Synthesizing Petri Nets from Hasse Diagrams

Robin Bergenthum$^{(\boxtimes)}$

FernUniversität in Hagen, Hagen, Germany
`robin.bergenthum@fernuni-hagen.de`

Abstract. Synthesis aims at producing a process model from specified sample executions. A user can specify a set of executions of a system in a specification language that is much simpler than a process modeling language. The intended process model is then constructed automatically.

Synthesis algorithms have been extensively explored for cases where the specification language is a reachability graph or a sequential language. Concerning synthesis from partial languages, however, there is a significant gap between theory and practical application. In the literature, we find two different synthesis methods for partial languages, but both have poor runtime even in reasonably sized practical examples. In this paper, we introduce a new and more efficient synthesis algorithm for partial languages based on Hasse diagrams.

1 Introduction

Complex business processes are often modeled by means of Petri nets [1,3,17,29, 30]. Petri nets have formal semantics, an intuitive graphical representation, and are able to express concurrency among the occurrence of actions. Petri nets are the formal basis for many workflow modeling languages. However, constructing a Petri net model for a real world process is a costly and error-prone task [1,3,28].

Fortunately, whenever we model a system, there are often some associated descriptions or even specifications of the desired processes. There may be logfiles of recorded behavior, example runs, or product specifications describing use cases. Such specifications can be formalized by a set of words, a reachability graph, or a partial language. Yet, only partial languages are able to explicitly express concurrency between events. Thus, partial languages have drawn much attention recently [5,20].

If a specification is incomplete or contains so-called noise, there are algorithms developed in the area of process mining [1,2] to still automatically generate a suitable business process model. If a specification is complete (i.e. is the desired behavior), we can synthesize a model. The synthesis problem is to compute a process model so that: (A) the specification is a subset of the language of the generated model and (B) the generated model has minimal additional behavior.

To showcase a typical use case of synthesis-based model generation, we assume a coffee brewing process together with a domain expert on this process called Robin. Robin has been brewing coffee for years, but just recently received a training in process modeling to document standard processes in his department. Robin observes a sample execution of his process and records the following

sequence of events: *grind beans*, *unlock machine*, *empty strainer*, *clean coffee pot*, *get water with coffee pot*, *fill kettle*, *fill strainer*, *assemble and turn on*. In a second sample, he uses a glass pot (instead of the coffee pot) to fetch water from the kitchen. With this in mind, Robin builds a naive Petri net-like process model of his process depicted in Fig. 1. All actions of the process are ordered in a sequence and there is an XOR-split modeling the choice between the two transitions *get water with coffee pot* and *get water with glass pot*.
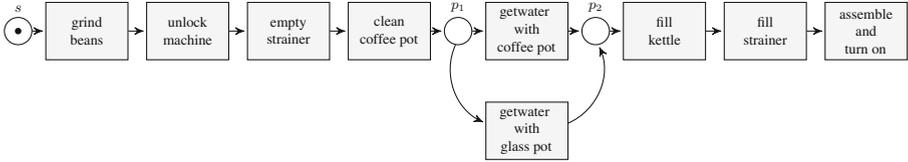
**Fig. 1.** Petri net-like process model of the coffee brewing process.

We (modeling experts) try to rate the validity of Robin's model. We realize that most of the modeled dependencies may be superfluous. For example, unlocking the coffee machine usually does not depend on grinding beans. Cleaning the coffee pot only depends on unlocking the machine. What is worse, some of these dependencies may change if we consider different executions of the process. In a scenario where we use the coffee pot to get water from the kitchen, the event *fill kettle* depends on the sequence of actions *unlock coffee machine*, *clean coffee pot*, and *get water with coffee pot*. In another scenario where we use the additional glass pot to fetch water, the event *fill kettle* only depends on unlocking the machine and getting water. Thus, the relation between the occurrences of *fill kettle* and *clean coffee pot* has changed.

There are many possible pitfalls in this small example. Even if Robin understands the concepts that may cause trouble here, he most likely will not be able to adapt his model accordingly. To tackle this problem, we apply a synthesis-based model generation approach. We revisit both initially observed sample executions and depicted them in Fig. 2. In a next step, we ask Robin to delete unnecessary dependencies between observed events. With his expert knowledge on the process at hand, he for example starts to delete the dependency between *grind beans* and *unlock machine* in the first sequence, thus creating a partial order step by step.
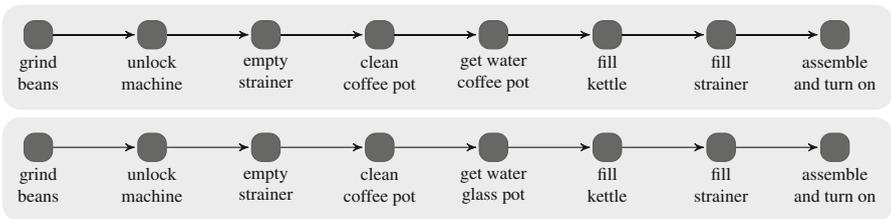
Fig. 2. Observed events of the coffee brewing process.

Robin continues to delete all superfluous dependencies from both observed sequences and finally comes up with the two labeled Hasse diagrams depicted in Fig. 3. Every labeled Hasse diagram specifies a different run of the coffee brewing process. In the first run, we grind beans and unlock the coffee machine. Once the machine is unlocked, we empty the strainer and clean the coffee pot. After it is cleaned, we fetch water from the kitchen using the coffee pot. When the strainer and the kettle are filled, we assemble and turn on the coffee machine. Every arc models a dependency between the occurrence of the related actions and unordered events occur concurrently. In the second run, we use a glass pot (instead of the coffee pot) to fetch water from the kitchen. This activity does not depend on unlocking or cleaning the coffee pot. We can get water right at the beginning of this sample run. Altogether, Fig. 3 depicts a complete and intuitive specification of our coffee brewing process.
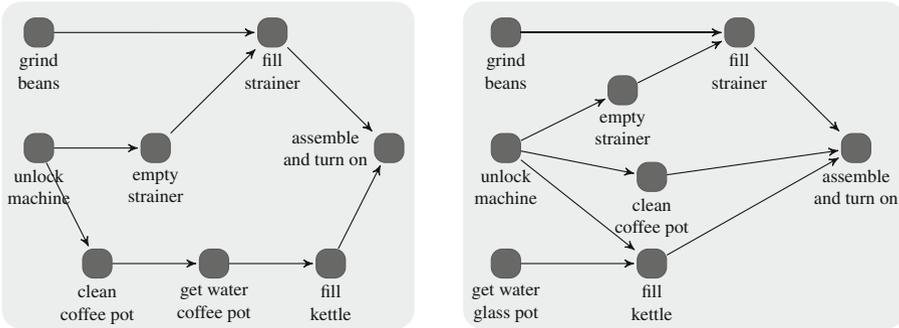

Fig. 3. Two labeled Hasse diagrams, i.e. a specification of the coffee brewing process.

The last step in a synthesis-based model generation approach is to construct a model matching Robin's specification, thus solving the synthesis problem. Such a model offers an integrated view of the process at hand, is more compact, can be analyzed by well-known Petri net algorithms, and can serve as an input for workflow engines.

In this paper, we present a new synthesis technique to automatically transform a specification into a valid process model. The main benefit is that single executions are much easier to model than the complex system itself. To confirm this claim, we take a look at the model depicted in Fig. 4 modeling the coffee brewing process specified in Fig. 3. This model is generated using the algorithm presented in the remainder of this paper. We will recall Petri nets and their partial language in the preliminaries; here, it is sufficient to state that this model has exactly the specified behavior of the coffee brewing process.

Taking a look at the literature, scenario-based modeling approaches are an acknowledged research topic. There is a vast variety of specialized approaches
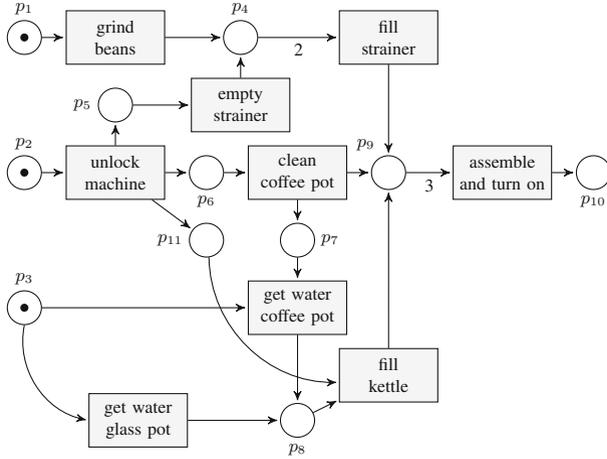
Fig. 4. A marked p/t-net modeling a coffee brewing process.

using different specification languages and process models (see for example [15, 16, 22, 23]). All these approaches have in common that they assume the specification to be valid and rather complete. This is the main requirement for applying precise generation algorithms like synthesis or folding methods. The theory of the related synthesis algorithms is called region theory [4, 21]. Region theory has been extensively explored for reachability graphs and sequential languages. There are many non-trivial theoretical results, notions, case studies, as well as tools like ProM [19], Genet [14], or Viptool [10] (see for example [11–13, 31, 32] for some recent publications). Concerning region theory for partial languages, however, there is a significant gap between theory and practical application. There are two different notions of regions for partial languages: tokenflow regions [9] and transition regions [8, 9]. Yet, both related algorithms perform poorly even in reasonably sized practical examples [8]. In Sect. 3, we introduce a synthesis algorithm based on a new concept called compact regions. The name stems from the fact that while tokenflow regions relate to occurrence nets and transition regions relate to step sequences, the new notion relates to compact tokenflows [6, 7] and Hasse Diagrams, i.e. a much more compact representation of a partial language. We show that the concept of compact regions introduced here leads to a much faster synthesis algorithm.

The paper is organized as follows: Sect. 2 introduces Petri nets, their partial language, and the synthesis problem. In Sect. 3, we recall the concept of compact tokenflows and introduce compact regions. We prove that compact regions solve the synthesis problem for Petri nets and partial languages. At the end of Sect. 3, we deduce our synthesis algorithm from the new definition of compact regions. In Sect. 4, we discuss the runtime of the new algorithm. We compare the algorithm to its predecessors i.e. algorithms based on transition regions and (ordinary) tokenflow regions. We implement all synthesis techniques in a tool called MoPeBs Eagle Owl and present runtime tests.

2   Preliminaries

Let f be a function and B be a subset of the domain of f. We write $f|_B$ to denote the restriction of f to B. We call a function $m : A \to \mathbb{N}$ a multiset and write $m = \sum_{a \in A} m(a) \cdot a$ to denote multiplicities of elements in m. Let $m' : A \to \mathbb{N}$ be another multiset. We write $m \geq m'$ if $\forall a \in A : m(a) \geq m'(a)$ holds. We denote the transitive closure of an acyclic and finite relation $<$ by $<^*$. We denote the skeleton of $<$ by $<^\diamond$. The skeleton of $<$ is the smallest relation $\lessdot$ such that $\lessdot^* = <^*$ holds. Let $(V, <)$ be some acyclic and finite graph, $(V, <^\diamond)$ is called its Hasse diagram. We model business processes by p/t-nets [3,18,29,30].

Definition 1. *A place/transition net (p/t-net) is a tuple (P, T, W) where P is a finite set of places, T is a finite set of transitions such that $P \cap T = \emptyset$ holds, and $W : (P \times T) \cup (T \times P) \to \mathbb{N}$ is a multiset of arcs. A marking of (P, T, W) is a multiset $m : P \to \mathbb{N}$. Let m_0 be a marking, we call the tuple $N = (P, T, W, m_0)$ a marked p/t-net and m_0 the initial marking of N.*

Figure 4 depicts a p/t-net modeling a coffee brewing process. Transitions are rectangles, places are circles, the multiset of arcs is represented by weighted arcs, and the initial marking is represented by black dots called tokens. There is a simple firing rule for transitions of a p/t-net: let t be a transition of a marked p/t-net (P, T, W, m_0). We denote $\circ t = \sum_{p \in P} W(p, t) \cdot p$ the weighted preset of t. We denote $t \circ = \sum_{p \in P} W(t, p) \cdot p$ the weighted postset of t. A transition t is enabled (can fire) at marking m if $m \geq \circ t$ holds. Once transition t fires, the marking changes from m to $m' = m - \circ t + t \circ$. In our example p/t-net, the transitions *grind beans*, *unlock machine*, and *get water with glass pot* can fire at the initial marking. If *unlock coffee machine* fires, this removes the token from the place in its preset and produces two new tokens: one token in the preset of *empty strainer* and another token in the preset of *clean coffee pot*. As soon as *get water glass pot* fires, the token from the lower left place is removed and one token is produced in the preset of *fill kettle*. Note: firing *get water glass pot* disables transition *get water coffee pot* for the rest of this process. Concerning arc weights, the transition *assemble and turn on* is enabled if there are at least three tokens in the rightmost place p_9. Repeatedly processing the firing rule produces firing sequences. These firing sequences are the most basic behavioral model of Petri nets. Let N be a marked p/t-net, the set of all initially enabled firing sequences of N is the sequential language of N.

Petri nets and most business process modeling languages are able to express concurrency of the occurrences of transitions. For example, transitions *grind beans* and *unlock machine* can fire independently from one another. Roughly speaking, they can fire without any order while not sharing resources. However, firing sequences are not able to capture or describe such behavior. The common behavioral model for partially ordered behavior of Petri nets is a so-called process nets [25]. A process net is a Petri net modeling only one single partially ordered run of a marked p/t-net. For a formal definition of process nets we refer to [25,30]. Here, as an example, we depict a process net of our coffee brewing Petri net of Fig. 4.

Every place of a process net relates to a token of the related p/t-net. For example in Fig. 5, there are three places labeled p_9 in the preset of the transition *assemble and turn on*. The set of process nets of a p/t-net is called its unfolding. Events, loops, tokens, and conflicts are unfolded to present single executions of the related net.
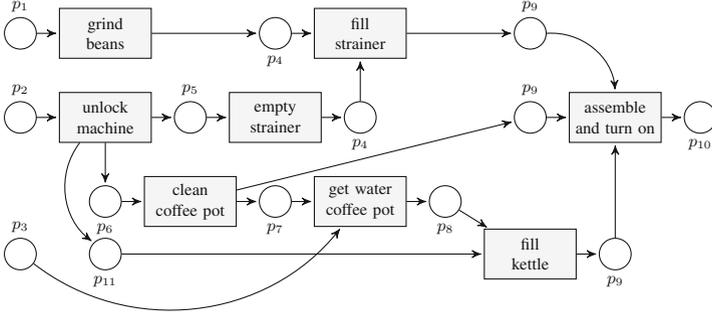
Fig. 5. One process net of the coffee brewing process.

If we abstract from the places of a process net related to a p/t-net N, we have a set of events arranged in a partial order. Just like some valid firing sequence, this partially ordered set of events is enabled in the p/t-net. In other words, we can replay such a partial order by firing transitions of N where unordered parts of the partial order can fire concurrently. The set of labeled partial orders induced by all processes of N is the partial language of N. For example, the left half of Fig. 3 depicts the labeled Hasse diagram representing the partial order underlying Fig. 5. Thus, this labeled partial order is in the partial language of the p/t-net depicted in Fig. 4.

Definition 2. *Let T be a set of labels. A labeled partial order (lpo) is a triple $lpo = (V, <, l)$ where V is a finite set of events, $< \subseteq V \times V$ is a transitive and irreflexive relation, and the labeling function $l : V \rightarrow T$ assigns a label to every event.*

Definition 3. *Let $K = (C, E, F, \rho)$ be a process net of a marked p/t-net (P, T, W, m_0) where C is a set of conditions, E is a set of events, F is a set of arcs, and $\rho : (C \cup E) \rightarrow (P \cup T)$ is a labeling function. The lpo $(E, F^*|_{E \times E}, \rho|_E)$ is the process lpo of K.*

Let N be a marked p/t-net and $L^{\Pi}(N)$ be the set of all process lpos of N. $L(N) = \{(E, <, l) | (E, <, l) \text{ an lpo}, (E, <', l) \in L^{\Pi}(N), <' \subseteq <\}$ is the partial language of N.

As we already pointed out in the introduction, our goal is to synthesize a p/t-net from a specification describing partially ordered behavior. The most suitable manner to represent a partial language is by means of labeled Hasse diagrams (see Fig. 2). A labeled Hasse diagram is a finite set of events ordered by the

skeleton of a partial order. Clearly, the transitive closure of a Hasse diagram is an lpo. Thus, the prefix- and sequentialisation-closure of a set of labeled Hasse diagrams is a partial language.

Definition 4. *A triple $run = (V, <, l)$ is a labeled Hasse diagram if $(V, <^*, l)$ is an lpo and $<^\diamond = <$ holds. A finite set of labeled Hasse diagrams is a specification. Let $run = (V, <, l)$ be a labeled Hasse diagram, we define $run^* = (V, <^*, l)$.*

Definition 5. *Let N be a marked p/t-net and $S = \{run_1, \ldots, run_n\}$ be a specification. We write $S \subseteq L(N)$ iff $\{run_1^*, \ldots, run_n^*\} \subseteq L(N)$ holds.*

Finally, we are able to define the synthesis problem. The synthesis problem is to construct a p/t-net such that its behavior matches a specification. If there is no such p/t-net, we construct a p/t-net such that its behavior includes the specification and has minimal additional behavior.

Definition 6. *Let S be a specification, the synthesis problem is to compute a marked p/t-net N such that the following conditions hold: $S \subseteq L(N)$ and for all marked p/t-nets $N' : L(N) \backslash L(N') \neq \emptyset \implies S \nsubseteq L(N')$.*

3   Compact Regions and Synthesis Algorithm

The algorithm presented in this paper is based on the theory of regions [21]. For an introduction to region theory, we refer the reader to [4]. As stated in the introduction, the input to our algorithm is a set of labeled Hasse diagrams (see Fig. 3). The first step is to construct a transition for every label to get an initial p/t-net without places. The language of this net includes arbitrary behavior because all the transitions have an empty preset and can fire in any order. Obviously, we need to add places and arcs to restrict the behavior of this initial net. To solve the synthesis problem, we are only allowed to add places and connected arcs that do not inhibit our specification.

Definition 7. *Let S be a specification and $N = (P, T, W, m_0)$ be a marked p/t-net. A place $p \in P$ is called feasible for S iff $S \subseteq L((\{p\}, T, W|_{(\{p\} \times T) \cup (T \times \{p\})}, m_0(p)))$ holds. Let S be a specification and $N = (\{p\}, T, W, m_0)$ be a marked one-place p/t-net. We call N feasible for S iff p is feasible for S.*

If we are able to identify feasible places, we can add these to our initially placeless p/t-net. These places restrict the behavior, yet such a net will still be able to execute all the labeled Hasse diagrams of the specification.

Remark 1. *Let S be a specification and let a set of p/t-nets $\{(\{p_1\}, T, W_1, m_1), \ldots, (\{p_n\}, T, W_n, m_n)\}$ be feasible for S. Let $N = (\bigcup_i \{p_i\}, T, \sum_i W_i, \sum_i m_i)$ be the union of all feasible nets, every place of N is feasible and $S \subseteq L(N)$ holds.*

Theoretically, we could restrict the behavior of the initial p/t-net by adding the set of all feasible places. This would guarantee that the behavior of the net cannot be restricted further without excluding some executions of the specification. This is a fundamental theorem of region theory (see for example [4]).

Theorem 1. *Let S be a specification and T be its set of labels. The p/t-net which is the union of all p/t-nets feasible for S is a solution of the synthesis problem.*

Practically, we need to construct a finite p/t-net with the same behavior as the union-of-all-feasible-places p/t-net. For partial languages, there are two strategies to tackle this problem: We can either calculate a basis of all feasible places and add them to the initial set of transitions. This is always possible and the basis is always finite. According to the firing rules of Petri nets, this net behaves like the infinite p/t-net. In other words, the finite basis p/t-net also solves the synthesis problem. Or we can use the technique of so-called wrong continuations. Roughly speaking, the set of wrong continuations is the border between the specified and all other behaviors. The set of wrong continuations is finite as long as the specification is finite as well. For each wrong continuation, we add one feasible place, thus excluding the wrong continuation from the language of the constructed net. The resulting finite p/t-net solves the synthesis problem as well.

The next step of our algorithm is to characterize the set of all feasible places. To develop an efficient synthesis algorithm, we rely on the behavioral model of compact tokenflows [6,7]. A compact tokenflow is a distribution of tokens along the Hasse diagram of a labeled partial order. A labeled Hasse diagram is in the partial language of a p/t-net if there is a compact tokenflow distributing tokens such that three conditions hold: first, every event receives enough tokens, second, no event has to pass too many tokens, and third, the initial marking is not exceeded. Tokens must be received from the particular presets of events. Thus, we ensure that consumed tokens are available before the actual event occurs. If a transition produces tokens, the related events are allowed to produce tokenflow in the Hasse diagram and pass these tokens to their particular postsets. If an event receives tokens, it consumes the tokenflow needed and passes the redundant tokenflow to later events. Tokens of the initial marking are free for all, i.e. any event can consume tokens from the initial marking.

Definition 8. *Let $N = (P, T, W, m_0)$ be a marked p/t-net and $run = (V, <, l)$ be a labeled Hasse diagram such that $l(V) \subseteq T$ holds. A compact tokenflow is a function $x : (V \cup <) \to \mathbb{N}$. x is compact valid for $p \in P$ iff the following conditions hold:*

(i) $\forall\, v \in V$: $x(v) + \sum_{v' < v} x(v', v) \geq W(p, l(v))$,
(ii) $\forall\, v \in V$: $\sum_{v < v'} x(v, v') \leq x(v) + \sum_{v' < v} x(v', v) - W(p, l(v)) + W(l(v), p)$,
(iii) $\sum_{v \in V} x(v) \leq m_0(p)$.

run is compact valid for N iff there is a compact valid tokenflow for every $p \in P$.
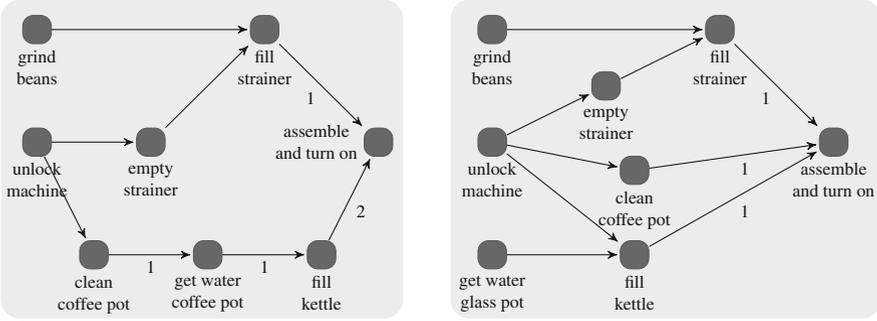
Fig. 6. Two valid compact tokenflows for place p_9 of the marked p/t-net of Fig. 4.

For an example of a compact tokenflow, we consider place p_9 of Fig. 4 and depict two related compact valid tokenflows for the two labeled Hasse diagrams introduced earlier in Fig. 6. We take a look at Fig. 4 and only consider p_9 and its related arcs. We see that all events labeled *assemble and turn on* need to receive three tokens, whereas all events labeled *fill strainer*, *clean coffee pot*, or *fill kettle* can produce one token. In the Hasse diagrams of Fig. 5, tokenflow is depicted as integers on the related arcs and events (the integer 0 is not shown). According to the depicted tokenflow, in the first diagram *fill strainer* and *clean coffee pot* create one token each. The event *get water coffee pot* cannot create tokens for p_9, but receives a token from *clean coffee pot* and passes this token to *fill kettle*. The event *fill kettle* receives one token and produces another one. Thus, *fill kettle* passes two tokens to *assemble and turn on*. Altogether, *assemble and turn on* receives three tokens and all the conditions for a valid compact tokenflow hold for p_9. All in all, we need to construct eleven compact tokenflows related to the eleven places of the p/t-net of Fig. 4 to deduce that this labeled Hasse diagram is in the language of the p/t-net. In the second Hasse diagram of Fig. 6, three tokens directly reach the event labeled *assemble and turn on*. Again, all conditions for a valid compact tokenflow hold for p_9. Compared to the notion of process nets [25] and to the notion of (ordinary) tokenflows [26,27], the main advantage of compact tokenflows is that they only consider the Hasse diagrams of the specification. Process nets as well as ordinary tokenflows need to consider the complete (i.e. transitive) relation. Previous work [6,7] proves that these three notions are equivalent, i.e. they all define the same partial language. For the proof, we refer to [6] but state the following theorem.

Theorem 2. *The language of a marked p/t-net is well-defined by the set of compact valid labeled Hasse diagrams.*

In our algorithm we take advantage of compact tokenflows and define a new notion of regions, i.e. compact regions, for partial languages.

Definition 9. *Let $S = \{(V_1, <_1, l_1), \ldots, (V_n, <_n, l_n)\}$ be a specification, T be its set of labels, and p be a place. We denote V_i' the set of events with an empty prefix*

in $(V_i, <_i, l_i)$. A function $r : (\bigcup_i (V'_i \cup <_i) \cup (T \times \{p\}) \cup (\{p\} \times T) \cup \{p\}) \to \mathbb{N}$ is a compact region for S iff $\forall i \in \mathbb{N} : r|_{\{V'_i \cup <_i\}}$ is compact valid for p in $(\{p\}, T, r|_{(T \times \{p\}) \cup (\{p\} \times T)}, r(p))$.

We reconsider our sample brewing process depicted in Fig. 5. We assume, both Hasse diagrams are the input to our synthesis algorithm. The main idea of the algorithm is to construct compact regions. In this example, we implement the domain of a compact region by 41 non-negative integer unknowns. The first 19 unknowns represent a place. A place may have one weighted arc leading to each of the nine transitions related to the labels of our example, one weighted arc coming from each of the nine transitions, and an additional unknown for its initial marking. The next ten unknowns represent a compact tokenflow of the first Hasse diagram. One unknown for each of the eight arcs and two additional unknowns for each of the two minimal events. The last twelve unknowns represent a compact tokenflow of the second Hasse diagram. Again, one unknown for each of the nine arcs and three additional unknowns for each of the three minimal events. Only if all 41 values of these unknowns relate to two compact tokenflows valid for the defined place, this vector is a compact region. For example, assume a fixed ordering of all 41 unknowns, the vector $(0, 1, 3, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 2, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0,$ $0, 0, 0, 0)$ may be a compact region. Assuming a correct ordering of all unknowns, the first third of this vector defines the arc weights of p_9 depicted in Fig. 3. The second third defines the compact tokenflow on the left side of Fig. 5. The last third defines the compact tokenflow on the right side of Fig. 5.

To state the correctness of our synthesis algorithm we have to prove that if the compact tokenflows are valid for the defined place, a region defines a feasible place.

Theorem 3. *Let S be a specification and T be its set of labels. Every compact region r for S defines a feasible p/t-net $N_r = (\{p\}, T, W, m_0)$ and vice versa.*

Proof. Let r be a compact region. For every labeled Hasse diagram in S, there is a valid compact tokenflow $r|_{\{V'_i \cup <_i\}}$ of p in $N_r = (\{p\}, T, r|_{(T \times \{p\}) \cup (\{p\} \times T)}, r(p))$. $S \subseteq L(N_r)$ holds and so N_r is feasible for S.

Let $N = (\{p\}, T, W, m_0)$ be a feasible p/t-net such that $S \subseteq L(N)$ holds. There is a valid compact tokenflow r_i for every labeled Hasse diagram of S. Without loss of generality, every r_i is zero on events with a non-empty prefix. This holds because as long as some valid compact tokenflow is positive for some event e with a non-empty prefix, move this tokenflow to an event e' in its direct prefix and adopt the compact tokenflow on the arc (e', e) accordingly. The union $r = \bigcup_i r_i \cup W \cup m_0$ is a compact region. $\square$

Every region is a vector of numbers respecting the conditions (i), (ii), and (iii) of Definition 9. With this in mind, we are able to express all feasible p/t-nets by a single inequality system. Again, in this system, there is an unknown for every element in the domain of a compact region, i.e. one unknown for every minimal event, another unknown for every arc, two unknowns for every label, and a single unknown for the initial marking. The inequality system is built from

the inequalities defined in Definition 8. According to (i) and (ii), there are two inequalities for every event of the specification. According to (iii), there is another inequality for every labeled Hasse diagram. The set of positive integer solutions of this inequality system is the set of all feasible nets. We call this inequality system the compact region inequality system. Every solution of this system defines one feasible place. Altogether, the compact region inequality system has 41 unknowns as well as 34 inequalities in our coffee brewing example.

Finally, we depict the complete synthesis algorithm using compact regions in Algorithm 1. Input is a set of Hasse diagrams H. We construct a Petri net with an empty set of places and a transition for every label, calculate the compact region inequality system, and the set of wrong continuations of H. For every wrong continuation c we check if it is still executable in the net constructed so far. If it is executable, we need to exclude the wrong continuation from the behavior of the net. This must be done with a feasible place, i.e. a compact region. We encode the non-executability of c in an additional inequality for the compact region inequality system. Every solution of this extended system is a region and excludes c. If this system has a solution, we add the related one-place net to our initially constructed set of transitions. If the extended compact region inequality system has no solution, the wrong continuation c cannot be excluded. We assure that the constructed net is a best approximation to H by adding the set of wrong continuations of c to C. Algorithm 1 will terminate because H is finite.

Algorithm 1.

1: **Input**: A set of labeled Hasse diagrams H
2: $(P, T, W, m_0) \leftarrow (\emptyset, T \leftarrow \bigcup_{(V,<,l) \in H} l(V), \emptyset, \emptyset)$
3: $M \leftarrow$ compactRegionInequalitySystem(H)
4: $C \leftarrow$ wrongContinuations(H)
5: **while** $C \neq \emptyset$ **do**
6:     $c \leftarrow C$.remove()
7:     **if** c.isExecutable(P, T, W, m_0) **then**
8:         $M' \leftarrow M$.addInequality(c)
9:         $s \leftarrow M'$.solve()
10:        **if** s.isSolution() **then**
11:            (P, T, W, m_0).add(s.getOnePlaceNet())
12:        **else**
13:            C.addAll(wrongContinuations(c))
14: **return** (P, T, W, m_0)

---

4   Comparison and Experimental Results

In this section, we first compare compact regions to the already existing concepts of transition regions [4,8] and ordinary tokenflow regions [9]. Secondly, we present a runtime experiment comparing all three related synthesis algorithms.

A transition region of a partial language L is based on the set of step sequences L^{step} of L (see [24]). As an example, we depict three of the numerous step sequences of the first labeled Hasse diagram of our coffee brewing process using two operators: composition of steps $+$ and sequential composition of steps $\cdot$:

$[(grind\ beans) + (unlock\ machine)] \cdot [(empty\ strainer) + (clean\ coffee\ pot)]\cdot$
$[(fill\ strainer) + (get\ water\ coffeepot)] \cdot (fill\ kettle) \cdot (assemble\ and\ turn\ on)$

$[(grind\ beans) + (unlock\ machine)] \cdot [(empty\ strainer) + (clean\ coffee\ pot)]\cdot$
$(get\ water\ coffee\ pot) \cdot [(fill\ strainer) + (fill\ kettle)] \cdot (assemble\ and\ turn\ on)$

$(unlock\ machine) \cdot [(grind\ beans) + (empty\ strainer) + (clean\ coffee\ pot)]\cdot$
$(get\ water\ coffee\ pot) \cdot [(fill\ strainer) + (fill\ kettle)] \cdot (assemble\ and\ turn\ on)$

The two events *grind beans* and *unlock coffee machine* are the first step of the first sequence. The second step is *empty strainer* and *clean coffee pot*. The number of maximal step sequences may grow exponentially with the number of events of a labeled Hasse diagram. Even in our small coffee brewing specification, the size of this language is huge. A transition region defines a place and requires that this place can fire every maximal step sequence of the specified partial language.

Definition 10. *Let $S = \{(V_1, <_1, l_1), \ldots, (V_n, <_n, l_n)\}$ be a specification, T be the set of labels of S, and p be a place. Let $L^{step}(S)$ be the language of step sequences of S. A function $r : ((T \times \{p\}) \cup (\{p\} \times T) \cup \{p\}) \to \mathbb{N}$ is a transition region for S iff for all maximal step sequences $(\tau_1 \ldots \tau_n) \in L(S)^{step}$ and all $j \in \{1, \ldots, n\} : r(p) + \sum_{t \in T}((\tau_1 + \ldots + \tau_{j-1})(t) \cdot r(t, p) - (\tau_1 + \ldots + \tau_j)(t) \cdot r(p, t)) \geq 0$ holds.*

We follow Definition 10 to define the transition region inequality system. The number of unknowns is $2 \cdot |T| + 1$, i.e. a place. The inequalities of the inequality system are a subset of the set of conditions of Definition 10. In the transition region inequality system, we discard all constraints that are equal to or less strict than other constraints. Let τ and τ' be two steps such that $\tau' \leq \tau$ holds and let $\tau_1 \ldots \tau_{j-1}$ and $\tau'_1 \ldots \tau'_{k-1}$ be two step sequences such that $\bigcup_{i<j} \tau_i$ and $\bigcup_{i<k} \tau'_i$ share the same multiset of labels. If τ can fire after the occurrence of $\tau_1 \ldots \tau_{j-1}$, the step τ' can fire after the occurrence of $\tau'_1 \ldots \tau'_{k-1}$. Thus, we build the transition region inequality system by merging matching presteps to so-called prefix steps. The number of inequalities of the transition region inequality system is equal to the number of prefix step continuations. A prefix step continuation is a prefix step Γ together with a step τ if there is a matching maximal step sequence $\pi_1 \ldots \pi_n$ of S such that $\Gamma = \bigcup_{i<n} \pi_i$ and $\tau = \pi_n$ holds. Altogether, the number of inequalities is approximately the number of cuts of S. In a worst-case scenario, the number of cuts is exponential in the size of our input. However, if we specify little concurrency, the number of cuts is small. The number of inequalities is, for example, equal to the number of all events if every labeled Hasse diagram is totally ordered. We refer the reader to [8] for a more detailed description of the transition region inequality system.

An ordinary tokenflow region of a partial language is based on the set of labeled partial orders of a specification. Let $S = \{(V_1, <_1, l_1), \ldots, (V_n, <_n, l_n)\}$ be a specification of L, obviously, $S^* = \{(V_1, <_1^*, l_1), \ldots, (V_n, <_n^*, l_n)\}$ specifies L using labeled partial orders. If we input a set of labeled Hasse diagrams we have to calculate the transitive closure in a first step of this algorithm. Figure 3 depicts two Hasse diagrams with a total of 17 arcs. The respective partial orders have 31. In the exceptional case where a Hasse diagram is transitive, both characterizations have the same number of arcs; in most cases, the number of arcs of a partial order may increase quadratic with the number of arcs of a Hasse diagram.

The concept of tokenflows is similar to the concept of compact tokenflows. Yet, ordinary tokenflows directly relate to process nets of Petri nets, whereas compact tokenflows are able to abstract from the history of tokens. In some sense, compact tokenflows rather relate to distributed transition systems than to process nets.

The domain of a tokenflow region r is the partial order of the specification, every event, and a place. Thus, the domain of a tokenflow region includes the domain of a compact region. If we specify little concurrency, the number of arcs of the labeled partial order is quadratic in the number of arcs of the Hasse diagram. The number of conditions of a tokenflow region is equal to the number of conditions of a compact region. Just like for compact regions, we define the tokenflow region inequality system. This system has $\sum_{(V,<,l) \in S}(|V| + |>^*|) + 2 \cdot |T| + 1$ unknowns and $\sum_{(V,<,l) \in S}(2 \cdot |V| + 1)$ inequalities.

Summing up, compared to transition regions and tokenflow regions, the new compact regions define by far the smallest region inequality system for partial languages. Since algorithms have to solve these systems multiple times during the synthesis procedure, compact tokenflows lead to the fastest synthesis algorithms for partial languages.

To support the scenario-based modeling approaches with Hasse diagrams we developed our tool called *MoPeBs eagle owl*. In *MoPeBs* we implement three synthesis algorithms, each using a different concept of regions. *MoPeBs* is a lightweight editor embedding Viptool [10] plug-ins. *MoPeBs* uses the Simplex algorithm of *LpSolve* to handle the occurring region inequality systems (http://lpsolve.sourceforge.net).

Figure 7 depicts a screenshot of *MoPeBs eagle owl*. The main application window can handle, save, and load synthesis projects. We see a list of specified tasks, two files specifying two different Hasse diagrams, *CoffeePot.lpo* and *GlassPot.lpo*, and three files relating to different p/t-net models. The list of all *.lpo*-files is the specification, i.e. the input of the synthesis algorithms. Every file in the list of p/t-net models was synthesized using a different concept of regions. In the bottom left-hand corner of the main application window, there are three buttons: *transition regions*, *tokenflow regions*, and *compact regions*. Every button starts the related synthesis algorithm. The second window of Fig. 7 depicts the MoPeBs editor showing the Hasse diagram of *CoffeePot.lpo*. Of course, *MoPeBs* can edit, save, and load Hasse diagrams.
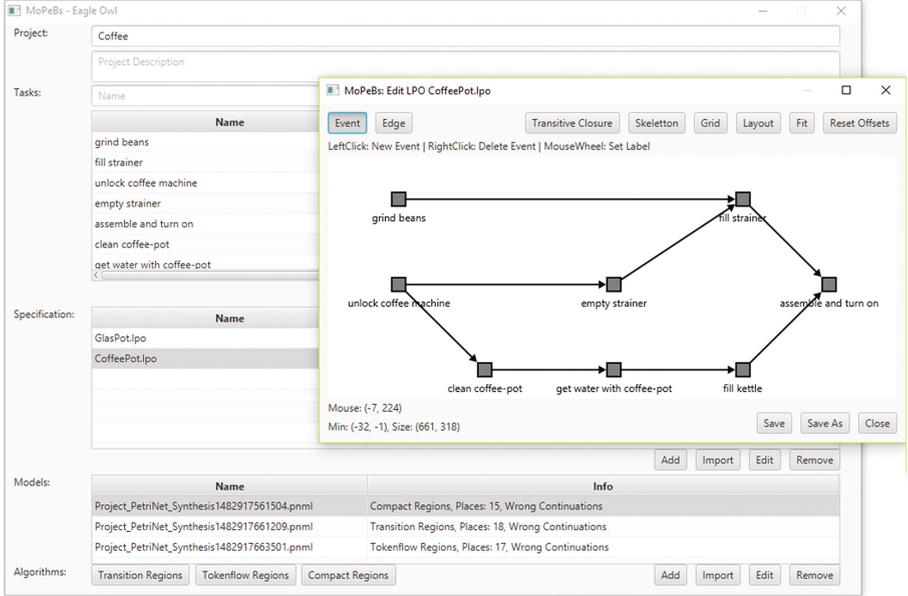
Fig. 7. A screenshot of MoPeBs eagle owl.

We perform the following experiments to measure how well the different synthesis approaches scale with respect to the size of the input and to compare the overall runtime. We use MoPeBs and an Intel Core i5 3.30 GHz (4 CPUs) machine with 8 GB RAM running a Windows 10 operating system. *MoPeBs eagle owl* is available on the *MoPeBs* homepage at www.fernuni-hagen.de/sttp/forschung/mopebs.shtml.

Experiment 1. *We consider five specifications S_1, S_2, S_3, S_4, and S_5. Specification S_1 is the sample specification depicted in Fig. 3. Every other specification is a sequential composition of copies of these two labeled Hasse diagrams. S_2 is the sequential composition of twice the first labeled Hasse diagram and twice the second labeled Hasse diagram. S_3, S_4, and S_5 are three, four, and five copies. We solve the synthesis problem using compact regions, tokenflow regions, and transition regions. We depict the mean of the runtimes of 20 runs of each algorithm in seconds if the algorithm terminates within 15 min in Fig. 8.*

In Experiment 1, the Hasse diagrams grow in length. Specification S_1 has 24 events and 17 arcs, Specification S_5 has 80 events and 105 arcs. Algorithm 1, which uses compact tokenflows, outperforms Algorithm 2 and Algorithm 3 in every test. This is not surprising because only compact regions are tailored to relate to small region inequality systems. As pointed out in the first part of this section, the compact region inequality system is much easier to solve than the tokenflow and the transition region inequality systems. If we compare Algorithm 2 and Algorithm 3, the specifications are rather short at first,
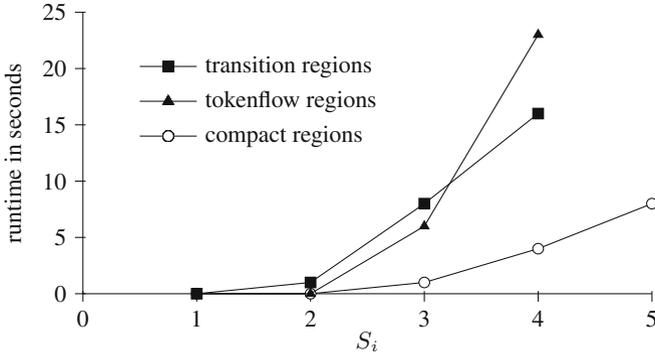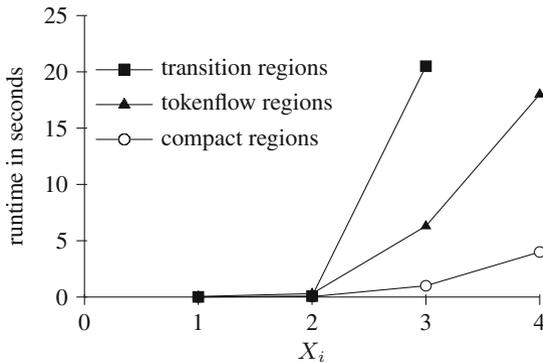
Fig. 8. Runtime results of Experiment 1.

so that the number of cuts is bigger than the number of all (transitive) arcs. The number of inequalities of the transition region inequality system is bigger than the number of unknowns of the tokenflow region inequality system. Thus, ordinary tokenflow regions are faster than transition regions in this example. Specifications S_4 and S_5 have rather little concurrency so that transition regions outperform ordinary tokenflow regions as soon as the number of (transitive) arcs exceeds the number of cuts. Compact regions are fast, independent of the level of concurrency. Considering S_5, only Algorithm 1 is able to solve the synthesis problem within 15 min.

Experiment 2. *We consider four specifications X_1, X_2, X_3, and X_4. Specification X_1 is three Hasse diagrams of the partial language of the so-called repair example from www.processmining.org. Every other specification is a parallel composition of copies of these diagrams, i.e. the specifications grow in width. The maximal size of a cut in X_1 is two, four in X_2, six in X_3, and eight in X_4. We solve the synthesis problem using compact regions, tokenflow regions, and tran-*

Fig. 9. Runtime results of Experiment 2.

sition regions. We depict the mean of the runtimes of 20 runs of each algorithm in seconds if the algorithm terminates within 15 min in Fig. 9.

In Experiment 2, the Hasse diagrams of the specifications grow in width. Specification X_1 has 30 events, X_2 has 45 events, X_3 has 72 events, and X_4 has 90 events. The length of the longest path in every specifications is 8 (at most one loop of the repair example). Just like in Experiment 1, Algorithm 1 outperforms Algorithm 2 and Algorithm 3 in every test. Again, the compact region inequality system is much easier to solve than the tokenflow and the transition region inequality systems. If we compare Algorithm 2 and Algorithm 3, the number of cuts is big and the number of transitive arcs is small. Thus, ordinary tokenflow regions are faster than transition regions in this example. In both experiments compact regions outperform both older algorithms independent from the structure of the specification.

5 Conclusion and Future Work

We presented an approach to generate a process model from a set of Hasse diagrams specifying a set of sample executions. Using our approach, a user can specify a set of executions in a very intuitive and simple specification language and get the complex Petri net model for free.

We presented a new concept of regions for partial languages. The definition is based on the semantics of compact tokenflows. The domain of a compact tokenflow is the number of arcs and the set of initial events of a labeled Hasse diagram. Both numbers grow neither like the number of events nor like the number of arcs of a labeled partial order. We compared compact regions to tokenflow and transition regions referring to the size of the related region inequality systems.

Furthermore, we presented a synthesis algorithm and experimental results of its implementation in the tool called *MoPeBs eagle owl*. *MoPeBs* supports the sample based modelling approach for Hasse Diagramms. We compared the runtime of the new algorithm to the runtime of both existing synthesis algorithms for partial languages.

An important topic for future research will be to develop a concept of wrong continuations using a concept of tokenflows. Right now, the definition of wrong continuations is based on the step language of a partial language. Even though the size of the compact region inequality system is reasonable, the huge number of wrong continuations corrupts the synthesis algorithm if a specification has many concurrent events.

References

1. van der Aalst, W.M.P., Dongen, B.F.: Discovering petri nets from event logs. In: Jensen, K., Aalst, W.M.P., Balbo, G., Koutny, M., Wolf, K. (eds.) Transactions on Petri Nets and Other Models of Concurrency VII. LNCS, vol. 7480, pp. 372–422. Springer, Heidelberg (2013). doi:10.1007/978-3-642-38143-0_10

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), 1128–1142 (2004)

3. van der Aalst, W.M.P.: The application of petri nets to workflow management. J. Circ. Syst. Comput. **8**(1), 21–66 (1998)

4. Badouel, E., Bernardinello, L., Darondeau, P.: Petri Net Synthesis. Texts in Theoretical Computer Science. Springer, Heidelberg (2015)

5. van Beest, N., Dumas, M., Garca-Bauelos, L., La Rosa, M.: Log delta analysis: interpretable differencing of business process event logs. Eprint no. 83018. Queensland University of Technology (2015)

6. Bergenthum, R., Lorenz, R.: Verification of scenarios in petri nets using compact tokenflows. Fundam. Informaticae **137**, 117–142 (2015). IOS Press

7. Bergenthum, R.: Faster verification of partially ordered runs in petri nets using compact tokenflows. In: Colom, J.-M., Desel, J. (eds.) PETRI NETS 2013. LNCS, vol. 7927, pp. 330–348. Springer, Heidelberg (2013). doi:10.1007/978-3-642-38697-8_18

8. Bergenthum, R., Desel, J., Mauser, S.: Comparison of different algorithms to synthesize a petri net from a partial language. In: Jensen, K., Billington, J., Koutny, M. (eds.) Transactions on Petri Nets and Other Models of Concurrency III. LNCS, vol. 5800, pp. 216–243. Springer, Heidelberg (2009). doi:10.1007/978-3-642-04856-2_9

9. Bergenthum, R., Desel, J., Lorenz, R., Mauser, S.: Synthesis of petri nets from finite partial languages. Fundam. Informaticae **88**, 437–468 (2008). IOS Press

10. Bergenthum, R., Desel, J., Lorenz, R., Mauser, S.: Synthesis of petri nets from scenarios with viptool. In: Hee, K.M., Valk, R. (eds.) PETRI NETS 2008. LNCS, vol. 5062, pp. 388–398. Springer, Heidelberg (2008). doi:10.1007/978-3-540-68746-7_25

11. Bergenthum, R., Desel, J., Lorenz, R., Mauser, S.: Process mining based on regions of languages. In: Alonso, G., Dadam, P., Rosemann, M. (eds.) BPM 2007. LNCS, vol. 4714, pp. 375–383. Springer, Heidelberg (2007). doi:10.1007/978-3-540-75183-0_27

12. Carmona, J.: Projection approaches to process mining using region-based techniques. Data Min. Knowl. Discov. **24**(1), 218–246 (2012)

13. Carmona, J., Cortadella, J., Kishinevsky, M.: New region-based algorithms for deriving bounded petri nets. IEEE Trans. Comput. **59**(3), 371–384 (2010)

14. Carmona, J., Cortadella, J., Kishinevsky, M.: Genet: a tool for the synthesis and mining of petri nets. Appl. Concurrency Syst. Des. **2009**, 181–185 (2009)

15. Desel, J., Juhás, G., Lorenz, R., Neumair, C.: Modelling and validation with viptool. In: Aalst, W.M.P., Weske, M. (eds.) BPM 2003. LNCS, vol. 2678, pp. 380–389. Springer, Heidelberg (2003). doi:10.1007/3-540-44895-0_26

16. Desel, J., Erwin, T.: Quantitative engineering of business processes with *VIPbusiness*. In: Ehrig, H., Reisig, W., Rozenberg, G., Weber, H. (eds.) Petri Net Technology for Communication-Based Systems. LNCS, vol. 2472, pp. 219–242. Springer, Heidelberg (2003). doi:10.1007/978-3-540-40022-6_11

17. Desel, J., Juhás, G.: "What is a petri net?" Informal answers for the informed reader. In: Ehrig, H., Padberg, J., Juhás, G., Rozenberg, G. (eds.) Unifying Petri Nets. LNCS, vol. 2128, pp. 1–25. Springer, Heidelberg (2001). doi:10.1007/3-540-45541-8_1

18. Desel, J., Reisig, W.: Place/transition Petri nets. In: Reisig, W., Rozenberg, G. (eds.) ACPN 1996. LNCS, vol. 1491, pp. 122–173. Springer, Heidelberg (1998). doi:10.1007/3-540-65306-6_15

19. van Dongen, B.F., Medeiros, A.K.A., Verbeek, H.M.W., Weijters, A.J.M.M., Aalst, W.M.P.: The ProM framework: a new era in process mining tool support. In: Ciardo, G., Darondeau, P. (eds.) ICATPN 2005. LNCS, vol. 3536, pp. 444–454. Springer, Heidelberg (2005). doi:10.1007/11494744_25

20. Dumas, M., García-Bañuelos, L.: Process mining reloaded: event structures as a unified representation of process models and event logs. In: Devillers, R., Valmari, A. (eds.) PETRI NETS 2015. LNCS, vol. 9115, pp. 33–48. Springer, Cham (2015). doi:10.1007/978-3-319-19488-2_2

21. Ehrenfeucht, A., Rozenberg, G.: Partial (set) 2-structures. part i: basic notions and the representation problem, part ii: state spaces of concurrent systems. Acta Inf. **27**(4), 315–368 (1990)

22. Fahland, D.: Scenario-based process modeling with Greta. In: BPM Demonstration Track 2010, vol. 615. CEUR (2010)

23. Fahland, D.: Oclets – scenario-based modeling with Petri nets. In: Franceschinis, G., Wolf, K. (eds.) PETRI NETS 2009. LNCS, vol. 5606, pp. 223–242. Springer, Heidelberg (2009). doi:10.1007/978-3-642-02424-5_14

24. Grabowski, J.: On partial languages. Fundam. Informaticae **4**, 427–498 (1981). IOS Press

25. Goltz, U., Reisig, W.: Processes of place/transition-nets. In: Diaz, J. (ed.) ICALP 1983. LNCS, vol. 154, pp. 264–277. Springer, Heidelberg (1983). doi:10.1007/BFb0036914

26. Juhás, G., Lorenz, R., Desel, J.: Can I execute my scenario in your net? In: Ciardo, G., Darondeau, P. (eds.) ICATPN 2005. LNCS, vol. 3536, pp. 289–308. Springer, Heidelberg (2005). doi:10.1007/11494744_17

27. Lorenz, R., Juhás, G., Bergenthum, R., Desel, J., Mauser, S.: Executability of scenarios in Petri nets. Theoret. Comput. Sci. **410**(12–13), 1190–1216 (2009). Elsevier

28. Mayr, H.C., Kop, C., Esberger, D.: Business process modeling and requirements modeling. In: ICDS 2007, pp. 8–14. IEEE Computer Society (2007)

29. Peterson, J.L.: Petri Net Theory and the Modeling of Systems. Prentice-Hall, Englewood Cliffs (1981)

30. Reisig, W.: Understanding Petri Nets - Modeling Techniques, Analysis Methods, Case Studies. Springer, Heidelberg (2013)

31. Solé, M., Carmona, J.: Region-based foldings in process discovery. IEEE Trans. Knowl. Data Eng. **25**(1), 192–205 (2013)

32. van Zelst, S.J., van Dongen, B.F., van der Aalst, W.M.P.: ILP-based process discovery using hybrid regions. In: ATAED 2015, vol. 1371, pp. 47–61. CEUR (2015)