Fundamenta Informaticae XX (2008) 1–32
IOS Press

Synthesis of Petri Nets from Finite Partial Languages

Robin Bergenthum, Jörg Desel, Robert Lorenz, Sebastian Mauser

Department of Applied Computer Science

Catholic University of Eichstätt-Ingolstadt

85072 Eichstätt, Germany

firstname.lastname@ku-eichstaett.de

Abstract. In this paper we present two algorithms that effectively synthesize a finite place/transition Petri net (p/t-net) from a finite set of labeled partial orders (a finite partial language). The synthesized p/t-net either has exactly the non-sequential behavior specified by the partial language, or there is no such p/t-net.

The first algorithm is based on the theory of *token flow regions* for partial languages developed by Lorenz and Juhás. Thus, this paper shows the applicability of this concept. The second algorithm uses the classical theory of regions applied to the set of step sequences generated by the given partial language. We finally develop an algorithm to test whether the net synthesized by either of the two algorithms has exactly the non-sequential behavior specified by the partial language.

We implemented all algorithms in our framework VipTool. In this paper, the implementations of the first two algorithms are used to compare the algorithms by means of experimental results.

1. Introduction

Synthesis of Petri nets from behavioral descriptions has been a successful line of research since the 1990s. There is a rich body of nontrivial theoretical results, and there are important applications in industry, in particular in hardware system design [8, 18], and recently also in workflow design [2]. Moreover, there are several synthesis tools that are based on the theoretical results. The most prominent one is Petrify [7].

Originally, synthesis meant algorithmic construction of an unlabeled Petri net from sequential observations. It can be applied to various classes of Petri nets, including elementary nets [14, 15, 12] and place/transition nets (p/t-nets) [3, 4]. Synthesis can start with a transition system representing the sequential behavior of a system or with a step transition system which additionally represents steps of concurrent events [3]. Synthesis can also be based on a language (a set of occurrence sequences or step sequences [9]). The *synthesis problem* is to decide whether, for a given behavioral specification, there

exists an unlabeled Petri net of the respective class such that the behavior of this net coincides with the specified behavior.

Up to now, the synthesis problem was only studied for sequential specifications and for specifications based on concurrent steps. The aim of this paper is to solve the synthesis problem for p/t-nets where the behavior is given in terms of a finite partial language L, i.e. as a finite set of labeled partial orders (LPOs – also known as *partial words* [16] or *pomsets* [26]). Partial order behavior of Petri nets truly represents the concurrency of events and is often considered the most appropriate representation of behavior of Petri net models. Moreover, we provide according synthesis algorithms. For typical applications of language based Petri net synthesis (e.g. process mining [1], business process design [10], software synthesis (UML)) the consideration of finite sets of LPOs is sufficient.

We start with a finite set of partially ordered sets of events together with a labeling function associating a transition to each event. Thus, a single possible run (of the unknown p/t-net) is represented by an LPO of events. The ordering relation defines a *possible ordering of the transition occurrences*: if events e and e' are ordered ($e < e'$), and if moreover e is labeled by t and e' is labeled by t', then in this run t can occur before t'. It might be the case that t' becomes enabled by the occurrence of t and hence both occurrences are causally dependent. However, in contrast to *occurrence net semantics*, we do not demand causal dependency between the transition occurrences. If two events e and e' are not ordered (neither $e < e'$ nor $e' < e$), then the respective transitions occur concurrently. We call an LPO *enabled* w.r.t. a given p/t-net if it represents a possible order (in the above sense) of transition occurrences of this p/t-net.

As in previous synthesis approaches, we apply the so called *theory of regions*. Since regions are concepts defined for the behavioral specification, we will define regions for LPOs.

All approaches to Petri net synthesis based on regions roughly follow the same idea:

- Instead of solving the synthesis problem (is there a net with the specified behavior?) and then – in the positive case – synthesizing the net, first a net is constructed from the given specification.

- The construction starts with the transitions taken from the behavioral specification.

- Since this net has too much concurrency, its behavior will be restricted by the addition of places and according initial markings. In particular, a place constitutes a dependency relation between the occurrences of the transitions in its pre-set and the occurrences of the transitions in its post-set.

- Each region yields a corresponding place in the constructed net. A region is defined in such a way that the behavior of the net with its corresponding place still includes the specified behavior.

- When all, or sufficiently many, regions are identified, all places of the synthesized net are constructed. The crucial point for this step is that the set of all regions can be very large or even infinite, whereas in most cases finite, smaller sets of regions suffice to represent all relevant dependencies.

- If the behavior of the synthesized net coincides with the specified behavior (where coincide is defined by an appropriate notion of isomorphism), then the synthesis problem has a positive solution; otherwise there is no Petri net with the specified behavior and therefore the synthesis problem has a negative solution.

In [17] regions for trace languages are presented without deducing effective synthesis algorithms. In [9] regions of regular languages and in [3] regions for step transition systems are proposed. In both cases polynomial synthesis algorithms are developed. The three definitions of regions are consistent. Combining ideas of [17, 9, 3] yields a synthesis algorithm for sets of step sequences. This algorithm can be adapted to a synthesis algorithm for a set of LPOs by considering the set of step sequences generated by the set of LPOs.

Let us explain this idea by means of a simple example, shown in Figure 1. This figure shows a simple LPO with five events, labeled by transition names a, b and c. Viewed as a run of a Petri net, this LPO states that transition a occurs first. Then transitions b and c occur concurrently, and after each of these occurrences there is another occurrence of a. The generated step sequences of this single LPO are given on the right hand side of Figure 1. Formally, a step sequence is generated by an LPO if it adds causality to this LPO, where transition occurrences in one step are considered to be concurrent and transition occurrences in different steps are considered to occur in the order given by the sequence. Since we start with languages given by sets of LPOs, we have to consider the union of all step sequences generated by LPOs of this language.

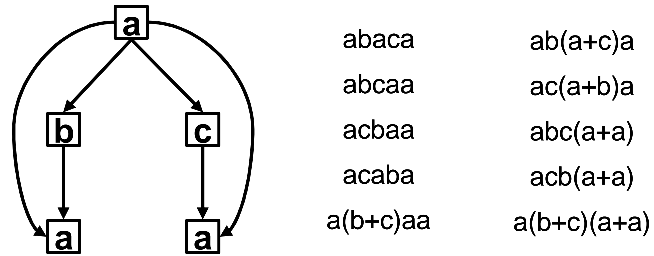

Figure 1. Set of maximal step sequences generated by an LPO.

Now the classical theory of regions can be applied to this set of step sequences. This synthesis approach suffers from the problem that the number of step sequences may be exponential in the size of the LPOs. Thus, this approach cannot yield a (worst case) efficient synthesis algorithm.

An alternative idea to synthesize a net from a set of LPOs is to directly define regions of the LPOs. Such regions are presented in [22], called *token flow regions* in the present paper. As the main theoretical result of this paper, we derive an effective synthesis algorithm from token flow regions.

Both algorithms are given in this paper and their complexity is compared. Since both algorithms do not solve the last step of the region based synthesis procedure described above (comparison of the behavior of the synthesized net with the specified behavior) we provide algorithms to perform this step as well.

The remainder of the paper is organized as follows: we start with a brief introduction to LPOs, partial languages, p/t-nets and enabled LPOs in Section 2. In Section 3 we present the synthesis algorithm based on token flow regions. In Subsection 3.1 we recall definitions and main results from [22] on the theory of regions for partial languages. In the subsequent subsections we develop the main theoretical results of this paper. In Subsection 3.2 we show how to compute regions as integer solutions of an homogenous linear inequation system, we prove that a finite set of basis solutions generating the set of all solutions already appropriately represents the set of all regions, and we briefly discuss implementation and performance details of the presented synthesis algorithm. Subsection 3.3 presents methods to test whether the synthesized finite p/t-net has exactly the specified non-sequential behavior. In Section 4 we present the synthesis approach based on regions of the set of step sequences generated by a partial language. Subsection 4.1 develops the synthesis algorithm. Subsection 4.2 compares its performance with the performance of the synthesis algorithm developed in Section 3, and Subsection 4.3 discusses further

variants of the synthesis of nets from partial languages. Finally, in Section 5 we provide experimental results on the time consumption of both presented synthesis algorithms. All described algorithms are presented compactly in pseudo code in algorithm boxes.

This paper is an extended version of the conference paper [21]. In the conference paper we presented parts of Section 3 without considering implementation and performance details.

2. Preliminaries

In this section we recall the definitions of *labeled partial orders* (LPOs), *partial languages*, *place/transition nets* (p/t-nets) and LPOs *enabled w.r.t. p/t-nets*. We start with basic mathematical notations: by $\mathbb{N}$ we denote the *nonnegative integers*. $\mathbb{N}^+$ denotes the positive integers. Given a function f from A to B and a subset C of A, we write $f|_C$ to denote the *restriction* of f to the domain C. Given a finite set A, the symbol $|A|$ denotes the *cardinality* of A. The set of all *multi-sets* over a set A is the set $\mathbb{N}^A$ of all functions $f : A \to \mathbb{N}$. Addition $+$ on multi-sets is defined as usual by $(m+m')(a) = m(a)+m'(a)$. We also write $\sum_{a \in A} m(a)a$ to denote a multi-set m over A and $a \in m$ to denote $m(a) > 0$. Given a binary relation $R \subseteq A \times A$ over a set A, the symbol R^+ denotes the *transitive closure* of R. We write aRb to denote $(a, b) \in R$. A *directed graph* is a pair $(V, \to)$, where V is a finite *set of vertices* and $\to \subseteq V \times V$ is a binary relation over V, called the *set of edges*. All graphs considered in this paper are finite.

Definition 2.1. (Partial order)
A *partial order* is a directed graph $po = (V, <)$, where $<$ is irreflexive and transitive.

Two nodes $v, v' \in V$ of a partial order $(V, <)$ are called *independent* if $v \not< v'$ and $v' \not< v$. By $\mathrm{co}_< \subseteq V \times V$ we denote the set of all pairs of independent nodes of V. A *co-set* is a subset $C \subseteq V$ fulfilling: $\forall x, y \in C : x \,\mathrm{co}_<\, y$. A *cut* is a maximal co-set. For a co-set C of a partial order $(V, <)$ and a node $v \in V \setminus C$ we write $v < (>)\, C$, if $v < (>)\, s$ for an element $s \in C$ and $v \,\mathrm{co}_<\, C$, if $v \,\mathrm{co}_<\, s$ for all elements $s \in C$. A partial order $(V', <')$ is a *prefix* of another partial order $(V, <)$ if $V' \subseteq V$, $(v' \in V' \wedge v < v') \Longrightarrow (v \in V')$ and $<' = < \cap V' \times V'$.

A partial order $po = (V, <)$ is called *linear* (or *total*) if $\mathrm{co}_< = id_V$, and *stepwise linear* if $\mathrm{co}_<$ is transitive. Given partial orders $po_1 = (V, <_1)$ and $po_2 = (V, <_2)$, po_2 is a *sequentialization* of po_1 if $<_1 \subseteq <_2$. If po_2 is linear, it is called *linearization* of po_1 and if po_2 is stepwise linear, it is called *step linearization* of po_1. We use partial orders with nodes (called events) *labeled by transition names* to specify scenarios describing the behavior of Petri nets.

Definition 2.2. (Labeled partial order)
A *labeled partial order* (LPO) is a triple $lpo = (V, <, l)$, where $(V, <)$ is a partial order, and $l : V \to T$ is a *labeling function* with *set of labels* T.

We use the above notations defined for partial orders also for LPOs. We will often consider LPOs only up to isomorphism. Two LPOs $(V, <, l)$ and $(V', <', l')$ are called *isomorphic*, if there is a bijective mapping $\psi : V \to V'$ such that $l(v) = l'(\psi(v))$ for $v \in V$, and $v < w \Longleftrightarrow \psi(v) <' \psi(w)$ for $v, w \in V$. By $[lpo]$ we will denote the set of all LPOs isomorphic to lpo. The LPO lpo is said to *represent* the isomorphism class $[lpo]$. The behavior of systems is formally specified by sets of (isomorphism classes of) LPOs. Such sets are called *partial languages*.

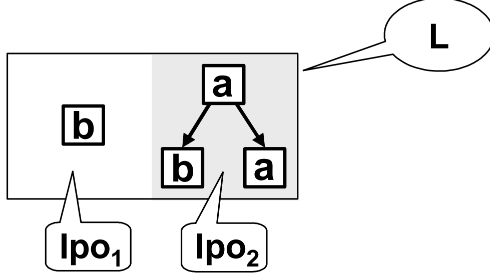

Figure 2. A partial language.

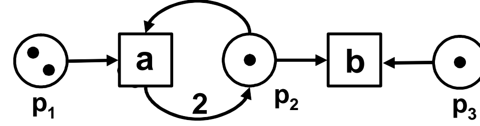

Figure 3. A marked p/t-net (N, m_0).

Definition 2.3. (Partial language)
Let T be a finite set. A set $\mathcal{L} \subseteq \{[lpo] \mid lpo$ is an LPO with set of labels $T\}$ is called *partial language over* T.

Usually, partial languages are given by sets of concrete LPOs representing isomorphism classes. We always assume that each label from T occurs in a partial language over T. Figure 2 shows a partial language represented by the set of LPOs $L = \{lpo_1, lpo_2\}$, which we will use as a running example.

A *net* is a triple (P, T, F), where P is a (possibly infinite) set of *places*, T is a finite set of *transitions* satisfying $P \cap T = \emptyset$, and $F \subseteq (P \times T) \cup (T \times P)$ is the *flow relation* of the net.

Definition 2.4. (Place/transition net)
A *place/transition-net* (*p/t-net*) N is a quadruple (P, T, F, W), where (P, T, F) is a net, and $W : F \to \mathbb{N}^+$ is a *weight function*.

We extend the weight function W to pairs of net elements $(x, y) \in (P \times T) \cup (T \times P)$ with $(x, y) \notin F$ by $W(x, y) = 0$. A *marking* of a net $N = (P, T, F, W)$ is a function $m : P \to \mathbb{N}$ assigning $m(p)$ tokens to a place $p \in P$, it is a multi-set over P. A *marked p/t-net* is a pair (N, m_0), where N is a p/t-net, and m_0 is a marking of N, called *initial marking*. Figure 3 shows a marked p/t-net (N, m_0). Places are drawn as circles including tokens representing the initial marking, transitions as rectangles and the flow relation as arcs annotated by the values of the weight function (the weight 1 is not shown).

A multi-set of transitions $\tau \in \mathbb{N}^T$ is called a *step* (of transitions). A step τ is *enabled to occur* (concurrently) in a marking m if and only if $m(p) \geq \sum_{t \in \tau} \tau(t) W(p, t)$ for each place $p \in P$. In this case, its occurrence leads to the marking $m'(p) = m(p) + \sum_{t \in \tau} \tau(t)(W(t, p) - W(p, t))$. We write $m \xrightarrow{\tau} m'$ to denote that τ is enabled to occur in m and that its occurrence leads to m'. A finite sequence of steps $\sigma = \tau_1 \ldots \tau_n,\ n \in \mathbb{N}$, is called a *step occurrence sequence enabled in a marking* m *and leading to* m_n, denoted by $m \xrightarrow{\sigma} m_n$, if there exists a sequence of markings $m_1, \ldots, m_n$ such that $m \xrightarrow{\tau_1} m_1 \xrightarrow{\tau_2} \ldots \xrightarrow{\tau_n} m_n$. In the marked p/t-net (N, m_0) from Figure 3 only the steps a and b are enabled to occur in the initial marking. In the marking reached after the occurrence of a, the step $a + b$ is enabled to occur. There are two equivalent formal notions of runs of p/t-nets defining non-sequential semantics based on [20, 30]. We only give the notion of enabled LPOs here: an LPO is enabled w.r.t. a marked p/t-net, if for each cut of the LPO the marking reached by firing all transitions corresponding to events smaller than the cut enables the step (of transitions) given by the cut.

Definition 2.5. (Enabled LPO)
Let (N, m_0) be a marked p/t-net, $N = (P, T, F, W)$. An LPO $lpo = (V, <, l)$ with $l : V \to T$ is called *enabled (to occur) in* (N, m_0) if $m_0(p) + \sum_{v \in V \wedge v < C}(W(l(v), p) - W(p, l(v))) \geq \sum_{v \in C} W(p, l(v))$ for every cut C of *lpo* and every $p \in P$. Its *occurrence* leads to the *final marking* m' given by $m'(p) = m_0(p) + \sum_{v \in V}(W(l(v), p) - W(p, l(v)))$.

Enabled LPOs are also called *runs*. The set of all isomorphism classes of LPOs enabled in (N, m_0) is $\mathfrak{Lpo}(N, m_0)$. $\mathfrak{Lpo}(N, m_0)$ is called the *partial language of runs* of (N, m_0).

There is an equivalent characterization of enabledness using step sequences and their correspondence to stepwise linear LPOs: a stepwise linear LPO $lpo' = (V, <', l)$ can be represented by the step sequence $\sigma_{lpo'} = \tau_1 \dots \tau_n$ defined by $V = V_1 \cup \dots \cup V_n$, $<' = \bigcup_{i<j} V_i \times V_j$ and $\tau_i(t) = |\{v \in V_i \mid l(v) = t\}|$ ($\sigma_{lpo'}$ is well defined, since $\mathrm{co}_{<'}$ is transitive.). An LPO $lpo = (V, <, l)$ is enabled in (N, m_0) if and only if, for each step linearization $lpo' = (V, <', l)$ of *lpo*, the step sequence $\sigma_{lpo'}$ is enabled in (N, m_0).

Observe that $\mathfrak{Lpo}(N, m_0)$ is always sequentialization and prefix closed, i.e. every sequentialization and every prefix of an enabled LPO is again enabled w.r.t. (N, m_0). Moreover, the set of labels of $\mathfrak{Lpo}(N, m_0)$ is always finite. Therefore, when specifying the non-sequential behavior of a searched p/t-net by a partial language, this partial language must necessarily be sequentialization and prefix closed and must have a finite set of labels. We assume that such a partial language $\mathcal{L}$ is given by a set of concrete LPOs L representing $\mathcal{L}$ in the sense that $[lpo] \in \mathcal{L} \iff \exists lpo' \in L : [lpo] = [lpo']$. Usually, we specify the non-sequential behavior by a set of concrete LPOs L which is *not* sequentialization and prefix closed and then consider the partial language $\mathcal{L}$ which emerges by adding all prefixes of sequentializations of LPOs in L. In this sense, the partial language $\mathcal{L}$ given by L in Figure 2 specifies the non-sequential behavior of a searched p/t-net. Both LPOs shown in this Figure are enabled w.r.t. the marked p/t-net (N, m_0) shown in Figure 3. It holds $\mathcal{L} = \{[lpo] \mid lpo$ is a prefix of a sequentialization of an LPO in $L\} = \mathfrak{Lpo}(N, m_0)$. Thus, (N, m_0) solves the synthesis problem w.r.t. L. In the following for technical reasons the LPOs in L are assumed to have pairwise disjoint node sets.[1]

3. Synthesis Based on Token Flow Regions

In this section we present an effective algorithm to synthesize a finite p/t-net from a partial language given by L. The algorithm is based on the definition of regions of partial languages presented in [22]. In this paper we denote these regions as *token flow regions*. We first briefly recall definitions and main results from [22] on the theory of token flow regions for partial languages. In the subsequent subsections we develop the main new results of this paper: we show how to compute token flow regions as integer solutions of an homogenous linear inequation system, we prove that a finite set of basis solutions generating the set of all solutions already appropriately represents the set of all token flow regions, and we briefly discuss implementation and performance details of the presented synthesis algorithm. Finally, we present methods to test whether the synthesized finite p/t-net has exactly the specified non-sequential behavior L.

[1] In particular this ensures that L requires all technical requirements used in [22] to prove Theorem 3.2.

3.1. Region-based Synthesis

We consider the problem of synthesizing a p/t-net from a partial language specifying its non-sequential behavior. As mentioned, such a partial language $\mathcal{L}$ is represented by a set of concrete LPOs L (which is not necessarily prefix or sequentialization closed). This means we develop an algorithm to compute a marked p/t-net (N, m_0) from a given set of LPOs L such that the partial language $\mathcal{L}$ emerging from L satisfies $\mathcal{L} = \mathfrak{Lpo}(N, m_0)$ (if such a net exists). In this section we recall the definitions and main results on region based synthesis from [22] in a consolidated version, which is better structured and easier to understand: we explain the ideas of region based synthesis in two independent parts, first defining axiomatically the so called saturated feasible net as the best upper approximation to a p/t-net having the specified behavior and second introducing the notion of token flow regions for the computation of this net.

3.1.1. Saturated Feasible Net

The idea to construct a net (N, m_0) solving the synthesis problem is as follows: the set of transitions of the searched net is the finite set of labels of L. Then each LPO in L is enabled w.r.t. the marked p/t-net consisting only of these transitions (having an empty set of places), because there are no causal dependencies between transitions. This net in general has many runs not specified by L. Thus, one restricts the behavior of this net by creating causal dependencies between the transitions through adding places. Such places are defined by their initial marking and the weights on the arcs connecting them to each transition (Figure 4).

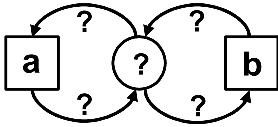

Figure 4. An unknown place of a p/t-net.

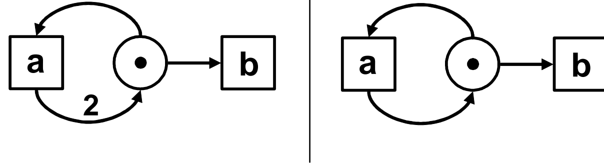

Figure 5. Left: a feasible place. Right: a place which is not feasible.

Two kinds of such places can be distinguished. In the case that there is an LPO in L which is not a run of the corresponding "one place"-net, this place restricts the behavior too much. Such a place is *non-feasible*. In the other case, the considered place is *feasible*.

Definition 3.1. (Feasible place)
Let $\mathcal{L}$ be a partial language over the finite set of labels T and let (N, m_p), $N = (\{p\}, T, F_p, W_p)$ be a marked p/t-net with only one place p. (N, m_p) is called associated to p. The place p is called *feasible (w.r.t. $\mathcal{L}$)*, if $\mathcal{L} \subseteq \mathfrak{Lpo}(N, m_p)$, otherwise *non-feasible (w.r.t. $\mathcal{L}$)*.

Figure 5 shows on the left side a place which is feasible w.r.t. the partial language specified by L in Figure 2. This is because, after the occurrence of a, the place is marked by 2 tokens. In this marking the step $a + b$ is enabled to occur (as specified by lpo_2). The place shown on the right side is non-feasible, because, after the occurrence of a, the place is again marked by only 1 token. In this marking the step $a + b$ is not enabled to occur. Thus lpo_2 is not enabled w.r.t. the one-place-net shown on the right side.

If we add all feasible places to the searched net, then the partial language of runs of the resulting net includes $\mathcal{L}$, and it is minimal with this property. We call this net the *saturated feasible net (w.r.t. $\mathcal{L}$)*. In general, the partial language of runs of the saturated feasible net is not necessarily equal to $\mathcal{L}$. If it is not equal to $\mathcal{L}$, there does not exist a marked p/t-net whose partial language of runs equals $\mathcal{L}$. The synthesis problem has a solution if and only if the partial language of runs of the saturated feasible net equals $\mathcal{L}$.

Definition 3.2. (Saturated feasible p/t-net)
Let $\mathcal{L}$ be a partial language over a finite set of labels T. The componentwise union of all nets associated to places feasible w.r.t. $\mathcal{L}$ is called *saturated feasible (w.r.t. $\mathcal{L}$)*.

Theorem 3.1. Let (N, m_0) be the saturated feasible net w.r.t. a partial language $\mathcal{L}$.

(i) $\mathcal{L} \subseteq \mathfrak{Lpo}(N, m_0)$.

(ii) The behavior of (N, m_0) is minimal with property (i):
$\forall (N' m_0') : (\mathfrak{Lpo}(N', m_0') \subsetneq \mathfrak{Lpo}(N, m_0)) \Longrightarrow (\mathcal{L} \not\subseteq \mathfrak{Lpo}(N', m_0'))$.

(iii) Either $\mathfrak{Lpo}(N, m_0) = \mathcal{L}$ or the synthesis problem has a negative answer.

Note that there are always infinitely many feasible places, because each place which is a non-negative linear combination of feasible places is feasible. For example, each place p_n with $W(a, p_n) = 2n$, $W(p_n, a) = n$, $W(p_n, b) = n$, $W(b, p_n) = 0$ and $m_0(p_n) = n$ is feasible w.r.t. the partial language given by L in Figure 2. Therefore, the problem of representing the infinite set of feasible places by a finite subset (restricting the behavior in the same way) must be solved.

3.1.2. Token Flow Regions

By so called *token flow regions* of partial languages it is possible to define the set of all feasible places structurally on the level of the partial language given by L. The idea of defining token flow regions of L is as follows: if two events x and y are ordered in an LPO $lpo = (V, <, l) \in L$ – this means $x < y$ – this specifies that the corresponding transitions $l(x)$ and $l(y)$ are causally dependent. Such a causal dependency arises exactly if the occurrence of transition $l(x)$ produces tokens in a place, and some of these tokens are consumed by the occurrence of the other transition $l(y)$. Such a place can be defined as follows: assign to every edge (x, y) of an LPO in L a natural number representing *the number of tokens which are produced by the occurrence of $l(x)$ and consumed by the occurrence of $l(y)$ in the place to be defined*. Then the number of tokens consumed overall by a transition $l(y)$ in this place is given as the sum of the natural numbers assigned to ingoing edges (x, y) of y. This number can then be interpreted as the weight of the arc connecting the new place with the transition $l(y)$. Similarly, the number of tokens produced overall by a transition $l(x)$ in this place is given as the sum of the natural numbers assigned to outgoing edges (x, y) of x, and this number can then be interpreted as the weight of the arc connecting the transition $l(x)$ with the new place. Moreover, transitions can also

- consume tokens from the initial marking of the new place (tokens which are not produced by another transition): in order to specify the number of such tokens, we extend an LPO by an *initial event* v_0 representing a transition producing the initial marking. The sum of the natural numbers assigned to outgoing edges (v_0, y) of the initial event v_0 can be interpreted as the initial marking of the new place.

- produce tokens in the new place which remain in the final marking after the occurrence of all transitions (tokens which are not consumed by some subsequent transition): in order to specify the number of such tokens, we extend an LPO by a *final event* v_{max} representing a transition consuming the final marking.

Figure 6 shows the LPOs lpo_1 and lpo_2 from Figure 2 extended by an initial and a final event. Such extensions we call $\star$-extensions of LPOs.

Definition 3.3. ($\star$-extension)
For a set of LPOs L we denote $W_L = \bigcup_{(V,<,l)\in L} V$, $E_L = \bigcup_{(V,<,l)\in L} <$ and $l_L = \bigcup_{(V,<,l)\in L} l$. A $\star$-*extension* $lpo^\star = (V^\star, <^\star, l^\star)$ of $lpo = (V, <, l)$ is defined by

(i) $V^\star = (V \cup \{v_0^{lpo}, v_{\max}^{lpo}\})$ with $v_0^{lpo}, v_{\max}^{lpo} \notin V$,

(ii) $<^\star = < \cup(\{v_0^{lpo}\} \times V) \cup (V \times \{v_{\max}^{lpo}\}) \cup \{(v_0^{lpo}, v_{\max}^{lpo})\}$,

(iii) $l^\star(v_0^{lpo}), l^\star(v_{\max}^{lpo}) \notin l(V)$, $l^\star(v_0^{lpo}) \neq l^\star(v_{\max}^{lpo})$ and $l^\star|_V = l$.

v_0^{lpo} is called the *initial event of lpo* and $v_{\max}^{lpo}$ the *maximal event of lpo*.
Let $lpo^\star = (V^\star, <^\star, l^\star)$ be a $\star$-extension of each $lpo \in L$ such that:

(iv) For each two LPOs $(V, <, l), (V', <', l') \in L$: $l^\star(v_0^{lpo}) = (l')^\star(v_0^{lpo'})$.

(v) For each two distinct LPOs $(V, <, l), (V', <', l') \in L$: $l^\star(v_{\max}^{lpo}), (l')^\star(v_{\max}^{lpo'}) \notin l_L(W_L), l^\star(v_{\max}^{lpo}) \neq (l')^\star(v_{\max}^{lpo'})$.

Then the set $L^\star = \{lpo^\star \mid lpo \in L\}$ is called $\star$-*extension of* L. We denote $W_L^\star = W_{L^\star}$, $E_L^\star = E_{L^\star}$ and $l_L^\star = l_{L^\star}$.

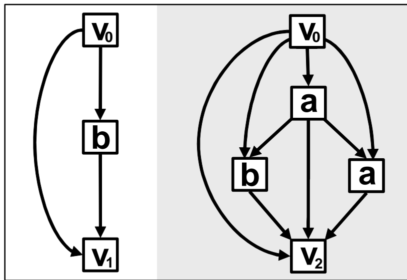

Figure 6. $\star$-extensions of LPOs.

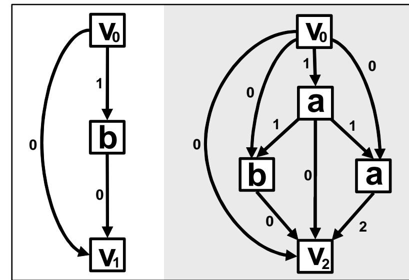

Figure 7. Token flow region of a partial language.

According to the above explanation, we can define a new place p_r by assigning in each LPO $lpo = (V, <, l) \in L$ a natural number $r(x, y)$ to each edge (x, y) of a $\star$-extension of lpo through a function $r : E_L^\star \to \mathbb{N}$:

- The sum of the natural numbers $In_{lpo}(y, r) = \sum_{x<^\star y} r(x, y)$ assigned to ingoing edges (x, y) of a node $y \in W_L$ defines $W(p_r, l(y)) = In_{lpo}(y, r)$. We call $In_{lpo}(y, r)$ the *in-token flow* of y.

- The sum of the natural numbers $Out_{lpo}(x,r) = \sum_{x <^\star y} r(x,y)$ assigned to outgoing edges (x,y) of a node $x \in W_L$ defines $W(l(x), p_r) = Out_{lpo}(x,r)$. We call $Out_{lpo}(x,r)$ the *out-token flow* of x.

- the sum of the natural numbers assigned to outgoing edges (v_0^{lpo}, y) of an initial node v_0^{lpo} (the out-token flow of v_0^{lpo}) defines $m_0(p_r) = Out_{lpo}(v_0^{lpo}, r)$. We call $Out_{lpo}(v_0^{lpo}, r)$ the *initial token flow* of lpo.

The value $r(x,y)$ we call the *token flow* between x and y. Since equally labeled nodes formalize occurrences of the same transition, this is well-defined only if equally labeled events have equal in-token flow and equal out-token flow. In particular all LPOs must have the same initial token flow. We say that $r : E_L^\star \to \mathbb{N}$ fulfills the properties (IN) and (OUT) on L if for all $lpo = (V, <, l), lpo' = (V', <', l') \in L$ and for all $v \in V^\star, v' \in (V')^\star$ holds:

(IN) $l(v) = l'(v') \Longrightarrow In_{lpo}(v,r) = In_{lpo'}(v',r)$.

(OUT) $l(v) = l'(v') \Longrightarrow Out_{lpo}(v,r) = Out_{lpo'}(v',r)$.

Observe that (OUT) in particular ensures that all LPOs have the same initial token flow. Altogether, each such function r fulfilling (IN) and (OUT) on L defines a place p_r. We call p_r *corresponding place* of r.

Definition 3.4. (Token Flow Region)
Let L be a set of LPOs which is sequentialization and prefix closed. Let further $\mathcal{L}$ be the partial language represented by L. A *token flow region* of $\mathcal{L}$ is a function $r : E_L^\star \to \mathbb{N}$ fulfilling (IN) and (OUT) on L.

If we define a function r fulfilling (IN) and (OUT) on a set of LPOs L which is not sequentialization and prefix closed, then this function is easily extended to a token flow region of the partial language defined by the set of all prefixes of sequentializations of LPOs in L without changing in- and outtoken flows of nodes:

- Assign the value 0 to each additional edge within a sequentialization of an LPO in L and keep the values of r on all other edges.

- Define r on a prefix of an LPO in L by gluing all nodes subsequent to the prefix to a maximal node of the prefix. If several edges are glued to one edge, then sum up the values of r on the glued edges. Keep the values of r on all remaining edges.

Thus, it is enough to specify a function fulfilling (IN) and (OUT) on some set of LPOs L, called *token flow region of* L, to define a token flow region of the partial language $\mathcal{L}$ defined by L. Figure 7 shows a function r fulfilling (IN) and (OUT) on the set L of LPOs given in Figure 2, which in this sense can be extended to a token flow region of the partial language defined by L. The corresponding place p_r is defined by $W(p_r, a) = 1$, $W(a, p_r) = 2$, $W(p_r, b) = 1$, $W(b, p_r) = 0$ and $m_0(p_r) = 1$ (p_r is the middle place of the p/t-net in Figure 3).

As the main result we showed in [22] that the set of places corresponding to token flow regions of a partial language equals the set of feasible places w.r.t. this partial language.[2]

[2]In [22] we assumed that the set of LPOs L representing $\mathcal{L}$ fulfills some technical requirements. These will be automatically fulfilled for all such sets L we consider in the following. Thus, we omit their detailed presentation here.

Theorem 3.2. ([22])
Let $\mathcal{L}$ be a partial language. Then it holds (i) that each place corresponding to a token flow region of $\mathcal{L}$ is feasible w.r.t. $\mathcal{L}$ and (ii) that each place feasible w.r.t. $\mathcal{L}$ corresponds to a token flow region of $\mathcal{L}$.

Thus the saturated feasible net can be given by the set of places corresponding to token flow regions:

Corollary 3.1. Let $\mathcal{L}$ be a partial language represented by the set of LPOs L. Denote $P = \{p_r \mid r \text{ is a token flow region of } \mathcal{L}\}$, T the set of labels of $\mathcal{L}$, $W(p_r, l_L(v)) = In_{lpo}(v, r)$ and $W(l_L(v), p_r) = Out_{lpo}(v, r)$ for $p_r \in P$ and some $lpo = (V, <, l) \in L$ with $v \in V$, $F = \{(x, y) \mid W(x, y) > 0\}$ and $m_{sat}(p_r) = Out_{lpo}(v_0^{lpo}, r)$ for $p_r \in P$ (and some $lpo \in L$). Then the p/t-net (N_{sat}, m_{sat}), $N_{sat} = (P, T, F, W)$, is the *saturated feasible p/t-net (w.r.t. $\mathcal{L}$).*

The saturated feasible net has infinitely many places, i.e. there are infinitely many token flow regions of $\mathcal{L}$. Moreover, even the description of one token flow region may be infinite, since there may exist infinitely many edges in $E_L^\star$. Therefore, we restrict ourselves in the following to finite partial languages, i.e. to partial languages which are represented by a finite set of LPOs L.

3.2. Computing a Finite Representation of all Regions

For finite partial languages we show in this subsection that the set of token flow regions can be computed as the set of non-negative integer solutions of a homogenous linear equation system $\mathbf{A} \cdot \mathbf{x} = \mathbf{0}$. It is well known that there is a finite set of basis solutions, such that every solution is generated as a non-negative linear sum of basis solutions. We prove that the set of places corresponding to basis solutions already restricts the behavior of the searched net in the same way as the set of all feasible places. Therefore there is a representation of the saturated feasible net by a net with finitely many places having the same partial language of runs. For this finite net it can be tested effectively if it has $\mathcal{L}$ as its partial language of runs (Subsection 3.3).

3.2.1. Computing Token Flow Regions

In this subsection we show how to compute token flow regions (and thus feasible places) of a partial language $\mathcal{L}$ represented by a finite set of LPOs L. For this, we rewrite the properties (IN) and (OUT) as a homogenous linear equation system $\mathbf{A}_L \cdot \mathbf{x} = \mathbf{0}$. To compute a token flow region r, we need to assign a value $r(x, y)$ to every edge $e = (x, y)$ in the finite set of edges $E_L^\star$. We interpret r as a $|E_L^\star|$-dimensional vector $\mathbf{x}_r = (x_1, \ldots, x_n)$, $n = |E_L^\star|$. Considering a fixed numbering of the edges in $E_L^\star = \{e_1, \ldots, e_n\}$, a value $r(e_i)$ equals x_i. Figure 8 shows a numbering of the edges of the $\star$-extension of the set of LPOs L given in Figure 2.

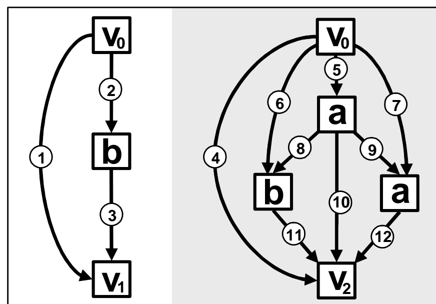

Figure 8. A numbering of edges.

Now, we encode the properties (IN) and (OUT) by a homogenous linear equation system $\mathbf{A}_L \cdot \mathbf{x} = \mathbf{0}$

in the sense that $r : E_L^\star \to \mathbb{N}$ fulfills (IN) and (OUT) on L if and only if $\mathbf{A}_L \cdot \mathbf{x}_r = \mathbf{0}$. This can be done by, loosely speaking, defining for pairs of equally labeled nodes a row $\mathbf{in}$ of $\mathbf{A}_L$ counting the token flow on ingoing edges of one node positively and of the other node negatively. Similarly, a row $\mathbf{out}$ of $\mathbf{A}_L$ counting the token flow on outgoing edges of one node positively and of the other node negatively can be defined. It is enough for each label t to ensure that the intoken (outtoken) flow of the first and second node with label t are equal that the intoken (outtoken) flow of the second and third node with label t are equal, and so on.

Formally, we denote $W_t = \{v \in W_L^\star \mid l_L^\star(v) = t\} = \{v_1^t, v_2^t, \ldots\}$ for all labels $t \in T$ and denote for $1 \leqslant m \leqslant |W_t| - 1$:

$$\mathbf{in}_m^t = (in_{m,1}^t, \ldots, in_{m,n}^t)$$

$$\mathbf{in}_{m,j}^t = \begin{cases} 1 & \text{if } e_j \text{ is an ingoing edge of } v_m^t, \\ -1 & \text{if } e_j \text{ is an ingoing edge of } v_{m+1}^t \\ 0 & \text{else.} \end{cases}$$

$$\mathbf{out}_m^t = (out_{m,1}^t, \ldots, out_{m,n}^t)$$

$$\mathbf{out}_{m,j}^t = \begin{cases} 1 & \text{if } e_j \text{ is an outgoing edge of } v_m^t, \\ -1 & \text{if } e_j \text{ is an outgoing edge of } v_{m+1}^t \\ 0 & \text{else.} \end{cases}$$

Clearly, $\mathbf{in}_m^t \cdot \mathbf{x}_r = \mathbf{0}$ resp. $\mathbf{out}_m^t \cdot \mathbf{x}_r = \mathbf{0}$ if and only if $In_{lpo}(v_m^t, r) = In_{lpo'}(v_{m+1}^t, r)$ resp. $Out_{lpo}(v_m^t, r) = Out_{lpo'}(v_{m+1}^t, r)$ for the LPOs $lpo = (V, <, l)$ and $lpo' = (V', <', l')$ with $v_m^t \in V$ and $v_{m+1}^t \in V'$.

Finally, to ensure that all LPOs have the same initial token flow, we denote $L = \{lpo_1, lpo_2, \ldots\}$ and add rows for $1 \leqslant m \leqslant |L| - 1$:

$$\mathbf{init}_m = (init_{m,1}, \ldots, init_{m,n})$$

$$\mathbf{init}_{m,j} = \begin{cases} 1 & \text{if } e_j \text{ is an outgoing edge of } v_0^{lpo_m}, \\ -1 & \text{if } e_j \text{ is an outgoing edge of } v_0^{lpo_{m+1}} \\ 0 & \text{else.} \end{cases}$$

Clearly, $\mathbf{init}_m \cdot \mathbf{x}_r = \mathbf{0}$ if and only if $Out_{lpo_m}(v_0^{lpo_m}, r) = Out_{lpo_{m+1}}(v_0^{lpo_{m+1}}, r)$.

Figure 9 shows the described homogenous linear equation system $\mathbf{A}_L \cdot \mathbf{x} = \mathbf{0}$ for the numbering of edges given in Figure 8. There exist two pairs of equally labeled nodes, and we need to ensure that each pair has the same intoken and outtoken flow. The first row $\mathbf{in}_1^a$ ensures for every function r given by a solution x_r that both a-labeled nodes have the same intoken flow. Therefore, the sum of the values on all ingoing edges of v_1^a (namely e_5) must equal the sum of the values on all ingoing edges of v_2^a (namely e_7 and e_9). We get the corresponding equation $x_5 - x_7 - x_9 = 0$ (this equation corresponds to the first row $\mathbf{in}_1^a$ of $\mathbf{A}_L$). Row number two $\mathbf{out}_1^a$ guarantees equal outtoken flow of the a-labeled nodes. Rows number three $\mathbf{in}_1^b$ and four $\mathbf{out}_1^b$ do the same for both nodes labeled by b. The last row of the matrix ensures that both LPOs have the same initial token flow. A possible non-negative integer solution is

$$
\begin{array}{r}
in_1^a \\ out_1^a \\ in_1^b \\ out_1^b \\ init_1
\end{array}
\begin{bmatrix}
0 & 0 & 0 & 0 & 1 & 0 & -1 & 0 & -1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & -1 \\
0 & 1 & 0 & 0 & 0 & -1 & 0 & -1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 \\
1 & 1 & 0 & -1 & -1 & -1 & -1 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}
\bullet
\begin{bmatrix}
① \\ ② \\ ③ \\ ④ \\ ⑤ \\ ⑥ \\ ⑦ \\ ⑧ \\ ⑨ \\ ⑩ \\ ⑪ \\ ⑫
\end{bmatrix}
=
\begin{bmatrix}
0 \\ 0 \\ 0 \\ 0 \\ 0
\end{bmatrix}
$$

(Columns labelled ① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩ ⑪ ⑫.)

Figure 9. Equation system defining token flow regions.

$\mathbf{x}_r = (0, 1, 0, 0, 1, 0, 0, 1, 1, 0, 0, 2)$ corresponding to the token flow region drawn in Figure 7 (and to the middle place shown in Figure 3).

By the above considerations the set of token flow regions r is in one-to-one-correspondence to the set of non-negative integer solutions $\mathbf{x} = (x_1, \ldots, x_n)$ of $\mathbf{A}_L \cdot \mathbf{x} = \mathbf{0}$ via $r(e_i) = x_i$. This means, every feasible place can be computed by such a solution. The place corresponding to a solution $\mathbf{x}$ we denote by $p_\mathbf{x}$. Note that the number of rows of $\mathbf{A}_L$ linearly depends on the number of nodes $|W_L|$. $\mathbf{A}_L$ contains at most $|W_L| - 1$ $\mathbf{in}$-rows, at most $|W_L| - 1$ $\mathbf{out}$-rows and exactly $|L| - 1 \leq |W_L| - 1$ $\mathbf{init}$-rows. Exactly $|W_L| - 1$ $\mathbf{in}$-rows resp. $\mathbf{out}$-rows occur in the case that all nodes in W_L carry the same label. The number of columns of $\mathbf{A}_l$ is equal to the number of edges $|E_L^\star| = 2|W_L| + |E_L|$.

3.2.2. Finite Representation

The homogenous linear equation system developed in the last subsection is in fact an inequation system, since we search for non-negative solutions, i.e. we require $\mathbf{x} \geqslant 0$ for solutions $\mathbf{x}$. Thus we compute token flow regions of a finite partial language $\mathcal{L}$ and subsequently places of the searched saturated feasible p/t-net by solving the finite homogenous linear inequation system $\mathbf{A}_L \cdot \mathbf{x} \leqslant \mathbf{0}$, $-\mathbf{A}_L \cdot \mathbf{x} \leqslant \mathbf{0}$, $-\mathbf{x} \leqslant \mathbf{0}$ with $n + 2N$ rows (N is the number of rows, n the number of columns of $\mathbf{A}_L$). The set of solutions of such a system is called a *polyhedral cone*. According to a theorem of Minkowski [24] polyhedral cones are finitely generated, i.e. there are finitely many vectors $\mathbf{y}_1, \ldots, \mathbf{y}_k$ (also called *basis solutions*) such that each element $\mathbf{x}$ of the polyhedral cone is a non-negative linear sum $\mathbf{x} = \sum_{i=1}^k \lambda_i \mathbf{y}_i$ for some $\lambda_1, \ldots, \lambda_k \geqslant 0$. In our case the cone is pointed. Then a minimal basis $\mathbf{y}_1, \ldots, \mathbf{y}_k$ is given by the rays of the cone [27]. Such basis solutions $\mathbf{y}_1, \ldots, \mathbf{y}_k$ can be effectively computed from $\mathbf{A}_L$ (see for example [25, 28]). If all entries of $\mathbf{A}_L$ are integral, then also the entries of all $\mathbf{y}_i$ can be chosen to be integral (i.e. as regions). The time complexity of the computation essentially depends on the number k of basis solution which is bounded by $k \leqslant \binom{n+2N}{n-1}$. This means, in the worst case the time complexity is exponential in the number of nodes, whereas in most practical examples of polyhedral cones there are only few basis solutions.

We finally claim that all places which do not correspond to basis solutions can be deleted from the saturated feasible p/t-net without changing its partial language of runs. Thus, the saturated feasible p/t-net has a finite representation. Consider places $p, p_1, \ldots, p_k$ of some marked p/t-net (N, m_0) and

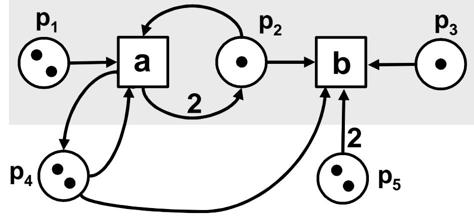

Figure 10. The p/t-net from Figure 3 extended by p_4 and p_5. p_4 and p_5 are non-negative linear combinations of p_1, p_2 and p_3.

non-negative real numbers $\lambda_1, \ldots, \lambda_k$ ($k \in \mathbb{N}^+$) such that (i) $m_0(p) = \sum_{i=1}^k \lambda_i m_0(p_i)$, (ii) $W(p,t) = \sum_{i=1}^k \lambda_i W(p_i, t)$ for all transitions t and (iii) $W(t,p) = \sum_{i=1}^k \lambda_i W(t, p_i)$ for all transitions t. In such a case we write $p = \sum_{i=1}^k \lambda_i p_i$. Figure 10 shows the p/t-net N from Figure 3 extended to a net N' by adding the two places p_4 and p_5. Neither p_4 nor p_5 restrict the behavior of N' more than $\{p_1, p_2, p_3\}$. In other words each LPO enabled in N is also enabled in N'. That is because the places p_4 and p_5 are non-negative linear combinations of the other three places. It holds $p_5 = 2p_3$ and $p_4 = \frac{1}{2}p_1 + \frac{1}{2}p_2 + \frac{1}{2}p_3$.

Lemma 3.1. Let (N, m_0), $N = (P, T, F, W)$, be a marked p/t-net with $P = \{p_1, \ldots, p_k, p\}$ and $p = \sum_{i=1}^k \lambda_i p_i$ for non-negative real numbers $\lambda_1, \ldots, \lambda_k$ ($k \in \mathbb{N}^+$). Denote $P' = \{p_1, \ldots, p_k\}$, $m_0' = m_0|_{P'}$ and $N' = (P', T, F|_{(P' \times T) \cup (T \times P')}, W|_{(P' \times T) \cup (T \times P')})$. Then each LPO enabled w.r.t. (N', m_0') is enabled w.r.t. (N, m_0).

Proof:
Let *lpo* be enabled w.r.t. (N', m_0'), *lpo* $= (V, <, l)$. According to Definition 2.5, for a cut C of *lpo* and $i \in \{1, \ldots, k\}$ it holds $m_0(p_i) + \sum_{v \in V \wedge v < C}(W(l(v), p_i) - W(p_i, l(v))) \geq \sum_{v \in C} W(p_i, l(v))$. This implies for an arbitrary cut C of *lpo* and the place p:

$$\begin{aligned} & m_0(p) + \sum_{v \in V \wedge v < C} (W(l(v), p) - W(p, l(v))) \\ = \; & \sum_{i=1}^{k} \lambda_i (m_0(p_i) + \sum_{v \in V \wedge v < C} (W(l(v), p_i) - W(p_i, l(v)))) \\ \geqslant \; & \sum_{i=1}^{k} \lambda_i \sum_{v \in C} W(p_i, l(v)) = \sum_{v \in C} W(p, l(v)). \end{aligned}$$

Thus, *lpo* is enabled w.r.t. (N, m_0). □

If $\mathbf{x} = \sum_{i=1}^k \lambda_i \mathbf{y}_i$ for integer basis solutions $\mathbf{y}_1, \ldots, \mathbf{y}_k$ of $\mathbf{A}_L \cdot \mathbf{x} = \mathbf{0}, \mathbf{x} \geq \mathbf{0}$, then $p_{\mathbf{x}} = \sum_{i=1}^k \lambda_i p_{\mathbf{y}_i}$. Thus, the finite net (N, m) having the places $p_{\mathbf{y}_1}, \ldots, p_{\mathbf{y}_k}$ satisfies $\mathfrak{Lpo}(N_{sat}, m_{sat}) = \mathfrak{Lpo}(N, m)$. In other words (N, m) generates the smallest partial language of runs including L. To compute (N, m), we compute such a finite set of integer basis solutions $\mathbf{y}_1, \ldots, \mathbf{y}_k$ (Algorithm 1). The set of regions $\{\mathbf{y}_1, \ldots, \mathbf{y}_k\}$ we call *basis representation* (of the set of all regions).

Algorithm 1 does not decide the synthesis problem (is there a net with the specified behavior?). In a further step of the synthesis procedure (Section 3.3) the net constructed with Algorithm 1 is exploited

```
1: A_L ← EmptyMatrix
2: for all t ∈ T do
3:    W_t ← {v ∈ W*_L | l*_L(v) = t}
4:    for m = 1 to |W_t| − 1 do
5:       A_L.addRow(in^t_m)
6:       A_L.addRow(out^t_m)
7:    end for
8: end for
9: for m = 1 to |L| − 1 do
10:    A_L.addRow(init_m)
11: end for
12: Solutions ← A_L.getBasisSolutions
13: (N, m) ← (∅, T, ∅, ∅, ∅)
14: for all r ∈ Solutions do
15:    (N, m).addCorrespondingPlace(r)
16: end for
17: return (N, m)
```

Algorithm 1: Calculates a net (N, m) from a partial language over T given by L, such that (N, m) generates the smallest partial language of runs including L.

to effectively decide the synthesis problem as explained in the Introduction. Nonetheless for practical applications in particular Algorithm 1 is of interest, because the main focus usually lies in the construction of a system model from a given specification (not in the decision of the synthesis problem). In this context Algorithm 1 is a useful standalone algorithm to compute a p/t-net system, which is a best upper approximation for a set of scenarios specified in terms of LPOs.

3.2.3. Implementation and Performance

We implemented Algorithm 1 as a plug-in for our framework VipTool [11]. We denote by n the number of nodes belonging to LPOs in the input L. Then the construction of the matrix $\mathbf{A}_L$ has linear runtime in n since the number of rows of $\mathbf{A}_L$ is linear in n. If k denotes the number of basis solutions of the considered cone, then constructing the synthesized net (N, m) from the set of basis solutions is linear in k. Thus, for the performance of Algorithm 1, the crucial factor is the algorithm computing the basis solutions, which, as mentioned, has exponential runtime in the worst case, i.e. $k = O(2^n)$. We implemented the algorithm of Tschernikow [28] to compute the integer basis of the cone from the integral homogeneous equation system $\mathbf{A}_L \cdot \mathbf{x} = \mathbf{0}$. Tschernikow's algorithm is an improved version of a previous algorithm to compute the basis of arbitrary homogeneous inequation systems developed by Motzkin and Burger, especially adopted for computing basis solutions for the set of non-negative integer solutions of an homogeneous equation system as it is given in our situation. Tschernikow's algorithm starts with the matrix $(\mathbf{I}, \mathbf{A}_L^T)$, consisting of the identity matrix $\mathbf{I}$ and the transpose of the matrix $\mathbf{A}_L$. This matrix is stepwise transformed. In each step a non-zero lead column corresponding to one of the equations is chosen and so called balanced pairs of rows are transformed to equilibrium rows w.r.t. the lead column. These equilibrium rows are added to the matrix. The number of added rows depends on

the number of non-zero components of the lead column. This number is at most n^2 (in the case all components are non-zero). For sparse matrices it is a lot smaller. After such transformation the previous lead column only contains zero values. The algorithm finishes when each column corresponding to an equation (the second part of the matrix) is zero, i.e. it performs at most as many steps as $\mathbf{A}_L$ has rows. The non-negative integer basis solutions of $\mathbf{A}_L$ can then directly be read out from the first part of the constructed matrix. Namely, the first part of each row (corresponding to $\mathbf{I}$) is such a basis solution. Thus, there are as many basis solutions as rows. These solutions define the set of places of (N, m) in our case. Since we have a sparse matrix (i.e. many entries of a lead column are already zero), the algorithm should perform quite well in our setting. In particular, the growth of the number of rows of the stepwise computed matrices and therefore the number of basis solutions should be very limited.

The size of the basis determining the size of the synthesized net is not only important for memory consumption, but also as the input for the equality test. There are methods to reduce this size. The synthesized net usually contains many implicit places which can be deleted by appropriate algorithms, without changing the behavior of the net. Therefore a simple and efficient procedure to delete implicit places of the synthesized net is added to the implementation of Algorithm 1. This procedure deletes places that are dominated (w.r.t. the behavioral restriction) by one another place, but combinations of places dominating other places are not searched. One place p' dominates another place p if $\lambda \cdot m_0(p) \geq m_0(p')$ and $\lambda \cdot W(t,p) \geq W(t,p')$ as well as $\lambda \cdot W(p,t) \leq W(p',t)$ for all transitions t and some $\lambda > 0$. In the implementation only $\lambda = 1$ is considered. Implicit places dominated by one another place are searched by comparing the computed places pairwise. Advanced methods to detect implicit places (considering combinations of places) still offer extensive improvement possibilities for this module of Algorithm 1. Experimental results on the performance of Algorithm 1 can be found in Section 5.

3.3. Equality Test

Up to now, we have shown how to compute from a finite set of LPOs L a finite marked p/t-net (N, m) which has the smallest partial language of runs $\mathfrak{Lpo}(N, m)$ including the specified partial language $\mathcal{L} = \{[lpo] \mid lpo \text{ is a prefix of a sequentialization of an LPO in } L\}$ (Algorithm 1). This net either solves the synthesis problem ($\mathfrak{Lpo}(N, m) = \mathcal{L}$) or there is no solution. In this section we develop two methods to test whether $\mathfrak{Lpo}(N, m) = \mathcal{L}$.

Let L_{seq}^{pref} be the set of all sequentializations of prefixes of LPOs in L. Since we already know $\mathfrak{Lpo}(N, m) \supseteq \mathcal{L}$, in order to test $\mathfrak{Lpo}(N, m) = \mathcal{L}$, we (1) either have to check if each enabled *lpo* of (N, m) is isomorphic to an LPO in L_{seq}^{pref} (optimistic equality test), or (2) to test that no LPO *lpo* which is not isomorphic to an LPO in L_{seq}^{pref} is enabled w.r.t. (N, m) (pessimistic equality test).

3.3.1. Optimistic Equality Test

In the first case (1), we calculate all enabled LPOs of (N, m). The set of (pairwise non-isomorphic) enabled LPOs of a p/t-net in general can be infinite, but $\mathfrak{Lpo}(N, m)$ is always finite. This can be proven for $\mathfrak{Lpo}(N_{sat}, m_{sat})$ ($= \mathfrak{Lpo}(N, m)$) as follows: for every transition t and every LPO $lpo = (V, <, l) \in L$ there is a finite number $n_{lpo,t}$ of nodes $v \in V$ labeled by t. Since L is finite we get a finite upper bound $n_t = max(\{n_{lpo,t} \mid lpo \in L\})$ for the maximal number of occurrences of t in an LPO $lpo \in L$. Consequently, the place p_t with the initial marking $m_0(p_t) = n_t$, an empty pre-set and t as the only

transition in its post-set with $W(p_t,t) = 1$ is feasible w.r.t. $\mathcal{L}$. This means, that each transition t can maximally occur n_t-times, and thus every LPO in $\mathfrak{Lpo}(N_{sat}, m_{sat})$ has at most $\sum_{t\in T} n_t$ nodes.

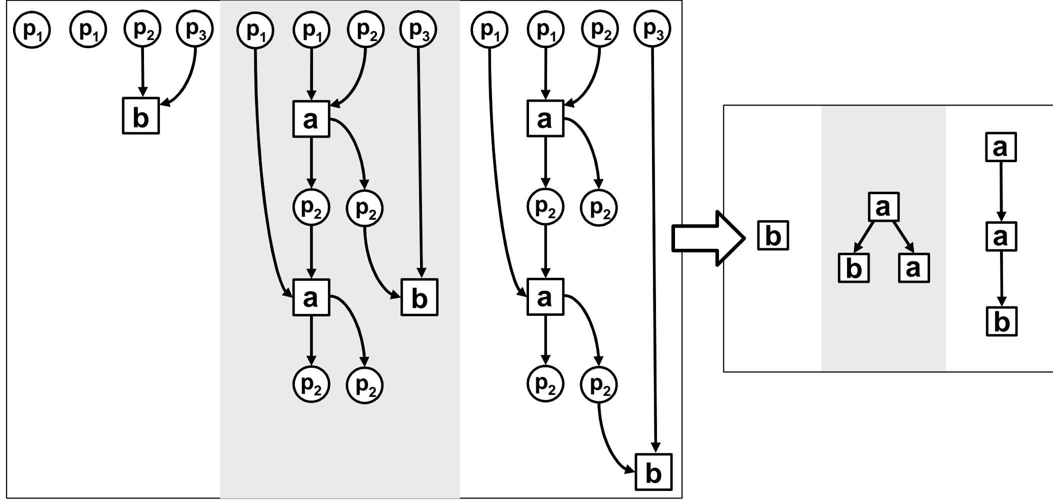

Figure 11. Maximal process nets of the p/t-net shown in Figure 3 and corresponding runs.

Since $\mathfrak{Lpo}(N,m)$ is finite, it can be calculated. In principle, we have to check if each run $lpo \in \mathfrak{Lpo}(N,m)$ is isomorphic to an LPO in L^{pref}_{seq}. But for a run lpo', which is a sequentialization of a prefix of another run lpo, it is enough to consider only lpo, because if lpo' is not isomorphic to an LPO in L^{pref}_{seq}, then the same holds for lpo. Therefore, we only have to regard runs which are not sequentializations of prefixes of other runs. The set of all such runs can be computed through the (finite) set of process nets with maximal length [30]: omitting conditions in a process net and only keeping the ordering between events yields an LPO, and it is well known that each such LPO underlying a process net is a run. Moreover, each run is a sequentialization of a prefix of an LPO underlying a process net with maximal length. Thus, it is enough to regard the LPOs underlying such process nets of (N,m). The synthesis problem has a solution if and only if each such LPO is isomorphic to some LPO in L^{pref}_{seq} (Algorithm 2). For example, the maximal process nets and the underlying LPOs of the p/t-net shown in Figure 3 are depicted in Figure 11. The first two LPOs are isomorphic to the two LPOs (of the running example) shown in Figure 2 and the third one is isomorphic to a sequentialization of the second LPO in Figure 2.

```
1: Process ← (N, m).getAllMaxProcesses
2: for all pro ∈ Process do
3:     if L^pref_seq.notContainsIso(pro.getLPO) then
4:         return false
5:     end if
6: end for
7: return true
```

Algorithm 2: Optimistic equality test: tests if $\mathfrak{Lpo}(N,m) = \mathcal{L}$ (indicated by a boolean variable).

An algorithm that calculates the set of maximal process nets of a p/t-net is implemented in our tool VipTool [11]. In general, the number of process nets is exponential in the size of the p/t-net, and the calculation of the process nets requires an exponential runtime in the worst case. Recently, we studied several new unfolding algorithms, which can be used to compute the searched set of LPOs underlying maximal process nets [6]. Compared to classical unfolding algorithms, we developed two very fast and less memory consuming algorithms for this purpose, called Method 1 and Method 2. In [6] we demonstrated their superior performance. In contrast to classical unfolding algorithms the two new algorithms abstract from the individuality of tokens using token flows (as presented in this paper). Instead of occurrence nets we use the concept of prime event structures. One of the unfolding models avoids to represent isomorphic processes while the other additionally reduces the number of (possibly non-isomorphic) processes with isomorphic underlying runs. Both proposed unfolding models still represent the complete partial order behavior in the sense that all LPOs underlying process nets are computed. In our special situation we expect that the number of process nets of (N, m) roughly coincides with the size of L, because in the case that there is a positive solution of the synthesis problem there holds $\mathfrak{Lpo}(N, m) = \mathcal{L}$, and in the negative case $\mathfrak{Lpo}(N, m)$ is the best upper approximation to $\mathcal{L}$.

It can directly be tested if a computed LPO underlying a maximal process net (constructed by some unfolding procedure) is isomorphic to a sequentialization of one of the specified LPOs in L. This test is a special graph isomorphism problem. Graph isomorphism problems are widely believed to form an own complexity class between P and NP. The common procedure to solve graph isomorphism problems is applying backtracking algorithms constructing a set of possible isomorphisms, and then checking each possible isomorphism, if it actually is an isomorphism. The efficiency of the procedure depends on restricting the set of possible isomorphisms in the backtracking algorithm as good as possible. We consider the following very restrictive strategy for eliminating possible isomorphisms in the backtracking procedure: we identify appropriate equivalence classes of events in the two considered LPOs, such that by an isomorphism events of the first LPO can only be mapped onto events of the second LPO being in the same equivalence class. These equivalence classes account for the label of an event as well as the labels of all events in the pre- and post-set of the event. Postulating coincidence of these labels is very restrictive in our case because of the transitivity of LPOs. Thus this strategy ensures an efficient isomorphism test. Actually the implemented algorithm (called Isotest 1) does not perform an isomorphism test between all sequentializations of LPOs in L and computed LPOs, but only tries to embed the computed LPOs as sequentializations of LPOs in L regarding isomorphism (using the principles described above), i.e. not all sequentializations of LPOs in L have to be computed. Finding an appropriate isomorphism is only problematic concerning runtime in cases of auto-concurrency (equally labeled concurrent events). In practical examples auto-concurrency often does not occur (for example in Mazurkiewicz traces).

We implemented Algorithm 2 as described in this subsection in our framework VipTool (as a plug-in). We applied the unfolding procedure called Method 1 in [6] to compute the set of runs associated to maximal process nets of the synthesized net (N, m) and the isomorphism test Isotest 1. Experimental results for Algorithm 2 are shown in Section 5.

3.3.2. Pessimistic Equality Test

The alternative possibility (2) to test $\mathfrak{Lpo}(N, m) = \mathcal{L}$ is to check, if no LPO *lpo* not isomorphic to some LPO in L_{seq}^{pref} is in $\mathfrak{Lpo}(N, m)$. For one such LPO *lpo* this can be tested in polynomial time in the number of nodes of *lpo* using the algorithm we presented in [19]. The problem is that there

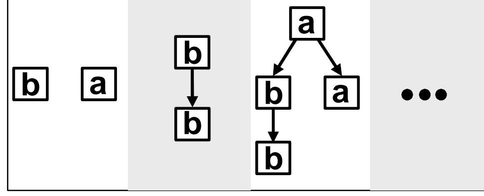

Figure 12. Some LPOs in L^c_{fin}.

are infinitely many such LPOs. Therefore, we define a finite set L^c_{fin} of LPOs representing the set of all LPOs L^c not specified by L in the following sense: if no LPO in L^c_{fin} is enabled in (N, m) then also no LPO in L^c is enabled in (N, m). The idea for the construction of L^c_{fin} is to extend each $lpo \in L^{pref}_{seq}$ in all possible ways by one event, such that the resulting LPO lpo' is not isomorphic to an LPO in L^{pref}_{seq}. This means, L^c_{fin} consists of all LPOs lpo' not isomorphic to an LPO in L^{pref}_{seq} defined by $lpo' = (V \cup \{v_t\}, < \cup <_t, l \cup (v_t, t))$, where $(V, <, l) \in L^{pref}_{seq}$, $t \in T$, $v_t \notin V$ and $<_t = \{v' \mid v' \in V' \vee v' < V'\} \times \{t\}$ for a co-set V' of $(V, <, l)$ (V' may be empty, which means that v_t becomes an additional minimal event). Figure 12 shows some of the LPOs in L^c_{fin} for the set of LPOs L shown in Figure 2. The most left LPO is constructed by appending an a-labeled event v_a to the empty co-set V' of $lpo1$ from Figure 2.

If there exists an LPO $lpo' \in L^c_{fin}$ which is enabled in (N, m), then obviously $\mathcal{L} \neq \mathfrak{Lpo}(N, m)$. On the other hand, if every such LPO lpo' is not enabled in (N, m), we conclude that $\mathcal{L} = \mathfrak{Lpo}(N, m)$ (Algorithm 3). This can be proven as follows by contradiction: assume that every LPO $lpo' \in L^c_{fin}$ is not enabled in (N, m), but there exists an $lpo \in L^c$ which is enabled in (N, m). Then, there is a maximal prefix lpo_{pre} of lpo (possibly empty) isomorphic to an LPO in L^{pref}_{seq}. Let lpo'_{pre} be a further prefix of lpo having one additional node (such lpo'_{pre} exists because lpo is not isomorphic to an LPO in L^{pref}_{seq}). The maximality of lpo_{pre} implies that lpo'_{pre} is not isomorphic to an LPO in L^{pref}_{seq}. By construction of L^c_{fin}, we conclude that lpo'_{pre} is isomorphic to an LPO in L^c_{fin}. Since lpo'_{pre} is a prefix of an LPO enabled in (N, m), it is also enabled in (N, m). This is a contradiction.

```
1: for all lpo' ∈ L^c_fin do
2:    if Lpo(N, m).contains(lpo') then
3:       return false
4:    end if
5: end for
6: return true
```

Algorithm 3: Pessimistic equality test: tests if $\mathfrak{Lpo}(N, m) = \mathcal{L}$ (indicated by a boolean variable).

The complexity of the pessimistic equality test (Algorithm 3) in particular depends on the size of the set L^c_{fin}. Each LPO in L^c_{fin} has to be tested if it is in $\mathfrak{Lpo}(N, m)$. As mentioned, this test can be performed in polynomial time in the number of nodes of the LPO using methods from [19]. Since the set L^c_{fin} can have exponential many LPOs in the size of L, the runtime is exponential in the worst case. We assume that a significantly smaller subset of L^c_{fin} suffices to represent the set of all LPOs L^c, and we

are currently working on methods to reduce the set L^c_{fin}. We did not implement Algorithm 3 so far.

4. Alternative Synthesis Approaches

It is also possible to indirectly derive a synthesis algorithm for p/t-nets from a partial language using "classical" region definitions. A partial language given by L defines a language S_L of step sequences given by the set of all step linearizations of LPOs in L. Following ideas in [17], where regions of trace languages are defined, it is possible to define regions of languages of step sequences. It turns out that each place defined by a token flow region of L (as defined in this paper) is also defined by a region of S_L and vice versa.[3] Thus, in our terminology the set of regions of S_L corresponds to the set of feasible places of L (compare Theorem 3.2). Consequently we can apply the region definition for p/t-nets and languages of step sequences to deduce a synthesis algorithm for the setting of p/t-nets and finite partial languages. Here the same problems appear as in the case of token flow regions, in particular there are also infinitely many regions. Therefore, it is necessary to compute a finite representation of the set of all regions. Such a finite representation is developed in [3] for regions of step transition systems and in [9] for regions of regular languages. The finite representations developed there we call *separation representations* in this paper. These have a fundamental different intuition to *basis representations*. Combining the ideas from [3] and [9], separation representations can be adopted for languages of step sequences.

Altogether, there is a second approach to synthesize a net from a partial language given by L through computing a separation representation of the set of classical regions of the set of step sequences S_L. This approach is presented in the next subsections in detail together with a short comparison of the performance of the two synthesis algorithms presented in this paper. Moreover, there are further variants of synthesis algorithms: it is observed in [23] that the two kinds of regions can be each combined with the two kinds of finite representations. These variants we discuss only very briefly, because it turns out that they do not promise a better performance.

4.1. Synthesis Based on a Separation Representation of Classical Regions

A (classical) region of a language of step sequences L' is simply a tuple of natural numbers which represents the initial marking of a place and the number of tokens each transition consumes respectively produces in that place, satisfying some property which ensures that no step sequence of the given language L' is prohibited by this place. The set of regions of L' defines the set of feasible places of L'.

Definition 4.1. (Region)
Denoting $T = \{t_1, \ldots, t_m\}$ the transitions occurring in L', a *region* of L' is a tuple $\mathbf{r} = (r_0, \ldots, r_{2m}) \in \mathbb{N}^{2m+1}$ satisfying for every $\sigma = \tau_1 \ldots \tau_n \in L'$ and every $j \in \{1, \ldots, n\}$:

$$(*) \qquad r_0 + \sum_{i=1}^{m}((\tau_1 + \ldots + \tau_{j-1})(t_i) \cdot r_i - (\tau_1 + \ldots + \tau_j)(t_i) \cdot r_{m+i}) \geq 0.$$

Every region $\mathbf{r}$ of L' defines a place p_r via $m_0(p_r) := r_0$, $W(t_i, p_r) := r_i$ and $W(p_r, t_i) := r_{m+i}$ for $1 \leqslant i \leqslant m$. The place p_r is called the *corresponding place to* $\mathbf{r}$.

[3] It was already shown in [22] that the region definition for trace languages (and p/t-nets) from [17] is consistent to the token flow region definition for partial languages of this paper.

The set of regions of L' can be characterized as the set of non-negative integral solutions of a homogenous linear inequation system

$$\mathbf{A}_{L'} \cdot \mathbf{r} \geq \mathbf{0}.$$

The matrix $\mathbf{A}_{L'}$ consists of rows $\mathbf{a}_\sigma^j = (a_{\sigma,0}^j, \ldots, a_{\sigma,2m}^j)$ for all $\sigma = \tau_1 \ldots \tau_n \in L'$, $j \in \{1, \ldots, n\}$, satisfying $\mathbf{a}_\sigma^j \cdot \mathbf{r} \geq \mathbf{0} \Leftrightarrow (*)$. This is achieved by setting:

$$a_{\sigma,i}^j = \begin{cases} 1 & \text{for } i = 0, \\ (\tau_1 + \ldots + \tau_{j-1})(t_i) & \text{for } i = 1, \ldots, m \\ -(\tau_1 + \ldots + \tau_j)(t_{i-m}) & \text{for } i = m+1, \ldots, 2m. \end{cases}$$

The ideas in [3, 9] to get an effective synthesis algorithm is to prohibit non-specified behavior by places corresponding to regions. In our setting we search for a region for each step sequence not specified in L' such that the corresponding place guarantees that this step sequence is not enabled. If $\tau_1 \ldots \tau_n$ is not enabled then also $\tau_1 \ldots \tau_n \tau_{n+1}$ and $\tau_1 \ldots \tau_n'$ with $\tau_n \leq \tau_n'$ are not enabled. Moreover, we assume that L' is step linearization closed, because this is the case for S_L. Therefore, we only have to consider certain step sequences not in L' which we call *wrong continuations*:

Definition 4.2. (Wrong continuation)
Denote $L'_{pref} = \{\tau_1 \ldots \tau_j \mid \tau_1 \ldots \tau_n \in L' \wedge j \in \{1, \ldots, n\}\} \cup \{\epsilon\}$ (ϵ is the empty step sequence) the set of all prefixes of L'. The set of wrong continuations of L' is defined by $L'_{wrong} = \{\tau_1 \ldots \tau_{n-1}(\tau_n + t_i) \notin L'_{pref} \mid \tau_1 \ldots \tau_n \in L'_{pref},\ i \in \{1, \ldots, m\}\} \cup \{\tau_1 \ldots \tau_n t_i \notin L'_{pref} \mid \tau_1 \ldots \tau_n \in L'_{pref},\ i \in \{1, \ldots, m\}\}$.

In order to compute a feasible place which prohibits a wrong continuation $\sigma' = \tau_1' \ldots \tau_n'$ of L', one defines so called *separating regions* defining such places:

Definition 4.3. (Separating region)
Let $\sigma' = \tau_1' \ldots \tau_n'$ be a wrong continuation. A region $\mathbf{r}$ of L' is a *separating region w.r.t.* σ' if

$$(**) \qquad r_0 + \sum_{i=1}^{m} ((\tau_1' + \ldots + \tau_{n-1}')(t_i) \cdot r_i - (\tau_1' + \ldots + \tau_n')(t_i) \cdot r_{m+i}) < 0.$$

A separating region $\mathbf{r}$ w.r.t. σ' can be calculated (if it exists) as a non-negative integer solution of a homogenous linear inequation system with integer coefficients of the form

$$\begin{aligned} \mathbf{A}_{L'} \cdot \mathbf{r} &\geq \mathbf{0} \\ \mathbf{b}_{\sigma'} \cdot \mathbf{r} &< \mathbf{0}. \end{aligned}$$

The vector $\mathbf{b}_{\sigma'} = (b_{\sigma',0}, \ldots, b_{\sigma',2n})$ is defined in such a way that $\mathbf{b}_{\sigma'} \cdot \mathbf{r} < \mathbf{0} \Leftrightarrow (**)$. This is achieved by setting

$$b_{\sigma',i} = \begin{cases} 1 & \text{for } i = 0, \\ (\tau_1' + \ldots + \tau_{n-1}')(t_i) & \text{for } i = 1, \ldots, m \\ -(\tau_1' + \ldots + \tau_n')(t_{i-m}) & \text{for } i = m+1, \ldots, 2m. \end{cases}$$

If there exists no non-negative integer solution of the system $\mathbf{A}_{L'} \cdot \mathbf{r} \geq \mathbf{0}, \mathbf{b}_{\sigma'} \cdot \mathbf{r} < \mathbf{0}$, there exists no separating region w.r.t. σ' and thus no feasible place prohibiting σ'. If there exists a non-negative

integer solution of the system, any such solution defines a feasible place prohibiting σ'. If we choose one arbitrary separating region $\mathbf{r}_{\sigma'}$ for each wrong continuation σ' for which such a region exist, then we call the set $\{\mathbf{r} \mid \exists \sigma' : \mathbf{r} = \mathbf{r}_{\sigma'}\}$ a *separation representation* (of the set of all regions). A place corresponding to each separating region of the separation representation is added to the synthesized net (N, m). It is easy to prove that there is a p/t-net having L'_{pref} as its set of step occurrence sequences if and only if there is a separating region for each wrong continuation in L'_{wrong}. In the positive case (N, m) is such net.

Since we are interested in solving the synthesis problem for a partial language given by L, we now shift to this setting. As mentioned above, L defines a language S_L of step sequences as follows:

Definition 4.4. Let L define a partial language. The language of step sequences defined by L is given by $S_L = \{\sigma_{lpo'} \mid \exists lpo \in L : lpo' \text{ is a step linearization of } lpo\}$.

As mentioned, the definitions of token flow regions of L and regions of S_L are consistent.

Theorem 4.1.

(i) Let r be a token flow region of L and p_r be the corresponding place. Then there is a region r' of S_L with $p_{r'} = p_r$.

(ii) Let r' be a region of S_L and $p_{r'}$ be the corresponding place. Then there is a token flow region r of L with $p_r = p_{r'}$.

Proof:
ad (i): Define $r'_0 = m_0(p_r), \forall 1 \leqslant i \leqslant m : \ r'_i = W(t_i, p_r) \wedge r'_{i+m} = W(p_r, t_i)$ and $r' = (r'_0, \ldots, r'_{2m})$. Since r is a token flow region of L, p_r is feasible w.r.t. L (Theorem 3.2). Consider the net N_{p_r} associated to p_r. Since p_r is feasible, all LPOs in L are enabled w.r.t. N_{p_r}. From the definition of enabled LPOs we deduce that each step sequence corresponding to a step linearization of an LPO in L, i.e. each step sequence in S_L, is enabled to occur in N_{p_r}. This means that r' satisfies $(*)$, i.e. r' is a region of S_L. By definition of r', $p_r = p_{r'}$.

ad (ii): Let r' be a region of S_L, $p_{r'}$ be the corresponding place and $N_{p_{r'}}$ be the associated net. According to $(*)$, each step sequence in S_L is enabled in $N_{p_{r'}}$. This means, each LPO in L is enabled w.r.t. $N_{p_{r'}}$, i.e. $p_{r'}$ is feasible w.r.t. L. We deduce that there is a token flow region r of L defining $p_{r'}$ (Theorem 3.2). $\square$

Now we can solve the synthesis problem for a partial language given by L by solving the synthesis problem as shown above for the language $L' = S_L$. If we denote the synthesized net by (N, m), we get:

Lemma 4.1. There is a solution of the synthesis problem for the partial language $\mathcal{L}$ (defined by L) if and only if $\mathfrak{Lpo}(N, m) = \mathcal{L}$.

Proof:
It is only necessary to prove the *only if*-part. Assume there is a solution (N', m') of the synthesis problem for the partial language $\mathcal{L}$ (defined by L) and $\mathfrak{Lpo}(N, m) \neq \mathcal{L}$. This implies $\mathfrak{Lpo}(N, m) \supsetneq \mathcal{L}$, because we know $\mathfrak{Lpo}(N, m) \supseteq \mathcal{L}$ from the fact that the set of regions of L' is consistent with the set of token flow regions of L (Theorems 3.1 and 4.1).

We can distinguish two cases: either the set of enabled step occurrence sequences of (N, m) coincides with L'_{pref} or not. If the set of enabled step occurrence sequences of (N, m) does not coincide with L'_{pref}, then for some wrong continuation $\sigma' \in L'_{wrong}$ there does not exist a separating region. In this case σ' corresponds to an LPO not specified by L, which is enabled in (N', m'). Otherwise, (N', m') would have a place prohibiting σ', and this place would correspond to a separating region. This is a contradiction.

Let the set of enabled step occurrence sequences of (N, m) coincide with L'_{pref} and let $lpo \notin L_{seq}^{pref}$ be enabled in (N, m). This means each step sequence $\sigma_{lpo'}$ associated to a step linearization lpo' of lpo is enabled in (N, m). Since the set of enabled step occurrence sequences of (N, m) coincides with L'_{pref}, we conclude $\sigma_{lpo'} \in L'_{pref}$. Thus all step linearizations lpo' of lpo are also enabled in (N', m'). Therefore also $lpo \notin L_{seq}^{pref}$ is enabled in (N', m') and thus $\mathfrak{Lpo}(N', m') \neq \mathcal{L}$. This is a contradiction.

$\square$

For the partial language $\mathcal{L}$ of the running example (Figure 2), Figure 13 shows on the left side the language L'_{pref} and one wrong continuation σ' of L' in grey. On the right the corresponding inequations leading to the inequation system $\mathbf{A}_{L'} \cdot \mathbf{r} \geq \mathbf{0}$ and $-\mathbf{b}_{\sigma'} \cdot \mathbf{r} \geq \mathbf{1}$ (in grey) are shown. The set of separating regions of $\mathcal{L}$ w.r.t. σ' is given by the set of non-negative integer solutions of this inequation system.

	r_0	r_1	r_2	r_3	r_4	$\geq$
a	1	0	0	-1	0	0
b	1	0	0	0	-1	0
aa	1	1	0	-2	0	0
ab	1	1	0	-1	-1	0
a(a+b)	1	1	0	-2	-1	0
aab	1	2	0	-2	-1	0
aba	1	1	1	-2	-1	0
a(a+b)a	-1	-2	-1	3	-1	1

Figure 13. Equation system defining separating regions of the partial language of the running example w.r.t. the wrong continuation shown in the last (grey) row ($t_1 = a, t_2 = b$).

Altogether, the synthesis algorithm for a partial language given by L is as follows: successively each wrong continuation $\sigma' \in L'_{wrong}$ is considered. If σ' is enabled in the net constructed so far, then an attempt is made to solve the respective inequation system $\mathbf{A}_{L'} \cdot \mathbf{r} \geq \mathbf{0}$, $\mathbf{r} \geq \mathbf{0}$, $\mathbf{b}_{\sigma'} \cdot \mathbf{r} < \mathbf{0}$. If there is no solution, the synthesis problem has a negative answer. Otherwise one solution is computed and the corresponding place is added to the net. In order to decide the solvability of the inequation system and to compte a solution in the positive case several linear programming solvers, such as the Simplex method, the method by Khachyan or the method of Karmarkar, can be applied.[4] The methods of Khachiyan (ellipsoid method) and Karmarkar (interior point method) need only polynomial runtime [27]. Nevertheless usually a better choice is the classical Simplex algorithm or variants of the Simplex algorithm [29]. While the Simplex algorithm is exponential in the worst case, probabilistic and experimental results [27] show that the Simplex algorithm has a significantly faster average runtime than

[4]Since the considered inequation systems are homogenous, we can apply solvers searching for rational solutions, because each rational solution of the system can be transformed to an integer solution by multiplying with the common denominator of the components of the solution vector.

for example the algorithm of Khachiyan. Having processed all wrong continuations, the first part of the synthesis algorithm is finished and the synthesized net (N, m) is constructed (Algorithm 4).

```
1:  A_{L'} ← EmptyMatrix
2:  (P, T, F, W, m_0) ← (∅, T, ∅, ∅, ∅)
3:  solvable ← true
4:  for all τ_1 ... τ_n ∈ L' do
5:     for j = 1 to n do
6:        A_{L'}.addRow(a_{τ_1...τ_j})
7:     end for
8:  end for
9:  for all σ' ∈ L'_{wrong} do
10:    if isStepOccurrenceSequence(σ', (P, T, F, W, m_0)) then
11:       r ← Solver.getIntegerSolution(A_{L'} · r ≥ 0, r ≥ 0, b_{σ'} · r < 0)
12:       if r ≠ null then
13:          P.add(correspondingPlace(r))
14:       else
15:          solvable ← false
16:       end if
17:    end if
18: end for
19: return [(P, T, F, W, m_0), solvable]
```

Algorithm 4: Computes (N, m) and the answer to the synthesis problem of L' from a partial language given by L.

The disadvantage of Algorithm 4 is that the number of step linearizations of an LPO may be exponential in the number of nodes of the LPO, i.e. the size of L' may be exponential in the size of L. Thus the considered inequation systems (i.e. the matrix $\mathbf{A}_{L'}$) may have exponential many rows in the size of the input L. Moreover, the number of wrong continuations $|L'_{wrong}|$ is exponential in the size of L (since the number of wrong continuations $|L'_{wrong}|$ may in the worst case be $2|T||L'_{pref}|$). Therefore, the number of inequations that have to be solved (bounded by $|L'_{wrong}|$) as well as the number of places (defined by solutions) in the constructed net may be exponential in the input L. But usually the number of places is much smaller, because one separating region typically prohibits many wrong continuations at once.

We implemented Algorithm 4 as a plug-in into our framework VipTool. As the linear programming solver we implemented the standard procedure to calculate a starting edge with the Simplex algorithm. This is a natural and in practical applications very efficient approach to decide, if there is a non-negative integer solution of the considered linear inequation systems and to find such solution in the positive case. Note that in the whole approach instead of choosing $L' = S_L$ we could only add to L' step sequences of S_L having maximal concurrency (the definition of L'_{wrong} has to be adapted in this case). As an alternative, the algorithm can also be optimized by searching for identical vectors $a_{\tau_1 \ldots \tau_j}$ and vectors $a_{\tau_1 \ldots \tau_j}$ defining less restrictive inequations than other vectors $a_{\tau'_1 \ldots \tau'_j}$ to delete respective inequations from $\mathbf{A}_{L'}$. This idea is realized in our implementation of Algorithm 4. Omitting such inequations significantly reduces the size of the considered inequation systems. Searching for identical or less restrictive vectors $b_{\sigma'}$

can also reduce the number of wrong continuations σ' that have to be considered. But this is not necessary, because the "**if** $isStepOccurrenceSequence(\sigma', (P, T, F, W, m_0))$"-test in Algorithm 4 very efficiently deals with such wrong continuations. Experimental results for this synthesis algorithm are shown in Section 5.

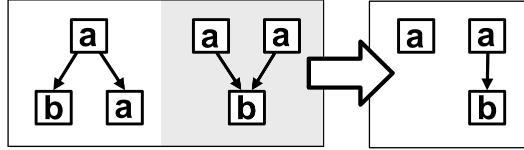

Figure 14. Left: a partial language given by L. Right: an LPO $lpo \notin L^{pref}_{seq}$ with $S_{\{lpo\}} = \{a(a+b), (a+a)b, aab, aba\} \subseteq S_L$.

The considerations in the last proof (Lemma 4.1) show that the synthesis problem for L has a negative answer, if it has a negative answer for $L' = S_L$. If it has a positive answer for L', $\mathfrak{Lpo}(N, m) = \mathcal{L}$ is still not ensured, because there still may exist LPOs $lpo \notin L^{pref}_{seq}$ fulfilling $\sigma_{lpo'} \in L'_{pref}$ for every step linearization lpo' of lpo. Such lpo is enabled in (N, m), but it is not specified in $\mathcal{L}$, i.e. the synthesis problem has a negative answer in this case. Figure 14 shows an example of a partial language given by L and such an LPO lpo. It can be tested similarly as in Algorithm 3 if such LPOs exist (see Algorithm 5). The performance of Algorithm 5 is similar to the performance of Algorithm 3. Only the testing for enabledness of LPOs in L^c_{fin} is replaced by a set inclusion test (or more precisely an isomorphism test) making Algorithm 5 independent from the net synthesized with Algorithm 4. We did not implement Algorithm 5 so far. Instead of using Algorithm 5, $\mathfrak{Lpo}(N, m) = \mathcal{L}$ can also be checked with Algorithm 2 or Algorithm 3 (because $\mathfrak{Lpo}(N, m) \supseteq \mathcal{L}$).

1: **for all** $lpo \in L^c_{fin}$ **do**
2: $\quad solvable \leftarrow false$
3: $\quad$ **for all** lpo' step linearization of lpo **do**
4: $\quad\quad$ **if** $\sigma_{lpo'} \notin L'$ **then**
5: $\quad\quad\quad solvable \leftarrow true$
6: $\quad\quad$ **end if**
7: $\quad$ **end for**
8: $\quad$ **if** $!solvable$ **then**
9: $\quad\quad$ **return** $false$
10: $\quad$ **end if**
11: **end for**
12: **return** $true$

Algorithm 5: Language based equality test: tests if $\mathfrak{Lpo}(N, m) = \mathcal{L}$ (indicated by a boolean variable).

4.2. Performance Comparison

It is not easy to compare the performance of Algorithm 1 and Algorithm 4 on the theoretical level for two reasons.

First, the region definitions applied in the two algorithms are of a fundamentally different form. This is observed in [23], where the regions applied in Algorithm 1 are called *token flow regions* (as in this paper) and the classical regions applied in Algorithm 4 are called *transition regions*. On the one hand, token flow regions specify the token flow between transition occurrences. They are determined by a variable for each possible token flow between two transition occurrences. Thus there are $O(n^2)$ such variables, where n is the number of events in L. As presented, token flow regions can be represented as solutions of a linear inequation system over these variables. The number of inequations linearly depends on n. On the other hand, transition regions directly specify the initial marking and the weights on the arcs connecting places with transitions. Thus the number of variables in this case is independent from n and $|L'|$. As presented, also transition regions can be represented as solutions of an appropriate linear inequation system. The number of inequations linearly depends on $|L'_{pref}| = O(2^n)$.

Second, in Algorithm 1 a finite representation of the infinite set of regions (feasible places) is computed in another way than in Algorithm 4. In [23] we call the finite representation applied in Algorithm 1 a *basis representation*, whereas we denote the classical finite representation applied in Algorithm 4 as *separation representation* (as in this paper). The basis representation consists of a finite basis of the set of all solutions of the considered inequation system. There can be exponential many basis solutions in the number of inequations resp. in n. As argued, from a basis representation we cannot directly deduce whether the computed net is a solution. This is because the net may still generate behavior additional to L. With a separation representation one tries to prohibit such in addition behavior. For this, one considers wrong continuations of step sequences in L'. A separation representation contains for each wrong continuation a place (region) prohibiting this wrong continuation, if such a place exists (each such place is a solution of a different inequation system which can be computed in polynomial time). Usually not all wrong continuations must be considered since some places prohibit more than one wrong continuation. This leads to the effect that typically the synthesized net has less places than using the basis representation approach. A further advantage of a separation representation is that an equality test in some cases is not necessary (when Algorithm 4 already gives a negative answer). The disadvantage of the separation representation is that the separation representation not necessarily minimizes the behavior of the constructed net. Algorithm 4 does not necessarily construct a net with minimal partial order behavior including L. In the case that Algorithm 4 already gives a negative answer to the synthesis problem, the constructed net does not even guarantee minimal step sequence behavior including L'.

Altogether, in both Algorithms the time bound is $O(2^n)$. For the computation of a basis representation of token flow regions of L, we compute a basis of potentially exponential size of an inequation system with $O(n)$ inequations and $O(n^2)$ variables. The equality test also needs in the worst case $O(2^n)$ time. For the computation of a separation representation of transition regions of L', we compute $O(2^n)$ times a solution of some inequation system with $O(2^n)$ inequations and a constant number of variables. In Section 5 we present experimental results comparing these two methods.

4.3. Variants

As observed in [23], the two kinds of regions can be each combined with the two kinds of finite representations. This leads to further variants of the two synthesis methods considered in the last subsection.

(A) It is possible to compute a separation representation of token flow regions of L. If we assume that L is given by LPOs with maximal length representing minimal causality, then a wrong continuation

is an LPO which extends a prefix of a sequentialization of an LPO in L by exactly one transition occurrence [23]. Consequently there are $O(2^n)$ wrong continuations. This is probably less efficient than computing the basis representation in average case since the basis representation is often very small in practice.

(B) It is possible to compute a basis representation of transition regions of L. Such regions are determined by $O(2^n)$ inequations [23]. Since the basis representation can be exponential in the number of inequations, this method is less efficient than using token flow regions.

(C) It is possible to compute a separation representation of transition regions of L. In this case one computes $O(2^n)$ times a solution of some inequation system with $O(2^n)$ inequations and a constant number of variables similar as when computing a separation representation of transition regions of L'.

(D) It is possible to compute a basis representation of transition regions of L'. This has a similar performance as (B).

(E) It is possible to compute a basis representation of token flow regions of L'. Such regions can be defined on LPOs lpo associated to step sequences $\sigma_{lpo} \in L'$. This involves more variables and inequations as in the case of a basis representation of token flow regions of L.

(F) It is possible to compute a separation representation of token flow regions of L'. This involves more variables and inequations as in the case of a separation representation of token flow regions of L (case (A)).

Since none of these variants promises a better performance, we decided to present experimental results only for computing the basis representation of token flow regions of L and the separation representation of transition regions of L' (i.e. for the two presented synthesis approaches Algorithm 1 and Algorithm 4).

5. Experimental Results

We implemented Algorithm 1, Algorithm 2 and Algorithm 4 and integrated them in a beta version of our framework VipTool [11]. Thus we can use Algorithm 1 or 4 to synthesize a p/t-net (N, m) from a partial language L, where in each case (N, m) is the only candidate to solve the synthesis problem. Algorithm 2 can be used to test if (N, m) actually solves the synthesis problem, i.e. in this way we get an answer to the synthesis problem in the setting of p/t-nets and partial languages.

Figure 15 shows a screenshot of VipTool. On the left side the two LPOs defining the partial language $\mathcal{L}$ (given by L) of the running example are depicted. On the right side the p/t-nets synthesized from L with Algorithm 1 and Algorithm 4 are shown. The partial language may be specified in the VipTool editor or as an xml-file. The synthesized net is stored as a pnml-file and can be visualized with VipTool. Figure 15 shows an example of the user interface of VipTool.

As explained, the practical performance of the presented synthesis algorithms is quite hard to estimate on a theoretical level. Therefore the implementation is used to get experimental results on the performance and on the size of the computed nets of the synthesis algorithms. In particular a practical comparison of the alternative synthesis approaches pursued by Algorithm 1 and Algorithm 4 is shown.

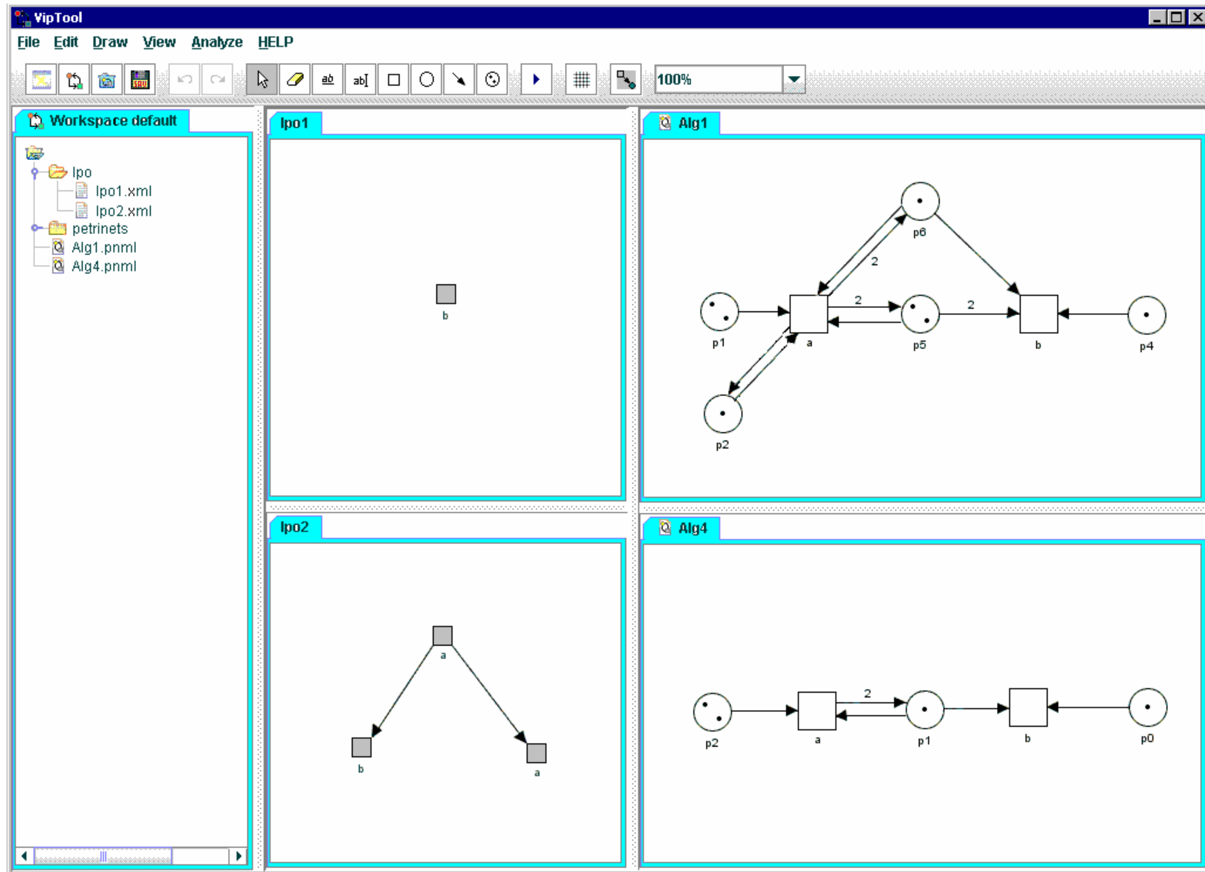

Figure 15. Screenshot of VipTool: the LPOs $lpo1$ and $lpo2$ define a partial language $\mathcal{L}$. $Alg1$ respectively $Alg4$ show the nets synthesized from these two LPOs by our implementation of Algorithm 1 respectively Algorithm 4.

We show experimental results of the implementation of the synthesis algorithm based on token flow regions combining Algorithm 1 and Algorithm 2 and of the synthesis algorithm based on classical regions combining Algorithm 4 and Algorithm 2.[5] The considered partial languages are given by subsets of the LPOs shown in Figure 15, Figure 16 and Figure 17. The table on the page after next shows experimental results for some example partial languages indicated in the column ”Language” by numbers referring to the LPOs defining the language. The table shows the runtime in milli seconds (”ms”) of Algorithm 1 and Algorithm 4, and in each case the runtime of the optimistic equality test Algorithm 2 for the synthesized nets. We also show the number of basis regions (”#b”) and the final number of places (”#p”) of the net computed with Algorithm 1 as well as the number of places (”#p”) of the net synthesized with Algorithm 4. Concerning Algorithm 2 we depict the final result of the equality test (”solv”).

We ran three test series. A first one increasing the number of considered LPOs (lpo1 - lpo6), a second one increasing the number of nodes in one LPO (lpo7a - lpo7c, lpo8) and a third one increasing the number of concurrent nodes in one LPO (lpo9a - lpo9e).

[5]We used Java SE 1.5.0 on an Intel Xeon 2.8 GHz machine with 3072 MB RAM running Microsoft Windows Server 2003 SE operating system.

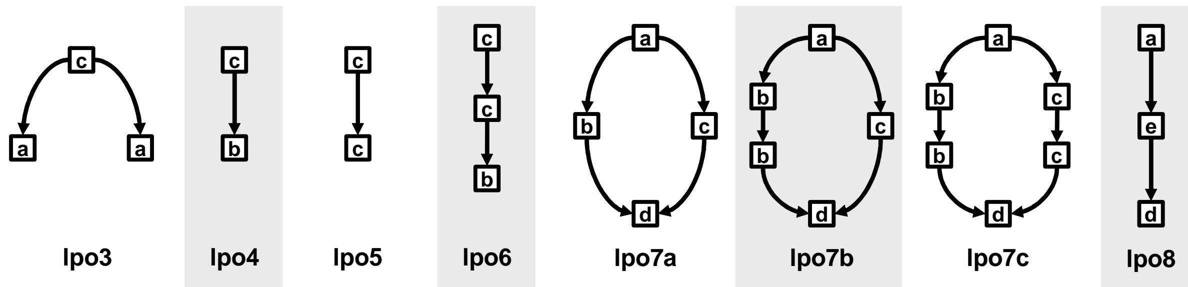

Figure 16. Example LPOs lpo3, lpo4, lpo5, lpo6, lpo7a, lpo7b, lpo7c, lpo8.

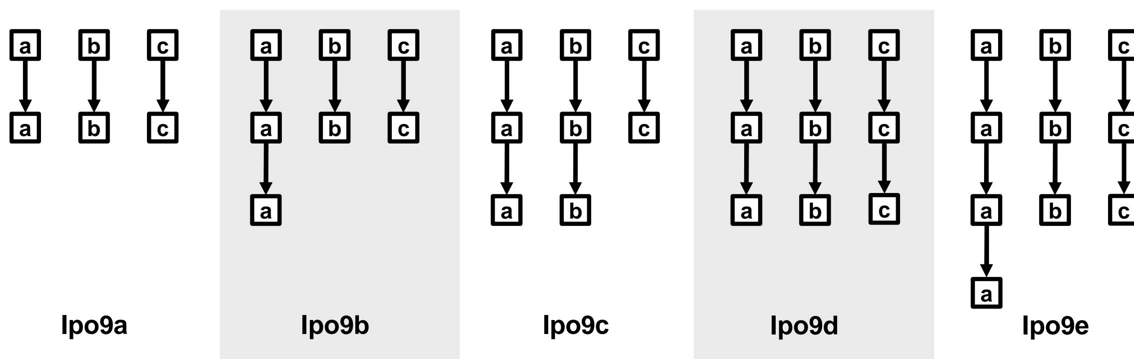

Figure 17. Example LPOs lpo9a, lpo9b, lpo9c, lpo9d, lpo9e.

It is obvious that Algorithm 4 generates smaller nets than Algorithm 1. The number of places of the synthesized nets is considerably smaller in all three test series. Thus the performance of the equality test Algorithm 2 is significantly better in the case of Algorithm 4. Concerning the runtime of Algorithm 1 in particular the follow-up equality test Algorithm 2 (or more precisely the unfolding algorithm of Algorithm 2) seems problematic (the runtime entry ”-” means that the algorithm did not finish in reasonable time), although we have already chosen an efficient unfolder. We are currently working on an even more efficient unfolding algorithm. Also further methods to delete implicit places in the synthesized nets could improve the situation. It could also be interesting to implement the pessimistic equality test (Algorithm 3), which does not include an unfolding algorithm, instead of the optimistic test used here.

Algorithm 1 is fast if the LPOs include a lot of concurrency as in the third test series. If the specified LPOs do not exhibit too much concurrency, as in the second test series, or even very low level concurrency, as in the first test series, Algorithm 4 is very efficient (compared to Algorithm 1). The follow-up equality test with Algorithm 2 is very fast in all examples. Thus in the case of Algorithm 4 it seems not necessary to shift to one of the alternative equality tests Algorithm 3 or Algorithm 5. The runtime of Algorithm 4 gets into serious difficulties, when the degree of concurrency in the LPOs is high. The third test series shows that in this case the runtime of Algorithm 4 grows very fast and that it is significantly worse than in Algorithm 1. Algorithm 4 does not synthesize a net in reasonable time already for quite small examples exhibiting much concurrency.

Language	Synthesis with token flow regions					Synthesis with classical regions			
	Algorithm 1			Algorithm 2		Algorithm 4		Algorithm 2	
	ms	#b	#p	ms	solv	ms	#p	ms	solv
{*1, 2*}	131	9	5	67	true	9	3	8	true
{*1, 2, 3*}	266	35	7	81	true	19	3	11	true
{*1, 2, 3, 4*}	390	88	12	134	true	29	4	23	true
{*1, 2, 3, 4, 5*}	1.890	220	17	-	-	32	4	24	true
{*1, 2, 6*}	359	48	14	5.901	false	28	4	false	
{*7a, 8*}	279	49	19	96	true	26	5	10	true
{*7b, 8*}	407	149	27	1301	true	35	6	20	true
{*7c, 8*}	734	321	41	-	-	76	7	26	true
{*9a*}	203	10	6	18	true	2107	6	25	true
{*9b*}	218	15	7	85	true	19759	6	59	true
{*9c*}	250	20	8	141	true	227680	6	77	true
{*9d*}	282	25	9	192	true	3621130	6	102	true
{*9e*}	312	55	9	403	true	-	-	-	-

6. Conclusion

In this paper we presented two methods, given a finite set of LPOs representing a partial language, how to compute a (finite) marked p/t-net being the only candidate to solve the synthesis problem. In both cases region theory is applied, such that each specified LPO is a run of the computed net. One method is based on the application of classical regions to the set of step sequences generated by the partial language. The other methods uses a novel definition of token flow regions of partial languages. For both methods, we developed effective methods to test whether the computed net has more runs than specified or not. This decides the synthesis problem, since the synthesis problem has a solution if and only if the computed net does not have more runs than specified. We implemented both methods in our tool VipTool and finally presented experimental results comparing the runtime of both methods. As expected, the innovative method based on token flow regions significantly outperforms the classical approach, if the given LPOs specify sufficient concurrency among events. If there is only low level concurrency the classical approach is very fast.

The restriction to finite partial languages need not be a problem in practice. An application field of synthesis methods is the discovery of workflow processes (given as Petri nets) from finite event logs [2, 5]. A finite event log is a finite set of finite sequences of observed actions, thus a special type of a finite partial language (without concurrency). Currently there is a research effort to deduce information about concurrency from event logs. Another natural application field is the specification of system behavior via message sequence charts, which are special types of LPOs. In both cases, the synthesized net can be used for analysis purposes.

The next step of research is the examination of the special instances of polyhedral cones used in the algorithm in view of a better upper bound for the number of basis solutions. We also work on a

generalization of the presented results to infinite partial languages which allow a finite representation (for example a term-based representation).

References

[1] van der Aalst, W. M. P., van Dongen, B. F., Herbst, J., Maruster, L., Schimm, G., Weijters, A. J. M. M.: Workflow mining: A survey of issues and approaches., *Data Knowl. Eng.*, **47**(2), 2003, 237–267.

[2] 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.

[3] Badouel, E., Darondeau, P.: *On the Synthesis of General Petri Nets.*, Technical Report 3025, Inria, 1996.

[4] Badouel, E., Darondeau, P.: Theory of Regions., *Petri Nets* (W. Reisig, G. Rozenberg, Eds.), Lecture Notes in Computer Science 1491, Springer, 1998, 529–586.

[5] Bergenthum, R., Desel, J., Lorenz, R., Mauser, S.: Process Mining Based on Regions of Languages., *BPM 2007* (G. Alonso, P. Dadam, M. Rosemann, Eds.), Lecture Notes in Computer Science 4714, Springer, 2007, 375–383.

[6] Bergenthum, R., Lorenz, R., Mauser, S.: Faster Unfolding of General Petri Nets., *Proceedings 14. Workshop Algorithmen und Werkzeuge für Petri Netze (AWPN)*, 2007, 63–68.

[7] Cortadella, J., Kishinevsky, M., Kondratyev, A., Lavagno, L., Yakovlev, A.: Petrify: A tool for manipulating concurrent specifications and synthesis of asynchronous controllers., *IEICE Trans. of Informations and Systems*, **E80-D**(3), 1997, 315–325.

[8] Cortadella, J., Kishinevsky, M., Kondratyev, A., Lavagno, L., Yakovlev, A.: Hardware and Petri Nets: Application to Asynchronous Circuit Design., *ICATPN 2000* (M. Nielsen, D. Simpson, Eds.), Lecture Notes in Computer Science 1825, Springer, 2000, 1–15.

[9] Darondeau, P.: Deriving Unbounded Petri Nets from Formal Languages., *CONCUR 1998* (D. Sangiorgi, R. de Simone, Eds.), Lecture Notes in Computer Science 1466, Springer, 1998, 533–548.

[10] Desel, J.: From Human Knowledge to Process Models., *to appear in: Proceedings of UNISCON*, 2008.

[11] Desel, J., Lorenz, R., Mauser, S., Bergenthum, R.: VipTool-Homepage, 2008, Http://www.informatik.ku-eichstaett.de/projekte/vip/.

[12] Desel, J., Reisig, W.: The Synthesis Problem of Petri Nets., *Acta Inf.*, **33**(4), 1996, 297–315.

[13] Donatelli, S., Thiagarajan, P. S., Eds.: *Petri Nets and Other Models of Concurrency - ICATPN 2006, 27th International Conference on Applications and Theory of Petri Nets and Other Models of Concurrency, Turku, Finland, June 26-30, 2006, Proceedings*, vol. Lecture Notes in Computer Science 4024 of *Lecture Notes in Computer Science*, Springer, 2006.

[14] Ehrenfeucht, A., Rozenberg, G.: Partial (Set) 2-Structures. Part I: Basic Notions and the Representation Problem., *Acta Inf.*, **27**(4), 1989, 315–342.

[15] Ehrenfeucht, A., Rozenberg, G.: Partial (Set) 2-Structures. Part II: State Spaces of Concurrent Systems., *Acta Inf.*, **27**(4), 1989, 343–368.

[16] Grabowski, J.: On partial languages., *Fundamenta Informaticae*, **4**(2), 1981, 428–498.

[17] Hoogers, P., Kleijn, H., Thiagarajan, P.: A trace semantics for Petri nets., *Information and Computation*, **117**(1), 1995, 98–114.

[18] Josephs, M. B., Furey, D. P.: A Programming Approach to the Design of Asynchronous Logic Blocks., *Concurrency and Hardware Design* (J. Cortadella, A. Yakovlev, G. Rozenberg, Eds.), Lecture Notes in Computer Science 2549, Springer, 2002, 34–60.

[19] Juhás, G., Lorenz, R., Desel, J.: Can I Execute My Scenario in Your Net?, *ICATPN 2005* (G. Ciardo, P. Darondeau, Eds.), Lecture Notes in Computer Science 3536, Springer, 2005, 289–308.

[20] Kiehn, A.: On the Interrelation Between Synchronized and Non-Synchronized Behaviour of Petri Nets., *Elektronische Informationsverarbeitung und Kybernetik*, **24**(1/2), 1988, 3–18.

[21] Lorenz, R., Bergenthum, R., Desel, J., Mauser, S.: Synthesis of Petri Nets from Finite Partial Languages., *ACSD 2007*, IEEE Computer Society, 2007, 157–166.

[22] Lorenz, R., Juhás, G.: Towards Synthesis of Petri Nets from Scenarios., in: Donatelli and Thiagarajan [13], 302–321.

[23] Lorenz, R., Juhás, G., Mauser, S.: How to Synthesize Nets from Languages - a Survey., *Proceedings of the Wintersimulation Confernce (WSC) 2007*, IEEE Computer Society, 2007, 637–647.

[24] Minkowski, H.: *Geometrie der Zahlen*, Teubner, 1896.

[25] Motzkin, T.: *Beiträge zur Theorie der linearen Ungleichungen*, Ph.D. Thesis, Jerusalem, 1936.

[26] Pratt, V.: Modelling Concurrency with Partial Orders., *Int. Journal of Parallel Programming*, **15**, 1986, 33–71.

[27] Schrijver, A.: *Theory of Linear and Integer Programming*, Wiley, 1986.

[28] Tschernikow, S. N.: Algorithm for finding a general formula for the non-negative solutions of a system of linear inequalities., *USSR Computational Mathematics and Mathematical Physics*, **5**(2), 1965, 228–233.

[29] Vanderbei, R. J.: *Linear Programming: Foundations and Extensions*, Kluwer Academic Publishers, 1996.

[30] Vogler, W.: *Modular Construction and Partial Order Semantics of Petri Nets*, vol. 625 of *Lecture Notes in Computer Science*, Springer, 1992.