal centralized solution among a set of communicating supervisors [19]. Bounded and unbounded communication delays have also been explored [31, 13]. The communication channel is modeled as a FIFO buffer; however, there is no notion of optimality since the communication protocol for each supervisor is to simply send all of its observations to the other supervisors. In the case of unbounded delay, the problem is undecidable [31, 13].

Decentralized control with communication extends the basic decentralized control problem in the general direction of distributed control; however, the synthesis of asynchronous communication protocols remains largely unexamined. Of some interest is the class of models where asynchronous interaction does not require the intervention of an *arbitrator* [17]. At present, though, none of the strategies for synthesizing communication protocols in decentralized control of DES meets the conditions of asynchronous communication required for distributed control.

Asynchronous control was investigated in [21] in the framework of recognizable trace languages and asynchronous automata [37, 20, 8]. In an asynchronous automaton, states are vectors of local states, and transitions are defined on *partial* states, i.e., projections of states on a subset of locations. Transitions that concern disjoint subsets of locations produce independent events, hence asynchronous automata have *partially ordered runs* (*ergo* there is no global clock). In [21], a DES is an asynchronous automaton where an uncontrollable event has exactly one location while a controllable event may have several locations, and every event is observed in all locations concerned in the generating transition. The goal is to construct for all locations *local supervisors* that jointly enforce a specified behavior on the DES, i.e., a specific subset of partially ordered runs. Local supervisors cooperate in two ways. First, a controllable event with multiple locations cannot occur in the controlled DES unless it is enabled by all supervisors in these locations. Second, at each occurrence of a controllable event, all local supervisors concerned with this event synchronize and communicate to one another all information that they have obtained so far by direct observation of local events and from prior communication with other supervisors. The information available to the supervisor in location ℓ is therefore the ℓ-view of the partially ordered run of the DES, i.e., the causal past of the last event with location ℓ. The local control in location ℓ is a map from the ℓ-view of runs to subsets of events with location ℓ, including all uncontrollable events that are observable at this location. Implementing asynchronous supervisors, as they are defined above, on distributed architectures requires implementing first a protocol for synchronizing the local supervisors responsible for an event whenever the firing of this event is envisaged. One of the main contributions of [21] is the identification of conditions under which one can effectively decide upon the existence of asynchronous supervisors, which, moreover, can always be translated to *finite* asynchronous automata.

A different form of asynchronous control has been examined in [5]. Here the goal is not to compute from scratch an asynchronous supervisor but, rather, to transform a centralized (optimal and non-blocking) supervisor into an equivalent system of local supervisors with *disjoint* subsets of controllable events. The data for the problem are a DES G and a centralized supervisor S, defined over a set Σ of observable/unobservable and controllable/uncontrollable events, plus a partition $\Sigma = \biguplus_{\ell \in \mathcal{L}} \Sigma_\ell$ of Σ over a finite set of locations. All local supervisors have the set of events Σ, but the local supervisor S_ℓ in location ℓ is the sole supervisor that can disable controllable events at location ℓ. The states of S_ℓ are the cells of a corresponding *control cover*, i.e., a covering of the set of states of S by *cells* with the following properties. First, if two transitions from a cell are labelled by the same event, then they must lead to the same cell. Second, if two states x and x' are in the same cell, then for any reachable states (x, q) and (x', q') of $S \times G$ and for any event e with location ℓ, if e is enabled at q and q' in G, then e should be coherently enabled or disabled at x and x' in S. Third, two states in a cell should be consistent w.r.t. marking information. Supervisor localization based on control covers is universal in the sense that it always succeeds in transforming a centralized supervisor S into an equivalent family of local supervisor S_ℓ. It is worth noting that a local supervisor S_ℓ with language $L(S_\ell)$ may be replaced equivalently by a local supervisor with the language $\pi(L(S_\ell))$ for any natural projection operator $\pi : \Sigma^* \to (\Sigma'_\ell)^*$ such that $\Sigma_\ell \subseteq \Sigma'_\ell \subseteq \Sigma$ and $L(S_\ell) = \pi^{-1}\pi(L(S_\ell))$. In the end, for any event $e \in \Sigma$, whenever e occurs in the controlled DES, all local supervisors S_ℓ such that $e \in \Sigma'_\ell$ should perform local transitions labelled by e *in a synchronized way.*

The approaches taken in [21] and [5] differ significantly. The former approach is based on asynchronous automata and on the assumption that several local supervisors may be responsible for the same controllable event. The latter approach is based on product automata and on the assumption that exactly one local supervisor is responsible for each controllable event. Both approaches, though, rely on similar requirements for the final implementation of the asynchronous supervisors. In both approaches, the only way for the local supervisors to communicate with one another is to synchronize on shared events. As a matter of fact, widely distributed architectures do not generally provide protocols for performing synchronized transitions in different locations.

By *distributed control*, we mean a situation in which the local supervisors and the DES cooperate as follows. First, the set of observable or controllable events of the DES is partitioned over the different locations ($\Sigma = \biguplus_{\ell \in \mathcal{L}} \Sigma_\ell$), and for each location $\ell \in \mathcal{L}$, every event in Σ_ℓ results from the synchronized firing of two e-labelled transitions, in the DES and in the local supervisor S_ℓ, respectively. Second, the set of events of each local supervisor S_ℓ is the union of Σ_ℓ and a set X_ℓ of auxiliary *send* and *receive* events, used to communicate with the other local supervisors in *asynchronous message passing mode.* Messages sent from one location to another are never lost; however, they may overtake one another and the arrival times cannot be predicted. Distributed controllers of this type, already suggested in [6], can be implemented almost directly on *any* distributed architecture. A slightly different type of distributed controllers, in which the local supervisors communicate by FIFO buffers, has recently been proposed in [15] together with synthesis algorithms for avoiding forbidden states in infinite systems. Prior work along the same line was presented in [34], where the goal was distributed state estimation in finite DES instead of distributed control.

In this paper, we do not intend to present a new theory, but, rather, we use a well-known example with which to illustrate the concept of *distributed control*, an area that is not yet fully understood, with the hope of motivating further research in this direction. The example chosen is the $n = 3$ version of the classical n dining philosophers problem, where the local supervisors are in the forks and both hands of each philosopher may act concurrently. The techniques that we use for constructing distributed controllers for this problem are drawn or adapted from [6], and they are briefly recalled in the next section before they are put into practice.

2 Distributed Controller Synthesis Based on Petri Net Synthesis

In this section, we describe the background of Petri net based controller synthesis and its extension to distributed controller synthesis.

Given a DES G over a set of events Σ, called the *plant*, let *Spec* be a specification of the desired behavior of G. Let $\Sigma = \Sigma_o \uplus \Sigma_{uo} = \Sigma_c \uplus \Sigma_{uc}$, where Σ_o and Σ_c are the sets of observable and controllable events, respectively. Henceforth, we assume that every controllable event is observable, i.e., $\Sigma_c \subseteq \Sigma_o$. Then the supervisory control problem consists of constructing a supervisor S over the set of events Σ_o such that the partially synchronized product of S and G, usually denoted by $S \times G$ but here denoted by (S/G), satisfies *Spec* and the following *admissibility* condition holds: for every reachable state (x, q) of (S/G), if an uncontrollable event $e \in \Sigma_{uc}$ is enabled at state q in G, then the event e should also be enabled at state x in S.

Let $S = (X, \Sigma_o, \delta, x_o)$ be a deterministic supervisor satisfying the above requirements, where X is the set of states, $x_0 \in X$ is the initial state, and $\delta : X \times \Sigma_o \to X$ is the partial transition map. For simplicity, we do not consider marked states here. We say that S is *Petri net definable* (PND) if it is isomorphic to the reachability graph of a Petri net system N with the set of transitions $T = \Sigma_o$. Let us recall some definitions.

Definition 1. *A* Petri net *is a bi-partite graph (P, T, F), where P and T are* finite *disjoint sets of vertices, called* places *and* transitions*, respectively, and $F : (P \times T) \cup (T \times P) \to \mathbb{N}$ is a set of directed edges with non-negative integer weights. A* marking *of N is a map $M : P \to \mathbb{N}$. A transition $t \in T$ is* enabled at *a marking M (denoted by $M[t\rangle$) if $M(p) \geq F(p, t)$ for all places $p \in P$. It t is enabled at M, then it can* be fired*, leading to the new marking M' (denoted by $M[t\rangle M'$) defined by $M'(p) = M(p) + F(t, p) - F(p, t)$ for all $p \in P$.*

Definition 2. *A* Petri net system *N is a tuple (P, T, F, M_0) where M_0 is a marking of the underlying net (P, T, F), called the initial marking. The* reachability set *$RS(N)$ of N is the set of all markings reached by sequences of transitions from M_0. The* reachability graph *$RG(N)$ of N is the transition system $(RS(N), T, \delta, M_0)$ with the partial transition map $\delta : RS(N) \times T \to RS(N)$ defined as $\delta(M, t) = M'$ iff $M[t\rangle M'$ for all markings M and M' in $RS(N)$.*

Thus, S is PND if there exists a Petri net system $N = (P, T, F, M_0)$ with the set of transitions $T = \Sigma_o$ and a bijection $\varphi : X \to RS(N)$ such that $\varphi(x_0) = M_0$ and for all $x \in X$ and $t \in \Sigma_o$, $\delta(x, t)$ is defined if and only if $M[t\rangle M'$ for $M = \varphi(x)$ and for some marking M', and then $M' = \varphi(x')$ where $x' = \delta(x, t)$.

PND controllers comprised of *monitor places* for PND plants date back to the work in [7, 36]. Monitor places are linear combinations of places of the net which define the plant, and they are added to the existing places of this net in order to constrain its behavior. PND controllers for DES given as labelled transition systems proceed from a similar idea with one significant difference: since G has no places, the places of the controller net N are synthesized directly from $Spec$ (the specification) using the theory of regions [1]. This controller net synthesis technique was inaugurated in [10]. In that work, the specification $Spec$ was the induced restriction of G on a subset of "good" states. The theory of regions may be applied both to state-oriented specifications given by transition systems and to event-oriented specifications given by languages. It is shown in [6] how the theory can be applied to the supervisory control problem with tolerance stated as follows:

Given a plant G and two prefix-closed regular languages L_{min} and L_{max}, such that $L_{min} \subseteq L_{max} \subseteq L(G)$, where $L(G)$ denotes the language generated by G, decide whether there exists and construct a Petri net controller N such that

$$L_{min} \subseteq L((RG(N)/G)) \subseteq L_{max}$$

After attempts to apply the theory to significant examples found in the literature (e.g., workflows, train systems), we convinced ourselves that in most cases where supervisory control is needed, the main objective is to avoid deadlocks or to enforce home states or both, but it is simply impossible to express such objectives by tolerance specifications because L_{min} is not known beforehand. The example developed in the next section also illustrates this situation.

At this stage, it may seem odd to search for PND controllers instead of general controllers defined by labelled transition systems. Indeed, the Petri nets that we consider are labelled injectively (their set of transitions T is equal to the set Σ_o of the observable events), and it is well-known that languages of injectively labelled and bounded Petri nets form a *strict subset* of the regular languages. So, while one loses generality, one gains a more compact representation of controllers, which appears to be poor compensation. Fortunately, things change in a radical way when one takes distributed control into account, since one can tailor controller synthesis to distributed Petri nets that can be converted to asynchronously communicating automata, as we explain below.

To begin with, we recall from [2] the definition of distributed Petri nets and their automated translation into asynchronous message passing automata.

Definition 3 (Distributed Petri net system). *A* distributed *Petri net system over a set of locations $\mathcal{L}$ is a tuple $N = (P, T, F, M_0, \lambda)$ where (P, T, F, M_0) is a Petri net system and $\lambda : T \to \mathcal{L}$ is a map, called a* location map, *subject to the following constraint: for all transitions $t_1, t_2 \in T$ and for every place $p \in P$, $F(p, t_1) \neq 0 \wedge F(p, t_2) \neq 0 \Rightarrow \lambda(t_1) = \lambda(t_2)$.*

In a distributed Petri net, two transitions with different locations cannot compete for tokens, hence *distributed conflicts cannot occur*, which makes the effective implementation easy. Two transitions with different locations may, though, *send* tokens to the same place. Implementing a distributed Petri net system N means producing an asynchronous message passing automaton ($AMPA$) behaving like N up to branching bisimulation (see Appendix for precise definitions). Given any non-negative integer bound B, let $RG_B(N)$ denote the induced restriction of the reachability graph $RG(N)$ of N on the subset of markings bounded by B, i.e., markings M such that $M(p) \leq B$ for all places $p \in P$. Transforming N into a B-bounded $AMPA$ with a reachability graph branching bisimilar to $RG_B(N)$ may be done as follows (see [2] for a justification).

– Given $N = (P, T, F, M_0, \lambda)$, extend $\lambda : T \to \mathcal{L}$ to $\lambda : (T \cup P) \to \mathcal{L}$ such that $\lambda(p) \neq \lambda(t) \Rightarrow F(p,t) = 0$ for all $p \in P$ and $t \in T$ (this is always possible by Def. 3).
– For each location $\ell \in \mathcal{L}$, construct a net system $N_\ell = (P_\ell, T_\ell, F_\ell, M_{\ell,0})$, called a *local net*, as follows.
  • $P_\ell = \{p \mid p \in P \wedge \ell = \lambda(p)\} \cup \{(p,\ell) \mid p \in P \wedge \ell \neq \lambda(p)\}$,
  • $T_\ell = \{t \in T \mid \lambda(t) = \ell\} \cup$
  $\{\ell\,!\,p \mid p \in P \wedge \ell \neq \lambda(p)\} \cup \{\ell\,?\,p \mid p \in P \wedge \ell = \lambda(p)\}$,
  • $F_\ell(p,t) = F(p,t)$ and $F_\ell(t,p) = F(t,p)$,
  • $F_\ell(t,(p,\ell)) = F(t,p)$,
  • $F_\ell((p,\ell),\ell\,!\,p) = 1$ and $F_\ell(\ell\,?\,p,p) = 1$,
  • $M_{\ell,0}(p) = M_0(p)$.

  In each local net N_ℓ, places (p,ℓ) are local clones of places p of other local nets. Whenever a transition $t \in T$ with location ℓ produces tokens for a *distant* place $p \in P$ ($\lambda(t) = \ell \neq \lambda(p)$), the transition $t \in T \cap T_\ell$ produces tokens in the local clone (p,ℓ) of p. These tokens are removed from the local clone (p,ℓ) of p by the auxiliary transition $\ell\,!\,p$, each firing of which models an asynchronous emission of the *message* p. Symmetrically, for any place p of N with location ℓ, each firing of the auxiliary transition $\ell\,?\,p$ models an asynchronous reception of the *message* p, resulting in one token put in the corresponding place $p \in P_\ell$.
– For each location ℓ, compute $RG_B(N_\ell)$. The desired $AMPA$ is the collection of local automata $\{A_\ell = RG_B(N_\ell) \mid \ell \in \mathcal{L}\}$. Each transition of an automaton A_ℓ is labelled either with some $t \in T \cap T_\ell$ or with some asynchronous message emission $\ell\,!\,p$ or with some asynchronous message reception $\ell\,?\,p$. A message p sent from a location ℓ' by a transition $\ell'\,!\,p$ of the automaton $A_{\ell'}$ is automatically routed towards the automaton A_ℓ with the location $\ell = \lambda(p)$, where it is received by a transition $\ell\,?\,p$ of the automaton A_ℓ. No assumption is made on the relative speed of messages, nor on the order in which they are received.

Closing the parenthesis, let us return to controllers. Let G be a DES over a set of events $\Sigma = \Sigma_o \uplus \Sigma_{uo} = \Sigma_c \uplus \Sigma_{uc}$ where $\Sigma_c \subseteq \Sigma_o$. Let *Spec* be a specification of the desired behavior of G. Let $\mathcal{L}$ be a finite set of locations (or sites). Let $\lambda : \Sigma_o \to$

$\mathcal{L}$ be a location map, specifying for each observable event e the location $\lambda(e)$ in which it may be observed and possibly controlled. Let $S = (X, \Sigma_o, \delta, x_o)$ be a finite-state admissible supervisor for G, such that (S/G) satisfies the specification *Spec*. Thus, for every reachable state (x, q) of (S/G), if $e \in \Sigma_{uc}$ is enabled in state q in G, then e is also enabled in state x in S. We say that S is a *distributed Petri net definable* supervisor (DPND) if it is isomorphic to the reachability graph of a distributed Petri net $N = (P, T, F, M_0, \lambda)$ with a set of transitions $T = \Sigma_o$. In view of this isomorphism, since X is a finite set of states, there must exist a finite bound B such that $M(p) \leq B$ for all places $p \in P$ and for all reachable markings M of N. Therefore, $S = (X, \Sigma_o, \delta, x_o)$ may be translated equivalently to an $AMPA = \{A_\ell = RG_B(N_\ell) \mid \ell \in \mathcal{L}\}$, realizing, in the end, *fully distributed control.*

Therefore, the approach to distributed control that we suggest is to search for admissible DPND controllers using Petri net synthesis techniques (see [1] for a survey). With the adaptation proposed in [2], these techniques allow us to answer the following types of questions:

– Given a finite automaton over Σ_o and a location map $\lambda : \Sigma_o \to \mathcal{L}$, can this automaton be realized as the reachability graph of a distributable Petri net $N = (P, T, F, M_0, \lambda)$ with set of transitions $T = \Sigma_o$?
– Given a regular language over Σ_o and a location map $\lambda : \Sigma_o \to \mathcal{L}$, can this language be realized as the set of firing sequences of a distributable Petri net $N = (P, T, F, M_0, \lambda)$ with set of transitions $T = \Sigma_o$?

The type of net synthesis techniques that should be applied when solving the distributed supervisory control problem depends upon the specification *Spec* of the control objective. A method to derive distributed supervisors for the basic supervisory control problem with tolerances was described in [6]. This method cannot be applied when the control objective is avoiding deadlocks or enforcing home states. In fact, we do not know of any systematic method to cope with such problems. The case study presented in next section aims at motivating work in this direction. To give a flavour of the approach, let us set ourselves in the simple case where G is given by a Petri net. One proceeds by a series of trial and error. At each step, one removes states from the reachability graph of G, and checks that it is enough to add distributed monitor places to confine the behavior to the remaining states. The distribution constraints on monitor places are represented in the synthesis procedure by sign constraints depending upon the chosen location of the monitor place.

Note that AMPA derived from distributed Petri nets are a *strict subclass* of AMPA. Thus, the distributed controller synthesis techniques that we propose may fail even though the distributed supervisory control problem may be solved using more general AMPA. This point illustrates, if it was not yet totally clear, that the field for research on distributed controller synthesis is wide open.

3 Distributed Controllers for Three Dining Philosophers

We would now like to experiment with our Petri net based synthesis method for distributed controllers on a toy example. We have chosen the famous problem of the dining philosophers [14]. The reasons for this choice are twofold. On the one hand, the problem is easily understood. On the other hand, the size of this problem can be adjusted to accommodate our partially-automated processing techniques by decreasing the usual number of philosophers, which is five, to three philosophers.

Let us recall the statement of the problem. Three philosophers φ_1, φ_2, and φ_3 are sitting at a table with a bowl of spaghetti in the center. Three forks f_1, f_2, f_3 are placed on the table, such that philosopher ϕ_i has the fork f_i on his right and the fork $f_{i+1 \mod 3}$ on his left (see Fig. 1(a)). A philosopher alternates periods of eating and periods of thinking. To eat, he needs both the fork to his direct left and the fork to his direct right. Therefore, he tries to grab them one after the other while thinking. A philosopher who thinks with a fork in each hand stops thinking and starts eating after a finite delay. A philosopher who eats eventually stops eating, puts down the forks, and starts thinking again after a finite delay. The basic problem is to avoid deadlock, i.e., the situation in which every philosopher has taken one fork. A classical solution is to let all philosophers but one, say φ_1, take first the fork to their right, and to let φ_1 take first the fork to his left. An augmented problem is to avoid the starvation of any philosopher.

The dining philosophers problem was used in [33] to illustrate the use of supervisory control techniques for removing deadlocks from multi-threaded programs with lock acquisition and release primitives. In that work, the emphasis was on optimal control, not on distribution. In this paper, we do the opposite, i.e., we give priority to distributed control over optimal control. Moreover, *we intend that local controllers will be embedded in forks*, not in philosopher threads. The idea is that resource managers have fixed locations while processes using resources are mobile. To make things precise, consider the PND plant G defined by the Petri net N (shown in Fig. 1(b)).

Places $f1f$, $f2f$, and $f3f$ are the resting places of the forks $f1$, $f2$, and $f3$, respectively (fif should be read as "f_i is free"), and they are initially marked with one token each. For $j \in \{1, 2, 3\}$ and $i \in \{j, j+1 \mod 3\}$, "philosopher φ_j can take fork f_i when it is free" (transition tji). After this, "philosopher φ_j holds fork f_i" (condition jhi). A philosopher φ_j with two forks may "start eating" (transition sej). A philosopher φ_j who is eating (condition ej) can "start thinking" (transition stj), and then forks f_j and $f_{j+1 \mod 3}$ return again to the table. With respect to distribution, there are four locations as follows. For each fork f_i, $i \in \{1, 2, 3\}$, the two transitions which compete for this fork, namely tii and tji with $j = i - 1 \mod 3$, are located on a site i specific to this fork. All other transitions are located on a default site 4 that does not matter.

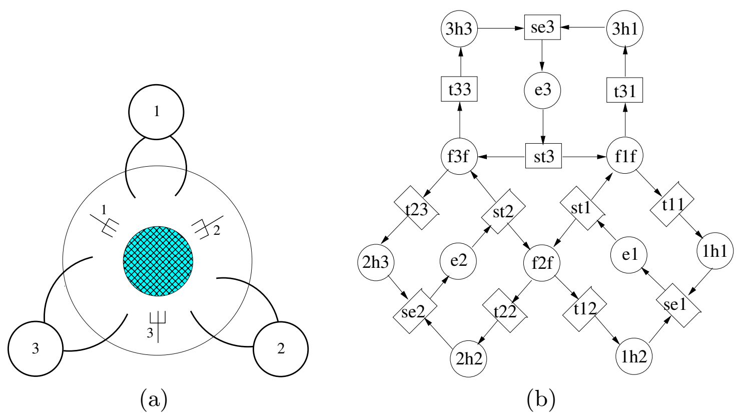


Fig. 1. Modeling the Dining Philosophers Problem:(a) three philosophers; (b) the uncontrolled plant

The control objective is to avoid deadlocks. For simplicity, we assume that all transitions are observable. The controllable transitions are the transitions which consume resources, i.e., the transitions tji (philosopher φ_j takes fork f_i). The control objective should be achieved by distributed control and, more precisely, by a distributed Petri net. The set of transitions T of the controller net may be any set included in $\{sej, stj, tjj, tji \mid 1 \leq j \leq 3 \wedge i = j+1 \mod 3\}$. The location map $\lambda : T \to \{1, 2, 3, 4\}$ is naturally the induced restriction of the map defined by $\lambda(tij) = j$ and $\lambda(sej) = \lambda(stj) = 4$ for $1 \leq j \leq 3$.

The PND plant G under consideration, i.e., the reachability graph of N, has two sink states M_1 and M_2, defined by $M_1(3h1) = M_1(1h2) = M_1(2h3) = 1$ and $M_2(1h1) = M_2(2h2) = M_2(3h3) = 1$, respectively (letting $M_1(p) = 0$ or $M_2(p) = 0$ for all other places p). If one disregards *distributed* control, it is quite easy to eliminate these two deadlocks by adding two monitor places p_1 and p_2, the role of which is to disable the instances of the transitions $t31, t12, t23$ and $t11, t22, t33$ that reach M_1 and M_2 in one step, respectively. The monitor places may be chosen so that no other transition instance is disabled, hence they do not cause new deadlocks. The PND controller consisting of the monitor places p_1 and p_2 realizes the optimal control of G where the objective is deadlock avoidance. mbox
This solution is straightforward because N is one-safe, i.e., for every reachable marking M and for every place p of N, $M(p) \in \{0, 1\}$. For any one-safe net with set of places P, the set of reachable markings is a convex subset of $\{0, 1\}^P$. As a consequence, a reachable marking M_x may always be separated from all other reachable markings by a hyperplane. Any separating hyperplane induces a monitor place that disables all (instances of) transitions crossing this hyperplane in a fixed direction. Therefore, for one-safe nets, optimal control can always be realized by PND controllers [9]. The two monitor places computed by SYNET

[29] are $p_1 = 2 + st1 + st2 + st3 - t31 - t12 - t23$ (i.e., p_1 is initially marked with 2 tokens, there are flow arcs with weight 1 from $st1$, $st2$ and $st3$ to p_1, and that there are flow arcs with weight 1 from p_1 to $t31$, $t12$ and $t23$), and $p_2 = 2 + st1 + st2 + st3 - t11 - t22 - t33$. This optimal controller is unfortunately not a distributed controller.

3.1 A distributed controller avoiding deadlocks

The reachability graph G of N has 36 states and 78 transitions. An inspection of G reveals the two subgraphs shown in Fig. 2, where the initial state is numbered 0 and the deadlock states M_1 and M_2 are numbered 25 and 11, respectively. All transitions that lead to a deadlock state in one step are, in fact, represented in Fig. 2. Note that all squares in the figure are *distributed diamonds*, i.e., the north/south edges and the west/east edges of a square are always labelled by two events which belong to different locations.

To avoid reaching the deadlock states M_1 and M_2, one can proceed as follows. First, all seven instances of the transitions $t31$ and $t22$ indicated in Fig. 2 are removed (manually) from the reachability graph G of N. Let G_1 be the reachable restriction of the resulting subgraph of G. G_1 has 23 states, none of which is a sink state.

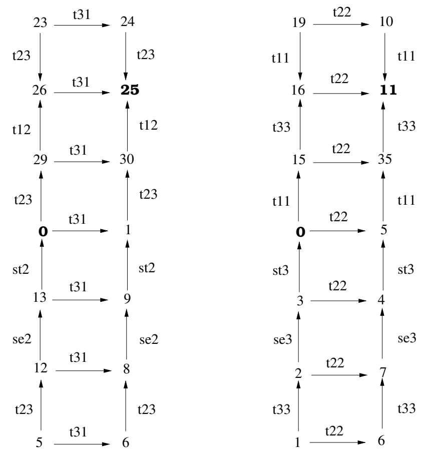
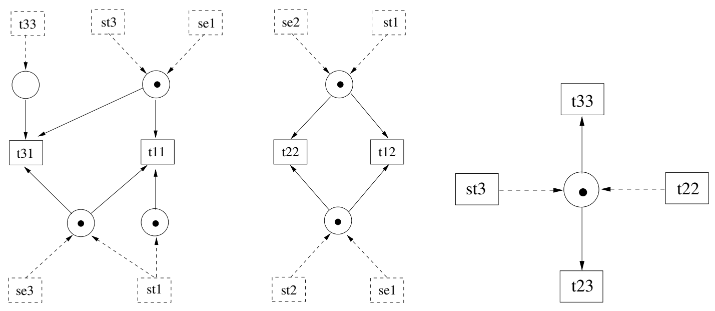

$$\begin{tikzcd}
23 \arrow[r, "t31"] \arrow[d, "t23"'] & 24 \arrow[d, "t23"] & & 19 \arrow[r, "t22"] \arrow[d, "t11"'] & 10 \arrow[d, "t11"] \\
26 \arrow[r, "t31"] & \mathbf{25} & & 16 \arrow[r, "t22"] & \mathbf{11} \\
29 \arrow[r, "t31"] \arrow[u, "t12"] & 30 \arrow[u, "t12"'] & & 15 \arrow[r, "t22"] \arrow[u, "t33"] & 35 \arrow[u, "t33"'] \\
\mathbf{0} \arrow[r, "t31"] \arrow[u, "t23"] & 1 \arrow[u, "t23"'] & & \mathbf{0} \arrow[r, "t22"] \arrow[u, "t11"] & 5 \arrow[u, "t11"'] \\
13 \arrow[r, "t31"] \arrow[u, "st2"] & 9 \arrow[u, "st2"'] & & 3 \arrow[r, "t22"] \arrow[u, "st3"] & 4 \arrow[u, "st3"'] \\
12 \arrow[r, "t31"] \arrow[u, "se2"] & 8 \arrow[u, "se2"'] & & 2 \arrow[r, "t22"] \arrow[u, "se3"] & 7 \arrow[u, "se3"'] \\
5 \arrow[r, "t31"] \arrow[u, "t23"] & 6 \arrow[u, "t23"'] & & 1 \arrow[r, "t22"] \arrow[u, "t33"] & 6 \arrow[u, "t33"']
\end{tikzcd}$$

Fig. 2. Two ladder-shaped subgraphs leading to deadlock

If this problem is feasible, the software SYNET will compute a distributed controller net K_1 such that G_1 is isomorphic to $RG(K_1)/G$. This is the case for our example, and SYNET produces a DPND controller with components K_1^1, K_1^2, K_1^3, K_1^4 to plug in the respective locations 1 to 4. K_1^4 does nothing but send the other components asynchronous signals informing them of the occurrences of the *start eating* and *start thinking* events sej and stj. The local controllers K_1^1, K_1^2, K_1^3 are depicted in Fig. 3. For each controller, the asynchronous flow from other local controllers is indicated by dashed arrows.



(a) K_1^1 (b) K_1^2 (c) K_1^3

Fig. 3. Local controllers for our example

Note that K_1^2 sends messages to K_1^3 (indicating that $t22$ has occurred) and K_1^3 sends messages to K_1^1 (indicating that $t33$ has occurred) but K_1^1 does not send any message to the other components. In the design of the distributed controller K_1, we have privileged the transitions $t31$ and $t22$. As a result, in the initial state of $RG(K_1)/G$, philosopher φ_1 can grab both forks f_1 and f_2 *in parallel* while philosophers φ_2 and φ_3 can only fight over fork f_3. Similar controllers may be obtained by choosing $t12$ and $t33$, or $t23$ and $t11$, instead of $t31$ and $t22$. Unfortunately, neither K_1 nor such controllers can avoid starvation. In $RG(K_1)/G$, for each philosopher there actually exists a reachable state in which he may act fast enough to prevent the other two from ever eating! Finally, note that all places of the local controller nets K_1^1, K_1^2 and K_1^3 stay bounded by 1 in all reachable states of $RG(K_1)/G$. Therefore, to transform K_1 into an equivalent asynchronous message passing automaton $AMPA$, as indicated in Sect. 2, it suffices to compute the bounded reachability graph $RG_B(K_1)$ for the bound $B = 1$.

3.2 Another DPND controller avoiding deadlocks

A quite different distributed controller K_2 may be obtained by shifting the focus to the transitions $t31$ and $t11$. By inspecting the reachability graph G of N again, one can locate the two subgraphs shown in Fig. 4(a). All squares in the figure are distributed diamonds. A first attempt to avoid the deadlock states 25 and 11 consists of manually removing from G all occurrences of the transitions $t31$ and $t11$ indicated in Fig. 4(a). Unfortunately, the two deadlock states can still be reached after these transitions have been removed. In a second effort, the occurrences of the transitions $t12$ and $t33$ indicated in Fig. 4(b) are manually removed from G. Let G_2 be the reachable restriction of the resulting subgraph of G. G_2 has 22 states, none of which is a sink state. From G_2 SYNET produces a distributed controller net K_2 such that G_2 is isomorphic to $RG(K_2)/G$. K_2 has four components K_2^1, K_2^2, K_2^3, K_2^4 to plug in the respective locations 1 to 4. K_2^2, K_2^3 and K_2^4 do nothing but send K_2^1 asynchronous signals informing this local controller of the occurrences of the sets of events $\{t12\}$, $\{t33\}$ and $\{se1, st1, se3, st3\}$, respectively. The local controller K_2^1 is depicted in Fig. 5(a).

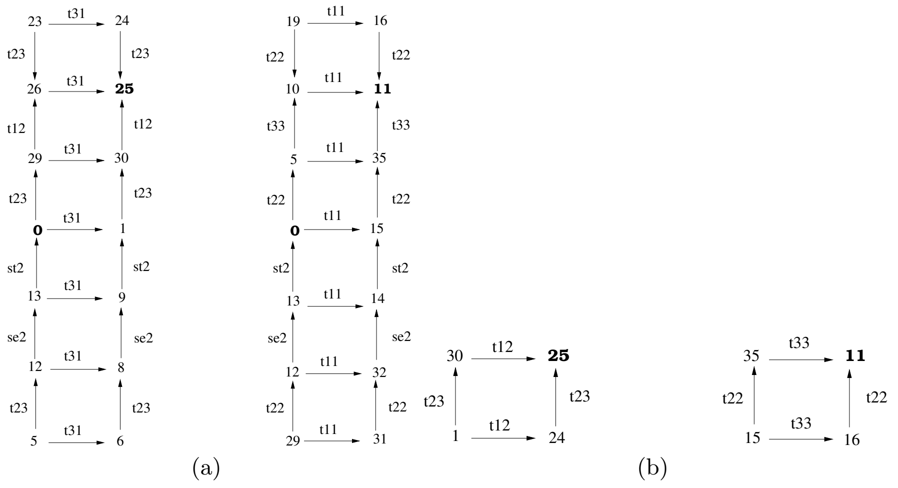
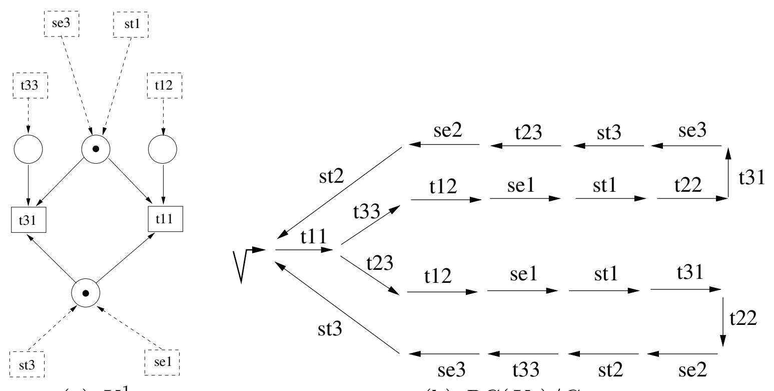


Fig. 4. Subgraphs leading to deadlock

Note that now, all control decisions are taken in location 1! In the initial state of $RG(K_2)/G$, philosopher φ_2 can take both forks f_2 and f_3 *in parallel*, while philosophers φ_1 and φ_3 can only compete with φ_2 to get forks f_2 and f_3, respectively. Note that all places of the local controller nets K_2^1 stay bounded by 1 in all reachable states of $RG(K_2)/G$. With respect to starvation, K_2 and all similar controllers have exactly the same drawbacks as K_1. K_1 and K_2 are incomparable controllers, and we suspect, but have not verified, that they are both maximally permissive amongst DPND controllers for the deadlock avoidance problem.

3.3 A distributed controller avoiding starvation

Using SYNET we produced a DPND controller K_3 that avoids starvation (and not just deadlocks). Specifically, we designed the right controller by examining the structure of cycles in the reachability graph of G, and used SYNET to confirm that it could be implemented with a distributed Petri net. The reachable state graph $RG(K_3)/G$ is shown in Fig. 5(b). Unfortunately, parallelism disappears completely, and there remains only one state where a choice is possible. It would be interesting to examine the same problem for a larger number of philosophers, but fully-automated strategies are necessary for this purpose.



(a) K_2^1 (b) $RG(K_3)/G$

Fig. 5. (a) An unfair controller (b) A fair controller

4 Conclusion

To move towards a theory of distributed control of DES, we have proposed a mixture of asynchronous control and communication. A significant advantage of the proposed methodology is that the way of encoding the information to be exchanged is automated: the messages sent asynchronously are names of places of a Petri net produced by synthesis. Yet there remains a sizeable amount of work to be done: a significant disadvantage of the methodology is at present the lack of a general theorem and a fully-automated controller synthesis method.

References

1. E. Badouel and P. Darondeau. Theory of Regions. In *Lectures on Petri Nets I: Basic Models*, Advances in Petri Nets, LNCS 915, pages 529-586, 1998.
2. E. Badouel, B. Caillaud and P. Darondeau. Distributing Finite Automata Through Petri Net Synthesis. *Formal Aspects of Computing* 13: 447-470, 2002.
3. G. Barrett and S. Lafortune. Decentralized Supervisory Control with Communicating Controllers. *IEEE Trans. Autom. Control*, 45(9), 1620-1638, 2000.
4. D. Brand and P. Zafiropulo. On Communicating Finite-State Machines. *J. of the ACM*, 30(2):323-342, 1983.
5. K. Cai and W.M. Wonham. Supervisor Localization: A Top-Down Approach to Distributed Control of Discrete-Event Systems. *IEEE Trans. Autom. Control*, 55(3), 605-618, 2010.
6. P. Darondeau. Distributed Implementation of Ramadge-Wonham Supervisory Control with Petri Nets. In *CDC-ECC 2005*, pages 2107-2112.
7. A. Giua, F. Di Cesare and M. Silva. Generalized Mutual Exclusion Constraints on Nets with Uncontrollable Transitions. In *IEEE-SMC 1992*, pages 974-979.
8. B. Genest, H. Gimbert, A. Muscholl and I. Walukiewicz. Optimal Zielonka-Type Construction of Deterministic Asynchronous Automata. In *ICALP 2010* (2), LNCS 6199, pages 52-63.
9. A. Ghaffari, N. Rezg and X. Xie. Algebraic and Geometric Characterization of Petri Net Controllers Using the Theory of Regions. In *WODES 2002*, pages 219-224.

10. A. Ghaffari, N. Rezg and X. Xie. Design of Live and Maximally Permissive Petri Net Controller Using the Theory of Regions. *IEEE Trans. Robot. Autom.* 19: 137-142, 2003.
11. R.J. van Glabbeek and W.P. Weijland. Branching Time and Abstraction in Bisimulation Semantics. *J. of the ACM* 43(3): 555-600, 1996.
12. N. Halbwachs, P. Caspi, P. Raymond and D. Pilaud. The synchronous dataflow programming language Lustre. *Proc. IEEE*, 79(9):1305-1320, 1991.
13. K. Hiraishi. On Solvability of a Decentralized Supervisory Control Problem with Communication. *IEEE Trans. Autom. Control*, 54(3), 468-480, 2009.
14. C.A.R. Hoare. Communicating Sequential Processes. *CACM* 21(8): 666-677, 1978.
15. G. Kalyon, T. Le Gall, H. Marchand and T. Massart. Synthesis of Communicating Controllers for Distributed Systems. *Private communication*, 2011.
16. H. Lamouchi and J. Thistle. Effective Control Synthesis for DES under Partial Observations. In *CDC 2000*, pages 22-28.
17. L. Lamport. Arbiter-Free Synchronization. *Distrib. Comput.* 16(2/3): 219-237, 2003.
18. F. Lin and W.M. Wonham. On observability of discrete-event systems. *Info. Sci.* 44:173-198, 1988.
19. A. Mannani and P. Gohari. Decentralized Supervisory Control of Discrete-Event Systems Over Comunication Networks. *IEEE Trans. Autom. Control*, 53(2):547-559, 2008.
20. M. Mukund and M. Sohoni. Gossiping, Asynchronous Automata and Zielonka's Theorem. Report TCS-94-2, Chennai Mathematical Institute, 1994.
21. P. Madhusudan, P.S. Thiagarajan, S. Yang. The MSO Theory of Connectedly Communicating Processes. In *FSTTCS 2005*, LNCS 3821, pages 201-212.
22. D. Potop-Butucaru and B. Caillaud. Correct-by-Construction Asynchronous Implementation of Modular Synchronous Specifications. In *ACSD 2005*, pages 48-57.
23. P.J. Ramadge and W.M. Wonham. Supervisory Control of a Class of Discrete Event Processes. *SIAM J. Control Optim.*, 25:206-230, 1987.
24. P.J. Ramadge and W.M. Wonham. The Control of Discrete Event Systems. *Proc. of the IEEE, Special issue on Dynamics of Discrete Event Systems*, 77:81-98, 1989.
25. S.L. Ricker and B. Caillaud. Mind the Gap: Expanding Communication Options in Decentralized Discrete-Event Control. In *CDC 2007*, pages 5924-5929.
26. S.L. Ricker. Asymptotic Minimal Communication for Decentralized Discrete-Event Control. In *WODES 2008*, pages 486-491.
27. K. Rudie, S. Lafortune and F. Lin. Minimal Communication in a Distributed Discrete-Event System. *IEEE Trans. Autom. Control*, 48(6):957-975, 2003.
28. K. Rudie and W.M. Wonham. Think Globally, Act Locally: Decentralized Supervisory Control. *IEEE Trans. Autom. Control*, 37(11):1692-1708, 1992.
29. B. Caillaud. http://www.irisa.fr/s4/tools/synet/
30. J.G. Thistle. Undecidability in Decentralized Supervision. *Syst. Control Lett.*, 54: 503-509, 2005.
31. S. Tripakis. Decentralized Control of Discrete Event Systems with Bounded or Unbounded Delay Communication. *IEEE Trans. Autom. Control*, 49(9):1489-1501, 2004.
32. W. Wang, S. Lafortune and F. Lin. Minimization of Communication of Event Occurrences in Acyclic Discrete Event Systems. *IEEE Trans. Autom. Control*, 53(9):2197-2202, 2008.
33. Y. Wang, S. Lafortune, T. Kelly, M. Kudlur and S. Mahlke. The Theory of Deadlock Avoidance via Discrete Control. In *POPL 2009*, pages 252-263.

34. X. Xu and R. Kumar. Distributed State Estimation in Discrete Event Systems. In *ACC 2009*, pages 4735-4740.
35. T.S. Yoo and S. Lafortune. A General Architecture for Decentralized Supervisory Control of Discrete-event Systems. *Discrete Event Dyn. Syst.*, 12(3):335-377, 2002.
36. K. Yamalidou, J. Moody, M. Lemmon and P. Antsaklis. Feedback Control on Petri Nets Based on Place Invariants. *Automatica* 32(1): 15-28, 1996.
37. W. Zielonka. Notes on Finite Asynchronous Automata. *RAIRO Informatique Théorique et Applications*, 21:99-135, 1987.

Appendix

Our Asynchronous Message Passing Automata (AMPA) differ from the communicating automata originally introduced by Brand and Zafiropulo [4] in that communications are not FIFO. Given a finite set of locations $\mathcal{L}$, a *B-bounded AMPA* is a collection of DFA $\{A_\ell \mid \ell \in \mathcal{L}\}$ together with a finite set of messages P, where $\lambda : P \to \mathcal{L}$ specifies the address for each message. Each $A_\ell = (Q_\ell, \Sigma_\ell \uplus \Sigma_\ell^! \uplus \Sigma_\ell^?, \delta_\ell, q_{0,\ell})$ is a DFA with a partial transition map δ_ℓ, and initial state $q_{0,\ell}$, where $\Sigma_\ell^! = \{\ell!p \mid p \in P \wedge \lambda(p) \neq \ell\}$ and $\Sigma_\ell^? = \{\ell?p \mid p \in P \wedge \lambda(p) = \ell\}$. The actions in Σ_ℓ are observable. The communication actions in $\Sigma_\ell^! \cup \Sigma_\ell^?$ are unobservable.

The dynamics of a B-bounded AMPA are defined by a transition system $RG(AMPA)$ constructed inductively from an initial configuration $\langle \overline{q_0}, \overline{m_0} \rangle$ as follows:

- $\overline{q_0}$ is an $\mathcal{L}$-indexed vector with entries $q_{0,\ell}$ for all $\ell \in \mathcal{L}$,
- $\overline{m_0}$ is an P-indexed vector with null entries for all $p \in P$,
- From any configuration $\langle \overline{q}, \overline{m} \rangle$, where $\overline{q}$ is an $\mathcal{L}$-indexed vector with entries $q_\ell \in Q_\ell$, for all $\ell \in \mathcal{L}$, and $\overline{m}$ is a P-indexed vector of integers with entries $m_p \geq 0$ for all $p \in P$, there is a transition $\langle \overline{q}, \overline{m} \rangle \xrightarrow{\sigma} \langle \overline{q'}, \overline{m'} \rangle$ in the following three cases:
  - $q'_\ell = \delta_\ell(q_\ell, \sigma)$ for some $\ell \in \mathcal{L}$ and $\sigma \in \Sigma_\ell$, $q'_k = q_k$ for all $k \neq \ell$ and $\overline{m'} = \overline{m}$;
  - $q'_\ell = \delta_\ell(q_\ell, \sigma)$ for some $\ell \in \mathcal{L}$ and $\sigma = \ell!p \in \Sigma_\ell^!$, $q'_k = q_k$ for all $k \neq \ell$, $m'_p = m_p + 1 \leq B$, and $m'_r = m_r$ for all $r \neq p$;
  - $q'_\ell = \delta_\ell(q_\ell, \sigma)$ for some $\ell \in \mathcal{L}$ and $\sigma = \ell?p \in \Sigma_\ell^?$, $q'_k = q_k$ for all $k \neq \ell$, $m'_p = m_p - 1 \geq 0$, and $m'_r = m_r$ for all $r \neq p$;

Branching bisimulation was defined by van Glabbeek and Weijland [11] for processes with a single unobservable action τ. The following is an adaptation of the original definition to processes defined by automata with several unobservable actions. Let $\Sigma = \Sigma_o \cup \Sigma_{uo}$ be a set of labels, where Σ_o and Σ_{uo} are the subsets of observable and unobservable labels, respectively. Let $A = (Q, \Sigma, \delta, q_0)$ and $A' = (Q', \Sigma, \delta, q'_0)$ be two automata over Σ. A and A' are *branching bisimilar* if there exists a symmetric relation $R \in Q \times Q' \cup Q' \times Q$ such that $(q_0, q'_0) \in R$ and whenever $(r, s) \in R$

- if $\delta(r, \sigma) = r'$ and $\sigma \in \Sigma_{uo}$, then $(r', s) \in R$;
- if $\delta(r, \sigma) = r'$ and $\sigma \in \Sigma_o$, then there exists a sequence $\sigma'_1 \ldots \sigma'_k \in \Sigma_{uo}^*$ (where $k = 0$ means an empty sequence) such that if one lets $\delta(s, \sigma'_1 \ldots \sigma'_j) = s'_j$ for $j \leq k$, and $\delta(s'_k, \sigma) = s'$, then s' and states s'_j are effectively defined and $(r', s') \in R$.

Mining with User Interaction

Robin Bergenthum, Sebastian Mauser

Department of Software Engineering, FernUniversität in Hagen
{robin.bergenthum,sebastian.mauser}@fernuni-hagen.de

Abstract. In this short paper we present an interactive mining approach which is based on net synthesis. First, a net is generated from a log file by a liberal mining algorithm such as the alpha-algorithm. Then, using concepts from the theory of regions, runs of this net which are not included in the log are calculated and feedbacked to a user who has to decide whether they are valid runs of the process or not. Finally, a net having the runs of the log and the additionally specified runs is synthesized.

1 Introduction

Many of today's information systems record information about performed activities of processes in so called event logs. Process mining techniques attempt to extract useful, structured information from such logs. In this paper we focus on the problem of constructing a process model which matches the actual workflow of the recorded information system, called process discovery. There are many process discovery techniques in the literature (e.g. [1]), often implemented in the ProM framework (www.promtools.org/prom6).

One main difficulty of process discovery is that a typical log contains only example runs of the recorded process (we do not discuss the problem of noise here), i.e. logs are incomplete. Therefore, precise mining algorithms based on net synthesis which exactly reproduce a log (see e.g. [2]) are often not appropriate. Consequently, most mining algorithms try to generate a process model which includes the recorded example runs and also allows for some additional behavior – they overapproximate the given event log. Since there is no information on runs missing in the log, overapproximation is a heuristic approach. When a mining algorithm allows a run which is not given in the log, it is not clear (without additional information) whether the added run (1) is a run of the process which coincidentally has not been observed in the log or (2) is no possible run of the process and for this reason is of course not included in the log. The only way to solve this problem is to ask a user whether (1) or (2) is true. Therefore, we here suggest an interactive mining approach where overapproximation is explicitly determined by a user (first ideas in this direction have been developed in [3]). Note that a crucial assumption of this approach is the existence of a process owner having enough insights to decide if a feedbacked run is a run of the process or not.

The crucial challenge of this approach is to choose an appropriate set of runs for user feedback. Of course it is not viable to feedback each run which is not given in the log. It is important to only consider such runs which have a "high probability" for being runs of the process. The basic idea of our approach is to feedback the difference between a liberally mined net and an exactly synthesized net.

Starting with the given log, we first use a liberal mining algorithm to generate a net representing an upper bound. Using the theory of regions we compute the runs which are allowed by this net but not included in the log. For each such run, a user has to decide whether it represents valid behavior of the process or not. Thus, the log is completed according to the feedback. The resulting log is then used as the input for an exact synthesis algorithm, i.e. a model is generated which allows the recorded runs and the runs explicitly specified by the user. In this way overapproximation becomes controllable. In particular, the conflict between underfitting, i.e. too much overapproximation, and overfitting, i.e. not enough overapproximation, can be solved.

2 Interactive Mining Algorithm

In this section we explain the interactive mining approach in detail. As a preparative step for our algorithm, we have to choose a mining algorithm (e.g. the alpha-algorithm [1]) which we use as a reference for the overapproximation performed by our approach. In the field of conformance checking there exist the notions of recall and precision. Our interactive algorithm heavily depends on the recall and precision of the underlying mining algorithm. While our approach works with any mining algorithm, an algorithm having a high recall is preferred, since our interactive approach always generates a net which allows all the behavior given in the log (we assume that there is no noise in the given log file). The precision of the underlying mining algorithm determines the search space of the interactive approach. The larger the search space the more queries are necessary.

Having chosen a reference mining algorithm, the interactive mining algorithm is as follows (see Figure 1):

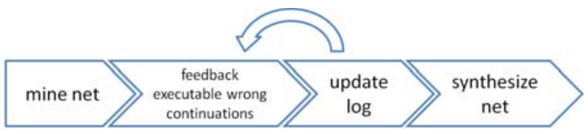


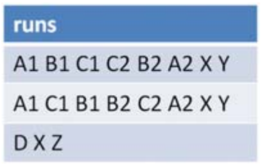
Fig. 1. Interactive mining algorithm.

- The starting point is a typical event log as described e.g. in [1, 2] which defines a set of runs. As a first step, the chosen mining algorithm is applied to the log generating a net which represents an upper bound for user feedback. The idea behind this approach is that a liberal mining algorithm performs a “reasonable” overapproximation but has a tendency to introduce “too much” overapproximation.
- The second step is to compute runs to be feedbacked to the user. For this purpose, we consider so called wrong continuations [2] of the given log. A wrong continuation extends a run (or more precisely the Parikh-vector of a run) of the log by one event such that the resulting run is not specified. That means, only minimal non-specified behavior is considered for feedback. Still the set of all wrong continuations is too large for practical purposes. Therefore, we only consider wrong

continuations which are allowed by our upper bound, i.e. which are executable w.r.t. the net mined by the chosen liberal mining algorithm. This set of wrong continuations is presented to the user for feedback. For each such run, the user then has to decide whether it is a run of the underlying process or not. For this purpose, the runs are illustrated in a list, and for each run there is a respective checkbox.

– As a third step, the log is updated by adding the runs which have been classified by the user as valid runs of the process. For all these new runs, new wrong continuations appear which have not yet been feedbacked. Therefore, there is a choice for the user now: He can decide to either proceed with the next final step (fourth step) or to repeat the feedback steps (second and third step) with the new wrong continuations. The latter choice means that the new wrong continuations which are executable w.r.t. the initially mined net are again feedbacked to the user and possibly added to the log. The second and third step can be iterated until no more new valid runs are found or the upper bound is reached.
– The fourth step consists of using an exact synthesis algorithm on the updated log, i.e. a net reproducing the log which has been completed by the user in the previous steps is generated. We here apply the mining algorithm based on regions of languages from [2, 4] which is implemented in the tool VipTool.

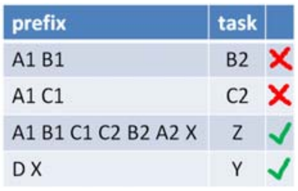
We now illustrate this procedure by a small example log which defines the three runs shown in Figure 2.



runs
A1 B1 C1 C2 B2 A2 X Y
A1 C1 B1 B2 C2 A2 X Y
D X Z

Fig. 2. Runs of the example log.

As a reference mining algorithm we use the alpha-algorithm for this example. The alpha-algorithm tries to extract from a log the direct dependencies of activities and translates them into places of a workflow net. Figure 4 shows a ProM screenshot of the net mined from the example log with the alpha-algorithm. This net overapproximates the log. It allows the three runs of the log and eleven further runs. The exact synthesis algorithm of VipTool generates the net shown in Figure 5 from the example log. It coincidentally has the same places as the net from Figure 4 and some additional places which are shown in grey (in general such a relation does not hold). Note that, since the log cannot precisely be represented by a Petri net, this net not only allows the three runs of the log but also two additional runs. The idea of our mining approach is now to construct a net in between the liberally mined net from Figure 4 and the synthesized net from Figure 5 by interacting with a user. That means, on the net level in this example the question is whether the grey places of Figure 5 should be included in the net or not.



prefix	task	
A1 B1	B2	✗
A1 C1	C2	✗
A1 B1 C1 C2 B2 A2 X	Z	✓
D X	Y	✓

Fig. 3. Feedback of wrong continuations.

Algorithmically, in the first step of our algorithm we mine the net from Figure 4. Then, in the second step first the set of all wrong continuations of the log is computed. For instance, A1 B1 is a prefix of a run of the log (see Figure 2). Since A1 B1 B2 is not contained in the log and extends the previous run by one event, it is a wrong continuation. Altogether, there are 97 wrong continuations in our example. However, only four of the 97 wrong continuations are enabled in the net from Figure 4. That means, for feedback we only consider these four runs which are given in Figure 3.

For each of these four wrong continuations the user has to decide whether in the process the last task can occur after the occurrence of the given prefix. In the context of the first two wrong continuations the question is if B2 (resp. C2) can immediately occur after B1 (resp. C1) or if B2 (resp. C2) has to be ordered behind C1 (resp. B1). For our example let us assume that the second case is true, i.e. the two wrong continuations are marked to be no runs of our process. In the context of the third and fourth wrong continuation the question is if task Z (resp. Y) can freely be chosen at the end of the process also in the case that the part of the net including A1, B1, C1, C2, B2, A2 (resp. D) has been chosen at the beginning of the process or if the choice of Z (resp. Y) depends on the choice at the beginning of the process. Here we assume that the first case is true, i.e. the two wrong continuations are marked as runs of our process. After this user feedback, in the third step of our algorithm the two positively evaluated wrong continuations are added to the log. In the following, a repetition of the feedback steps is not necessary in our example, since all wrong continuations of the two newly added runs are not enabled in the net from Figure 4, i.e. the upper bound is reached. Thus, in the fourth step, the VipTool synthesis algorithm is applied to the completed log with five runs. Since we use an exact synthesis method, for this step we only need the original log and the positively evaluated runs here. The result is the net from Figure 5 without the two grey places connected with the tasks Y and Z, i.e. there is a free choice between Y and Z.

3 Alternative Approach

In the described approach, the liberally mined net is just used as the upper bound for feedback when completing the log. Afterwards, this net is dropped and the final net is constructed from scratch by using synthesis methods. However, as it is also the case for our example, the net mined by a typical liberal mining approach is often nicely readable. The places generated by such mining algorithms are mostly introduced according to simple rules which in a natural way reflect the dependencies given in the log, i.e. they often nicely reflect the intuition of users of the business process. Thus, an interesting idea for the construction of the final net in our interactive mining approach is to keep the initially mined places and to only complement them by newly synthesized places as far as necessary. In this way, we support the generation of an intuitive and readable net.

Still, keeping the initially mined places causes the following problem: For most mining algorithms it is possible that so called unfeasible places which prohibit runs given in the log are generated. These places have to be deleted for enabling the approach of this section.

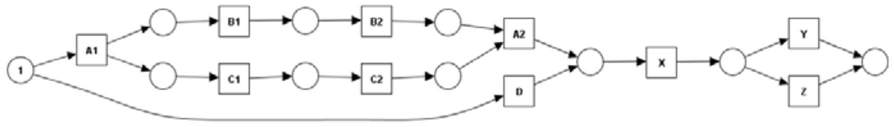


Fig. 4. Net mined with the alpha-algorithm of ProM.

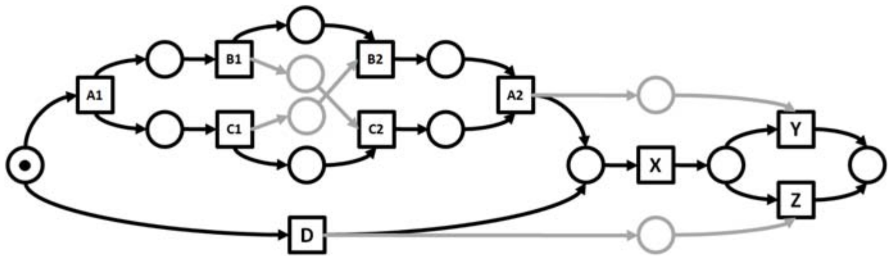


Fig. 5. Net synthesized with VipTool.

Altogether this alternative approach yields the algorithm shown in Figure 6. The difference to the algorithm in Figure 1 is the additional step "delete unfeasible places" and the changed last step. In this new last step, the runs which have explicitly been specified by the user to be no runs of the process are considered. For this set of runs, places which prohibit this set but are feasible w.r.t. the updated log (i.e. they allow the runs of the log) are computed and added to the initially mined net.

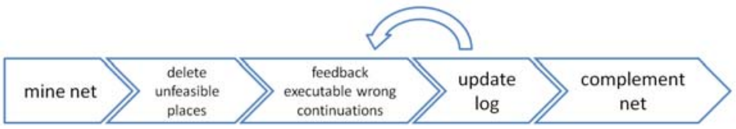


Fig. 6. Alternative mining algorithm.

For the example log of Figure 2, again first the net given in Figure 4 is generated using the alpha-algorithm of ProM. In this example, all places are feasible, i.e. all runs of the log are executable in the initially mined net. Thus, the same set of wrong continuations as in the first presented algorithm (Figure 3) are feedbacked to the user in a first feedback step. We assume that the user makes the same choices as in the last section. Consequently, the last two wrong continuations are added to the log and the first two wrong continuations have to be stored to be prohibited later on. As before, after the first feedback step the upper bound is reached. Therefore, it is proceeded with the last step of the algorithm. With standard synthesis methods, the algorithm computes two regions, each prohibiting one of the two stored wrong continuations. The initially mined net is then extended by the two places corresponding to the two regions yielding the same net as in the case of the first interactive approach.

When comparing the two approaches, it is a coincidence that we get the same result in our example. In particular, it was a coincidence of the first approach that the resulting synthesized net included the liberally mined net from Figure 4. In this alternative approach, it is the central idea to keep the initially mined net and to extend it with additional places to always get such a nice result. Moreover, we do not anymore focus on completing the log but on identifying runs to be prohibited. As a consequence, in the case a user decides to stop the feedback phase, i.e. to not finish the completion of the log, there is an important difference to the algorithm presented in the last section. While the algorithm of the last section tries to prohibit all the behavior which has not been feedbacked anymore in such case, the algorithm of this section keeps the liberal control flow of the initially mined net.

Finally, it remains to mention that processes with loops are problematic for the presented basic interactive mining approaches. If a loop is included in the liberally mined net, it is possible to repeat the feedback steps of the interactive approaches arbitrarily often and thus the user has to cancel the iterated feedback at some point. In the case of the first algorithm, the precise mining approach of VipTool then tries to detect such loops from the extended log as described in [4]. In the case of the second algorithm, the loop is simply kept from the liberally mined net. But, if it is not detected by the approach from [4], it is still possible that the loop is coincidentally prohibited by the additional places. In this context several improvements are possible, in particular mechanisms to automatically stop the feedback steps in the case of loops and improved methods for loop detection.

4 Conclusion

In this paper, we tackle the problem of incomplete logs by applying mining with user interaction. We introduce overapproximation in a user controlled way. The starting point is a net generated by a liberal mining algorithm. The behavior of this net is then restricted by applying concepts from the theory of regions. We presented two different approaches implementing this idea.

As future work it is important to both evaluate the quality of our interactive mining approach and the practicability of the approach for users.

References

1. van der Aalst, W.M.P., Weijters, T., Maruster, L.: Workflow Mining: Discovering Process Models from Event Logs. IEEE Trans. Knowl. Data Eng. **16**(9) (2004) 1128–1142
2. Bergenthum, R., Desel, J., Lorenz, R., Mauser, S.: Process Mining Based on Regions of Languages. In: BPM 2007, LNCS 4714, Springer (2007) 375–383
3. Esparza, J., Leucker, M., Schlund, M.: Learning Workflow Petri Nets. In: Petri Nets 2010, LNCS 6128, Springer (2010) 206–225
4. Bergenthum, R., Desel, J., Kölbl, C., Mauser, S.: Experimental Results on Process Mining Based on Regions of Languages. In: Workshop CHINA, Petri Nets 2008, X'ian (2008)

Do Petri Nets Provide the Right Representational Bias for Process Mining?

(short paper)

W.M.P. van der Aalst

Department of Mathematics and Computer Science,
Technische Universiteit Eindhoven, The Netherlands.
`W.M.P.v.d.Aalst@tue.nl`

Abstract. Process discovery is probably the most challenging process mining task. Given an event log, i.e., a set of example traces, it is difficult to automatically construct a process model explaining the behavior seen in the log. Many process discovery techniques use Petri nets as a language to describe the discovered model. This implies that the search space—often referred to as the *representational bias*—includes many inconsistent models (e.g., models with deadlocks and livelocks). Moreover, the low-level nature of Petri nets does not help in finding a proper balance between overfitting and underfitting. Therefore, we advocate a new representation more suitable for process discovery: *causal nets*. Causal nets are related to the representations used by several process discovery techniques (e.g., heuristic mining, fuzzy mining, and genetic mining). However, unlike existing approaches, C-nets use declarative semantics tailored towards process mining.

1 Challenges in Process Mining

Process mining is an emerging research area combining techniques from process modeling, model-based analysis, data mining, and machine learning. The goal is to extract knowledge about processes from event data stored in databases, transaction logs, message logs, etc. Process mining techniques are commonly classified into: (a) *discovery*, (b) *conformance*, and (c) *enhancement* [2]. In this paper, we restrict ourselves to control-flow discovery, i.e., learning a process model based on example traces.

A *trace* is a *sequence of events* for a particular *process instance* (also referred to as *case*). Events refer to some *activity*. For example, the trace $\langle a, b, c, d\rangle$ refers to a process execution starting with activity a and ending with activity d. An *event log* is a multiset of traces, e.g., $L = \{\langle a, b, c, d\rangle^{25}, \langle a, c, b, d\rangle^{35}, \langle a, e, d\rangle^{30}\}$ describes the execution sequences of 90 cases. There are dozens of process discovery techniques that are able to construct a process model from such an event log. Many of these techniques use Petri nets as a target representation [4,5,7,12,19,22,23]. Given event log L, these techniques have no problems discovering the Petri net in which, after a, there is a choice between doing b and *c* concurrently or just *e*, followed by *d*. Note that this example is misleadingly simple as process discovery based on real-life event logs is extremely challenging.

Generally, we use four main quality dimensions for judging the quality of the discovered process model: *fitness* (the model should allow for the behavior observed), *simplicity* (the model should be as simple as possible), *precision* (the model should not allow for behavior that is very unlikely given the event log), and *generalization* (the model should not just represent the observed examples and also allow for behavior not yet observed but very similar to earlier behavior).

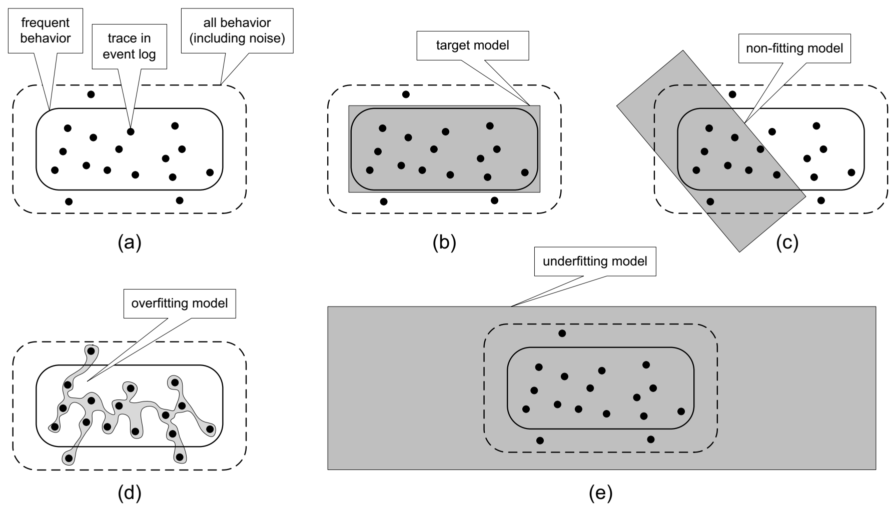


Fig. 1. Illustrating typical problems encountered when discovering process models from event logs: (c) a non-fitting model, (d) an overfitting model (poor generalization), and (e) an underfitting model (poor precision)

The simplicity dimension refers to *Occam's Razor*; the simplest model that can explain the behavior seen in the log, is the best model. Figure 1 explains some of the main challenges related to the other three quality dimensions. Each black dot represents a trace (i.e., a sequence of activities) corresponding to one or more cases in the event log. (Recall that multiple cases may have the same corresponding trace.) An event log typically contains only a fraction of the possible behavior, i.e., the dots should only be seen as *samples* of a much larger set of possible behaviors. Moreover, one is typically primarily interested in frequent behavior and not in all possible behavior, i.e., one wants to abstract from *noise* (i.e., infrequent or exceptional behavior) and therefore not all dots need to be relevant for the process model to be constructed.

It is interesting to analyze such noisy behaviors. However, when constructing the overall process model, the inclusion of infrequent or exceptional behavior leads to complex diagrams. Moreover, it is typically impossible to make reliable statements about noisy behavior given a relatively small set of observations. Figure 1(a) distinguishes between frequent behavior (solid rectangle with rounded corners) and all behavior (dashed rectangle), i.e., normal and noisy behavior. The difference between normal and noisy behavior is a matter of definition, e.g., normal behavior could be defined as the 80% most frequently occurring traces.

Let us assume that the two rectangles with rounded corners can be determined by observing the process infinitely long while the process is in steady-state (i.e., no concept drift [9]). Based on these assumptions, Fig. 1 sketches four discovered models depicted by shaded rectangles. These discovered models are based on the example traces in the log, i.e., the black dots. The "ideal process model" (Fig. 1(b)) allows for the behavior coinciding with the frequent behavior seen when the process would be observed ad infinitum. The "non-fitting model" in Fig. 1(c) is unable to characterize the process well as it is not even able to capture the examples in the event log used to learn the model. The "overfitting model" (Fig. 1d)) does not generalize and only says something about the examples in the current event log. New examples will most likely not fit into this model. The "underfitting model" (Fig. 1(e)) lacks precision and allows for behavior that would never be seen if the process would be observed ad infinitum.

Figure 1 illustrates the challenges that process discovery techniques need to address: *How to extract a simple target model that is not underfitting, overfitting, nor non-fitting?*

2 Petri nets as a Representational Bias for Process Mining

One can think of process mining as a search problem with a *search space* defined by the class of process models considered, i.e., the goal is to find a "best" process model in the collection of all permissible models. The observation that the target language defines the search space is often referred to as the *representational bias*.

Many process discovery techniques use Petri nets as a representational bias [4,5,7,12,19,22,23]. Examples of such techniques are the α-algorithm and its variants [5,22], state-based region techniques [4,19], and language-based region techniques [7,23]. Some of these techniques allow for labeled transitions, i.e., there may be invisible/silent steps (τ transitions not leaving a mark in the event log) or multiple transitions with the same label. However, all of the models have a clearly defined initial marking and one or more final markings. In fact, most techniques aim at discovering a so-called *workflow net* (WF-net) [1]. A WF-net has one source place (modeling the start of the process) and one sink place (modeling the end), and all nodes are on a path from source to sink. Ideally, such a discovered WF-net is *sound*. Soundness is a common correctness criterion for WF-nets requiring that from any reachable marking it is possible to reach the final marking (weak termination) and there are no dead transitions (i.e., there are no activities that can never happen).

In the remainder, we assume that the goal is to discover sound WF-nets from event logs. The particular soundness notion used is not very relevant. Moreover, the syntactical requirements imposed on WF-nets may be relaxed. However, a basic assumption of any process discovery algorithm is that all traces in the event log start in some initial state and ideally end in a well-defined end state.

Petri nets allow for a wide variety of analysis techniques and provide a simple, yet powerful, graphical representation. This is the reason why they were chosen as a target language for dozens of process discovery techniques described in literature [4,5,7,12,19,22,23]. Nevertheless, in this paper, we pose the question *"Are Petri nets a suitable representational bias for process discovery?"*.

In our view, there are several problems associated to using Petri nets as a representational bias.

– *The search space is too large (including mostly "incorrect" models).* When randomly generating a Petri net, the model is most likely not sound. The fraction of sound process models is small. As a result, most of the process discovery techniques tend to create incorrect process models. For example, the α-algorithm can generate models that have deadlocks and livelocks. Region-based techniques may also suffer from such problems; they can replay the event log but also exhibit deadlocks and livelocks.
– *Petri nets cannot capture important process patterns in a direct manner.* Process modeling languages used by end-users tend to support higher-level constructs, often referred to as workflow patterns [3]. Examples are the OR-split (Multi-Choice pattern) and OR-join (Synchronizing Merge pattern). Many of these patterns can be expressed in terms of Petri nets, i.e., the higher-level construct is mapped onto a small network. This is no problem for model-based analysis (e.g., verification). However, the discovered process model needs to be interpreted by the end-user. Whereas it is relatively easy to translate higher-level constructs to Petri nets, it is difficult to translate lower-level constructs to languages such as BPMN, EPCs, UML, YAWL, etc.
– *It is difficult to "invent" modeling elements.* If all transitions need to have a unique visible label, then the only task of a process discovery algorithm is to "invent" places. If two transitions can have the same visible label, then the process discovery algorithm may also need to duplicate transitions. If transitions can be silent, e.g., to skip an activity, then the process discovery algorithm needs to "invent" such silent transitions (if needed). Places, duplicate transitions, and silent transitions cannot be coupled directly to observations in the event log. The fact that such modeling elements need to be "invented" makes the search space larger (often infinite) and the relation between event log and model more indirect.
– *The representational bias does not help in finding a proper balance between overfitting and underfitting.* Because of the low-level nature of Petri nets, there are no natural patterns to support generalization. Algorithms tend to overfit or underfit the event log. One of the reasons is that the representational bias does not help in guiding the discovery algorithm towards a desirable model. Note that some of the more advanced region-based algo-

rithms allow for the formulation of additional constraints (e.g., the target model should be free-choice and the number of input and output arcs per node is bounded) [23].

Note that the above problems are not specific for Petri nets. Most of the current representations suffer from a subset of these problems. Consider for example BPMN; the fraction of sound BPMN models is small and the mining algorithm needs to "invent" process fragments consisting of gateways and events to capture behavior adequately. However, compared to Petri nets, BPMN can capture more patterns directly.

3 Towards a Better Representational Bias: Causal Nets

The goal of this paper is *not* to provide a solution for all of the problems induced by using Petri nets as a representational bias for process mining. Instead, we would like to discuss potential notations that provide a *more suitable representational bias.* We do not propose new discovery techniques. Instead, we note that most of the existing process discovery techniques can be modified to support a more refined representational bias.

To trigger this discussion, we advocate a new representation more suitable for process discovery: *causal nets* (C-nets) [2]. On the one hand, C-nets are related to the representations used by several process discovery techniques (e.g., heuristic mining [17,21], fuzzy mining [17], and genetic mining [18]). Moreover, in [6] a similar representation is used for conformance checking. On the other hand, C-nets use declarative semantics not based on a local firing rule. This way a larger fraction of models (if not all) is considered to be correct.

A C-net is a graph where nodes represent *activities* and arcs represent *causal dependencies.* Each activity has a set of possible *input bindings* and a set of possible *output bindings.* Consider, for example, the causal net shown in Fig. 2. Activity a has only an empty input binding as this is the start activity. There are two possible output bindings: $\{b, d\}$ and $\{c, d\}$. This means that a is followed by either b and d, or c and d. Activity e has two possible input bindings ($\{b, d\}$ and $\{c, d\}$) and three possible output bindings ($\{g\}$, $\{h\}$, and $\{f\}$). Hence, e is preceded by either b and d, or c and d, and is succeeded by just g, h or f. Activity z is the end activity having two input bindings and one output binding (the empty binding). This activity has been added to create a unique end point. All executions commence with start activity a and finish with end activity z. Note that unlike, Petri nets, there are no places in the causal net; the routing logic is solely represented by the possible input and output bindings.

Definition 1 (Causal net [2]). *A* Causal net *(C-net) is a tuple* $C = (A, a_i, a_o, D, I, O)$ *where:*

- A *is a finite set of* activities*;*
- $a_i \in A$ *is the* start activity*;*
- $a_o \in A$ *is the* end activity*;*

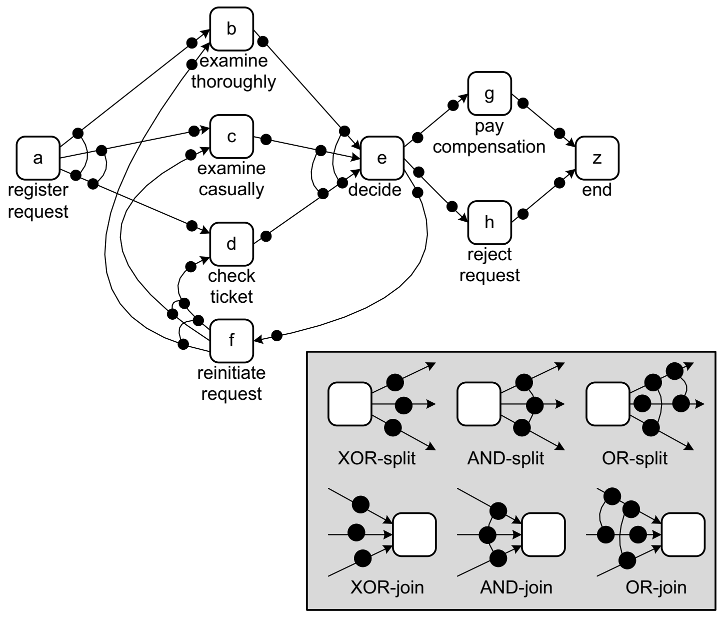


Fig. 2. Example of a C-net and some of the typical input and output bindings present in conventional business process modeling languages [2]

– $D \subseteq A \times A$ *is the* dependency relation,
– $AS = \{X \subseteq \mathcal{P}(A) \mid X = \{\emptyset\} \ \vee \ \emptyset \notin X\}$;[1]
– $I \in A \rightarrow AS$ *defines the set of possible* input bindings *per activity; and*
– $O \in A \rightarrow AS$ *defines the set of possible* output bindings *per activity,*

such that

– $D = \{(a_1, a_2) \in A \times A \mid a_1 \in \bigcup_{as \in I(a_2)} as\}$;
– $D = \{(a_1, a_2) \in A \times A \mid a_2 \in \bigcup_{as \in O(a_1)} as\}$;
– $\{a_i\} = \{a \in A \mid I(a) = \{\emptyset\}\}$;
– $\{a_o\} = \{a \in A \mid O(a) = \{\emptyset\}\}$; *and*
– *all activities in the graph* (A, D) *are on a path from* a_i *to* a_o.

An *activity binding* is a tuple (a, as^I, as^O) denoting the occurrence of activity a with input binding as^I and output binding as^O. For example, $(e, \{b, d\}, \{f\})$ denotes the occurrence of activity e in Fig. 2 while being preceded by b and d, and succeeded by f. A *binding sequence* σ is a sequence of activity bindings. A possible binding sequence for the C-net of Fig. 2 is $\sigma_{ex} = \langle(a, \emptyset, \{b, d\})$, $(b, \{a\}, \{e\})$, $(d, \{a\}, \{e\})$, $(e, \{b, d\}, \{g\})$, $(g, \{e\}, \{z\})$, $(z, \{g\}, \emptyset)\rangle$.

[1] $\mathcal{P}(A) = \{A' \mid A' \subseteq A\}$ is the powerset of A. Hence, elements of AS are *sets of sets* of activities.

A binding sequence is *valid* if a predecessor activity and successor activity always "agree" on their bindings. For a predecessor activity x and successor activity y we need to see the following "pattern": $\langle \dots, (x, \{\dots\}, \{y, \dots\}), \dots, (y, \{x, \dots\}, \{\dots\}), \dots \rangle$, i.e., the occurrence of activity x with y in its output binding needs to be followed by the occurrence of activity y and the occurrence of activity y with x in its input binding needs to be preceded by the occurrence of activity x. σ_{ex} is an example of a valid sequence.

For technical details regarding these notions we refer to [2]. It is important to note that *the behavior of C-nets is limited to valid binding sequences.* C-nets are not driven by local firing rules (like a Petri net), but by the more declarative notion of valid binding sequences in which activities always "agree" on their bindings.

It can be shown that *C-nets are more expressive than Petri nets.* For any sound WF-net one can construct a C-net such that any full firing sequence of the WF-net corresponds to a valid binding sequence of the C-net and vice versa. Note that at first sight, C-nets seem to be related to *zero-safe nets* [10]. The places in a zero-safe net are partitioned into stable places and zero places. Observable markings only mark stable places, i.e., zero places need to be empty. In-between observable markings zero places may be temporarily marked. In [15] an approach is described to synthesize zero-safe nets. However, zero places cannot be seen as bindings because the "agreement" between two activities may be non-local, i.e., an output binding may create the obligation to execute an activity occurring much later in the process.

For process discovery, the representational bias provided by C-nets is more suitable than the representational bias provided by Petri nets.

– Since the behavior of C-nets is limited to valid binding sequences, any C-net is in principle correct. Therefore, we do not need to consider a search space in which most models are internally inconsistent (deadlocks, etc.)
– C-nets can capture important process patterns in a direct manner. For example, OR-splits (Multi-Choice pattern) and OR-joins (Synchronizing Merge pattern) can be modeled directly. Moreover, there is no need to introduce silent transitions or multiple transitions with the same label to discover a suitable model for event logs such as $L = [\langle a, b, c \rangle^{20}, \langle a, c \rangle^{30}]$. C-nets are closely connected to languages such as BPMN, EPCs, UML, YAWL, etc. However, the interpretation is different as we only consider valid binding sequences. Models may be "cleaned up" as a post optimization.
– There is no need to "invent" modeling elements such as places and silent transitions. We only need to find the set of possible input and output bindings per activity. Note that input and output bindings have a more direct connection to the event log than routing elements encountered in conventional languages such as Petri nets (places and silent/duplicate transitions), BPMN (gateways, events, etc.), and EPCs (connectors and events).
– The representational bias of C-nets is tailored towards finding a proper balance between overfitting and underfitting. It is easy to define the types of input and output bindings that are preferred, e.g., an AND-split or XOR-

split is preferred over an OR-split. For example, it is possible to associate thresholds to extending the set possible bindings.

4 Conclusion

This short paper does *not* aim to provide a new process discovery algorithm. Instead, its purpose is to trigger a discussion on the representational bias used by existing process mining algorithms. We showed that Petri nets are less suitable as a target language. We introduced C-nets as an alternative representational bias. C-nets are able to express behavioral patterns in a more direct manner. Moreover, by limiting the behavior of C-nets to valid binding sequences, we obtain a more suitable search space. We believe that our formalization sheds new light on the representations used in [6,17,18,20,21].

It is interesting to investigate how classical region theory [8,11,13,14,16] can be applied to the synthesis of C-nets. For example, it seems possible to adapt region-based mining approaches as described in [23] to C-nets. However, the straightforward encoding of the synthesis problem into an Integer Linear Programming (ILP) problem makes discovery intractable for realistic examples. Moreover, existing region-based mining techniques have problems dealing with noise and incompleteness. As a result, the discovered models typically do not provide a good balance between overfitting and underfitting.

C-nets are very suitable for genetic process mining [18]. It is also possible to use a mixture of heuristic mining [20,21] and genetic mining [18]. For example, one can first discover the dependency relation D using heuristics and then optimize the input bindings I and output bindings O using genetic algorithms. Genetic operators such as crossover and mutation can be defined on C-nets in a straightforward manner. The fitness function can be based on replay, e.g., the fraction of events and process instances in the log that fit into the model. Here we suggest using the technique described in [6]. Moreover, we also suggest to incorporate the complexity of the model in the fitness function.

Currently, ProM already provides basic support for C-nets (ProM 6 can be downloaded from www.processmining.org). In the future, we aim to add more plug-ins working directly on C-nets.

References

1. W.M.P. van der Aalst. The Application of Petri Nets to Workflow Management. *The Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.
2. W.M.P. van der Aalst. *Process Mining: Discovery, Conformance and Enhancement of Business Processes*. Springer-Verlag, Berlin, 2011.
3. W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow Patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.
4. W.M.P. van der Aalst, V. Rubin, H.M.W. Verbeek, B.F. van Dongen, E. Kindler, and C.W. Günther. Process Mining: A Two-Step Approach to Balance Between Underfitting and Overfitting. *Software and Systems Modeling*, 9(1):87–111, 2010.

5. W.M.P. van der Aalst, A.J.M.M. Weijters, and L. Maruster. Workflow Mining: Discovering Process Models from Event Logs. *IEEE Transactions on Knowledge and Data Engineering*, 16(9):1128–1142, 2004.
6. A. Adriansyah, B.F. van Dongen, and W.M.P. van der Aalst. Towards Robust Conformance Checking. In M. zur Muehlen and J. Su, editors, *BPM 2010 Workshops, Proceedings of the Sixth Workshop on Business Process Intelligence (BPI2010)*, volume 66 of *Lecture Notes in Business Information Processing*, pages 122–133. Springer-Verlag, Berlin, 2011.
7. R. Bergenthum, J. Desel, R. Lorenz, and S. Mauser. Process Mining Based on Regions of Languages. In G. Alonso, P. Dadam, and M. Rosemann, editors, *International Conference on Business Process Management (BPM 2007)*, volume 4714 of *Lecture Notes in Computer Science*, pages 375–383. Springer-Verlag, Berlin, 2007.
8. R. Bergenthum, J. Desel, R. Lorenz, and S. Mauser. Synthesis of Petri Nets from Finite Partial Languages. *Fundamenta Informaticae*, 88(4):437–468, 2008.
9. R.P. Jagadeesh Chandra Bose, W.M.P. van der Aalst, I.Zliobaite, and M. Pechenizkiy. Handling Concept Drift in Process Mining. In H. Mouratidis and C. Rolland, editors, *International Conference on Advanced Information Systems Engineering (Caise 2011)*, volume 6741 of *Lecture Notes in Computer Science*, pages 391–405. Springer-Verlag, Berlin, 2011.
10. R. Bruni and U. Montanari. Zero-Safe Nets: Comparing the Collective and Individual Token Approaches. *Information and Computation*, 156(1-2):46–89, 2000.
11. M.P. Cabasino, A. Giua, and C. Seatzu. Identification of Petri Nets from Knowledge of Their Language. *Discrete Event Dynamic Systems*, 17(4):447–474, 2007.
12. J. Carmona and J. Cortadella. Process Mining Meets Abstract Interpretation. In J.L. Balcazar, editor, *ECML/PKDD 210*, volume 6321 of *Lecture Notes in Artificial Intelligence*, pages 184–199. Springer-Verlag, Berlin, 2010.
13. J. Cortadella, M. Kishinevsky, L. Lavagno, and A. Yakovlev. Deriving Petri Nets from Finite Transition Systems. *IEEE Transactions on Computers*, 47(8):859–882, August 1998.
14. P. Darondeau. Unbounded Petri Net Synthesis. In J. Desel, W. Reisig, and G. Rozenberg, editors, *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 413–438. Springer-Verlag, Berlin, 2004.
15. P. Darondeau. On the Synthesis of Zero-Safe Nets. In *Concurrency, Graphs and Models*, volume 5065 of *Lecture Notes in Computer Science*, pages 364–378. Springer-Verlag, Berlin, 2008.
16. A. Ehrenfeucht and G. Rozenberg. Partial (Set) 2-Structures - Part 1 and Part 2. *Acta Informatica*, 27(4):315–368, 1989.
17. C.W. Günther and W.M.P. van der Aalst. Fuzzy Mining: Adaptive Process Simplification Based on Multi-perspective Metrics. In G. Alonso, P. Dadam, and M. Rosemann, editors, *International Conference on Business Process Management (BPM 2007)*, volume 4714 of *Lecture Notes in Computer Science*, pages 328–343. Springer-Verlag, Berlin, 2007.
18. A.K. Alves de Medeiros, A.J.M.M. Weijters, and W.M.P. van der Aalst. Genetic Process Mining: An Experimental Evaluation. *Data Mining and Knowledge Discovery*, 14(2):245–304, 2007.
19. M. Sole and J. Carmona. Process Mining from a Basis of Regions. In J. Lilius and W. Penczek, editors, *Applications and Theory of Petri Nets 2010*, volume 6128 of *Lecture Notes in Computer Science*, pages 226–245. Springer-Verlag, Berlin, 2010.

20. A.J.M.M. Weijters and W.M.P. van der Aalst. Rediscovering Workflow Models from Event-Based Data using Little Thumb. *Integrated Computer-Aided Engineering*, 10(2):151–162, 2003.
21. A.J.M.M. Weijters and J.T.S. Ribeiro. Flexible Heuristics Miner (FHM). BETA Working Paper Series, WP 334, Eindhoven University of Technology, Eindhoven, 2010.
22. L. Wen, W.M.P. van der Aalst, J. Wang, and J. Sun. Mining Process Models with Non-Free-Choice Constructs. *Data Mining and Knowledge Discovery*, 15(2):145–180, 2007.
23. J.M.E.M. van der Werf, B.F. van Dongen, C.A.J. Hurkens, and A. Serebrenik. Process Discovery using Integer Linear Programming. *Fundamenta Informaticae*, 94:387–412, 2010.