Folding Example Runs to a Workflow Net

Robin Bergenthum, Thomas Irgang, Benjamin Meis

Department of Software Engineering,
FernUniversität in Hagen
{firstname.lastname}@fernuni-hagen.de

Abstract. We present a folding algorithm to construct a business process model from a specification. The process model is a workflow net, i.e. a Petri net with explicit split- and join-transitions and the specification is a set of example runs. Each example run is a labeled partial order of events, and each event relates to the occurrence of an activity of the underlying business process. In contrast to sequentially ordered runs, a partially ordered run includes information about dependencies and independencies of events. Consequently, such a run is a precise and intuitive specification of an execution of a business process [5, 11].
The folding algorithm is based on the algorithm introduced in [1]. This algorithm constructs a process model which is able to execute all example runs of the specification, but may introduce a significant amount of not specified behavior to the business process model. We show how to improve this folding procedure, by adapting ideas known from the theory of regions, in order to restrict additional and not specified behavior of the process model whenever possible.

1 Introduction

Business process management [2–4] aims to identify, supervise and improve business processes within companies. It is essential to adapt existing processes to rapidly changing requirements in order to increase and guarantee corporate success. The basis for every business process management activity is a valid and faithful model of the business process; yet constructing such a model is a challenge.

It is particularly challenging to build a complex process model from scratch. For most applications it is easier to first explore single example runs and set up a formal specification before building a complex model [5–7]. In the literature there are many different approaches to automatically generating a process model from a specification. According to the requirements, several approaches exist using different types of specifications, process modeling languages, and model generation strategies.

Process mining algorithms (see [8] for an overview) provide very good runtime, construct readable models, and take into account that recorded or specified behavior can be incomplete or even faulty. Synthesis algorithms (see for example [9–11]) assume a complete and valid specification and construct a process model representing the specification as precisely as possible. Synthesis algorithms are only applicable for medium-size models, but provide excellent control of the produced model and its behavior. In this paper, we present a process mining algorithm and use strategies common in the area of synthesis to improve the construction of the process model. We will show that our new algorithm is fast and able to provide perfect control of the constructed model.

We consider a specification to be a set of labeled partial orders. A labeled partial order is a partially ordered set of events. An event and its label relate to the occurrence of an activity in the business process. In contrast to a sequence of events, a partial order can express dependencies and independencies of events. A set of labeled partial orders is a precise and intuitive specification of a business process [5, 11].

As an example we consider our coffee brewing process. Figure 1 and Figure 2 depict two labeled partial orders (we omit transitive arcs) specifying two different runs of this process. In Figure 1, we grind beans and switch off the coffee machine. We unlock the machine once it is turned off. We fill the strainer as soon as it is empty. We fetch water from the kitchen using the coffee-pot. This pot is only available after the machine is unlocked. Once the strainer is filled and the water is fetched, we assemble the coffee machine. In Figure 2, we use a glass-pot (instead of the coffee-pot) to fetch water from the kitchen. This activity does not depend on unlocking the coffee machine. We can fetch the water right at the beginning of the process. Figure 1 and Figure 2 depict a complete and intuitive specification of our coffee brewing process.

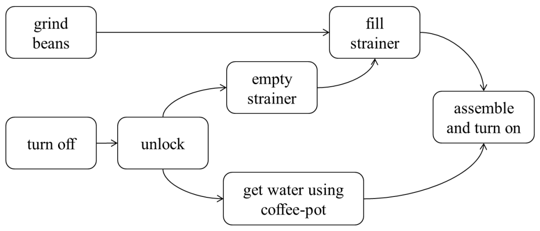


Fig. 1. A Coffee brewing process.

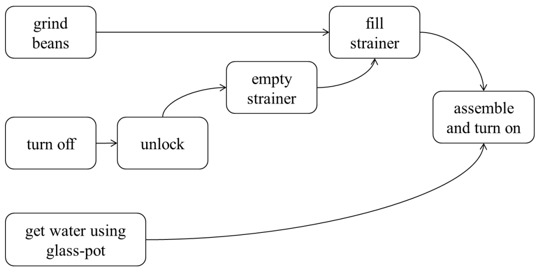


Fig. 2. Another coffee brewing process.

We present an algorithm to construct a workflow net from a specification. As stated above, a specification is a set of labeled partial orders. A workflow net is a Petri net with explicit split- and join-connectors. *and*-splits and *and*-joins duplicate and merge the control flow of a workflow net if actions of the business process occur concurrently.

xor-splits and xor-joins act like switches and steer the course of the control flow of a workflow net if activities of the business process are in conflict.

The new algorithm is based on the folding algorithm presented in [1]. This algorithm adds an initial and a final event to every labeled partial order of the specification and builds a workflow net so that the specification is enabled in this net, i.e. all step sequences of the labeled partial orders of the specification are enabled. The unique start and final events are only necessary for technical purposes and are removed after the folding which leads to a workflow net with a unique start and final place. To build such a net, every labeled partial order is reduced to its underlying Hasse-diagram. A Hasse-diagram of a labeled partial order is the set of events together with the smallest relation so that its transitive closure equals the original partial order. In other words, all transitive arcs are removed to receive a compact and easy to handle representation of the specified example run. We use the Hasse-diagrams of the specification to analyze the neighborhood relation on activities of the business process. For every activity there is a set of equally labeled events. According to the specification, each of these events is enabled by the set of events in its direct preset. Of course, equally labeled events can occur in different contexts. Folding is to arrange actions using splits and joins according to the neighborhood relation present in the Hasse-diagrams of the specification. An xor-split ($\times$) marks exactly one place of its postset, an xor-join ($\times$) needs only one marked place in its preset to be enabled. and-connectors ($\wedge$) use the common Petri net transition semantics.

Folding the specification depicted in Figure 1 and Figure 2 results in the workflow net depicted in Figure 3. Both labeled partial orders of the specification are enabled in this net. For simplification, we omit places between transitions.

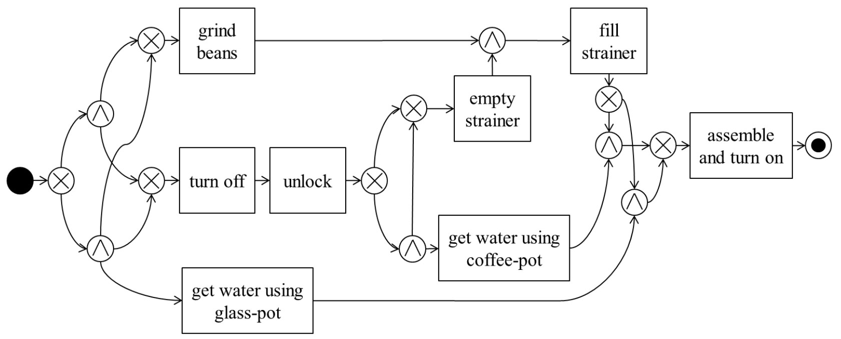


Fig. 3. A business process model of our coffee brewing process.

Folding labeled partial orders is an elegant approach to generating a business process model from a specification. Folding constructs well readable results in very good runtime. Every labeled partial order of the specification is enabled in the generated process model. Unfortunately, folding algorithms tend to introduce additional, not specified behavior to the business process model. Such additional behavior is either a suitable completion of the specification or an inadequate extension yielding an inaccurate business process model. The risk of creating an unfaithful model is due to the fact that the basis for folding is the neighborhood relation introduced by the Hasse-diagrams. Some of the causal structure of the business process may be hidden in the transitive closure of the specified example runs. Transitive dependencies are not considered by the folding algorithm.

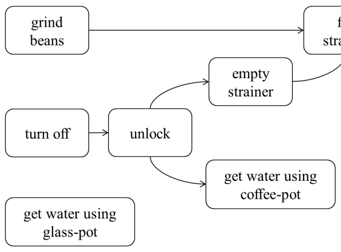


Fig. 4. A run of the business process model depicted in Figure 3.

Figure 4 illustrates additional behavior enabled in the workflow net depicted in Figure 3. In compliance with the diagram in Figure 1, we grind beans, turn the machine off, and get water using the glass-pot right at the beginning. We also unlock the coffee machine and fetch water using the coffee-pot. This behavior is possible in the workflow net, because the alternative between using the glass- and coffee-pot is not reflected by any neighborhood relation of the diagrams depicted in Figure 1 and Figure 2.

In this paper we present a revised folding algorithm. We fold the Hasse-diagrams of the specification into a workflow net. If the workflow net contains additional and inadequate behavior, the revised folding algorithm proceeds to exclude this behavior by changing the workflow net. Nevertheless, all changes lead to a new model which is still able to perform the specified behavior. To improve the model, we use methods known from the theory of synthesis. Each non-specified run of a workflow net has a maximal, specified (not necessarily unique) prefix. Any event ordered after this specified part should not occur at this point. Such specified prefix together with such an event is called a wrong continuation of the workflow net. To eliminate such a wrong continuation from our workflow net, we modify the Hasse-diagrams of our specification. We add a set of transitive arcs to the Hasse-diagrams, so that folding regarding these updated diagrams leads to a workflow net without the wrong continuation. Note, we only add arcs of the transitive closure of the Hasse-diagrams. Thereby, the initial specified behavior is not changed. A more detailed neighborhood relation yields a more restrictive workflow net. We stop if we can not get a more restrictive workflow net by adding transitive arcs. This solution is not unique. It depends on the unfolding and the selected arcs.

We will present the theory and implementation of this revised folding algorithm and we will show that it constructs well readable models. The main advantage of such a folding procedure is that it provides good control of the behavior of the constructed business process model. In an interactive version of our algorithm, it is even possible to distinguish two sets of wrong continuations. The first set is not specified but valid process behavior and extends the specification, the second set is excluded from the model.

The paper is organized as follows: Section 2 defines workflow nets and labeled partial orders. Section 3 presents a folding algorithm. Section 4 presents our new revised folding algorithm and outlines its implementation. Section 5 concludes the paper.

2 Workflow Nets and Labeled Partial Orders

In this paper, a workflow net is a Petri net with additional connector nodes. We consider a special class of these nets where transitions and places do not branch. This class of nets has a very intuitive semantics and is built with elements present in almost every other process modeling language. A workflow net can easily be translated into any business process modeling language such as place/transition nets [12], Event-driven Process Chains (EPCs) [13], Business Process Model and Notation (BPMN) [14], Yet Another Workflow Language (YAWL) [15] or Activity Diagrams (a part of the Unified Modeling Language (UML) [16, 17]).

Definition 1. *A workflow net structure is a tuple $wn = (T, P, C_{xor}, C_{and}, F)$ where T is a finite set of transitions, P is a finite set of places, C_{xor} resp. C_{and} are finite sets of xor- resp. and-connectors, and $F \subseteq ((T \cup C_{xor} \cup C_{and}) \times P) \cup (P \times (T \cup C_{xor} \cup C_{and}))$ is a set of directed arcs connecting transitions and connectors to places and vice versa. A workflow net structure is a workflow net if:*

(i) There is one place, called initial place, having one outgoing and no incoming arc. There is one place, called final place, having one incoming and no outgoing arc. All other places have one incoming and one outgoing arc.
(ii) Transitions have one incoming and one outgoing arc.
(iii) Connectors have either one incoming and multiple outgoing arcs, or multiple incoming and one outgoing arc.

Figure 3 depicts a workflow net. For the sake of clarity, places are hidden in this figure. Let $n \in \{T \cup C_{xor} \cup C_{and}\}$ be a node. We call $\bullet n := \{p \in P \mid (p, n) \in F\}$ the preset of n. We call $n\bullet := \{p \in P \mid (n, p) \in F\}$ the postset of n. A marking of a workflow net assigns tokens to places.

Definition 2. *Let $w = (T, P, C_{xor}, C_{and}, F)$ be a workflow net. A marking of w is a function $m : P \to \mathbb{N}_0$. A pair (w, m) is called marked workflow net. A place $p \in P$ is called marked if $m(p) > 0$, marked by one if $m(p) = 1$ and unmarked if $m(p) = 0$ holds. The initial marking m_0 of a workflow net is defined as follows: The initial place is marked by one and all other places are unmarked.*

There is a simple firing rule for workflow nets. A node is enabled to fire if every place in its preset is marked. An xor-join is also enabled if there is at least one marked place in its preset.

Definition 3. *Let $wn = (T, P, C_{xor}, C_{and}, F, m)$ be a marked workflow net. A node $n \in \{T \cup C_{and}\}$ is enabled if every place in $\bullet n$ is marked. A node $n \in C_{xor}$ is enabled* *if there is a marked place in $\bullet n$. If a node is enabled, it can fire, changing the marking of the workflow net. Firing $n \in (T \cup C_{and})$ leads to the marking m' defined by:*

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

If a node $n \in C_{xor}$ is enabled, choose a marked place $p_{in} \in \bullet n$ and a place $p_{out} \in n\bullet$. Firing n leads to the marking m' defined by:

$$m'(p) = \begin{cases} m(p) - 1, p_{in} = p \neq p_{out} \\ m(p) + 1, p_{out} = p \neq p_{in} \\ m(p) \qquad else. \end{cases}$$

If an enabled node n fires and changes m to m', we write $m[n\rangle m'$.

A set of nodes is called a step. In a workflow net there are no conflicts regarding the consumption of tokens. Consequently, a step is enabled if each node of the step is enabled. Firing a step leads to the same marking as firing all nodes.

Definition 4. *Let $N \subseteq \{T \cup C_{xor} \cup C_{and}\}$ be a step and m be a marking. N is enabled in m if each $n \in N$ is enabled in m. If an enabled step N fires and changes m to m', we write $m[N\rangle m'$. Firing $N = \{n_1, \ldots, n_n\}$ leads to the same marking as firing all $n \in N$, i.e. $m[n_1\rangle m_1[n_2\rangle \ldots [n_n\rangle m'$.*

Let $\sigma = N_1\, N_2\, \ldots\, N_n$ be a sequence of steps. The sequence σ is enabled in m if there are $m_1, m_2, \ldots, m_n$, so that $m[N_1\rangle m_1[N_2\rangle \ldots [N_n\rangle m_n$ holds. If σ is enabled, we define $\sigma_T^{\emptyset} = (N_1 \cap T)\,(N_2 \cap T)\,\ldots\,(N_n \cap T)$. We omit all empty sets in $\sigma_T^{\emptyset}$ to define σ_T. We call σ_T the transition step sequence of σ.

We call a step N a transition step if $N \subseteq T$. A sequence of transition steps τ is enabled in m if there is an enabled sequence of steps σ' so that $\tau = \sigma'_T$ holds.

In Figure 3, the transition step sequence $\{$*grind beans*, *turn off*$\}\{$*unlock*$\}\{$*empty strainer*, *get water using coffee-pot*$\}\{$*fill strainer*$\}\{$*assemble and turn on*$\}$ is enabled in the initial marking. We use transition step sequences to define enabled labeled partial orders [18, 19].

Definition 5. *Let T be a set of labels, a labeled partial order is a triple $lpo = (V, <, l)$, where V is a finite set of events, $<$ is an irreflexive and transitive binary relation over V, and $l : V \to T$ is a labeling function. We consider labeled partial orders without autoconcurrency, i.e. $e, e' \in V, e \neq e', e \not< e', e' \not< e \Rightarrow l(e) \neq l(e')$.*

The Hasse-diagram of a labeled partial order is $lpo^{\lhd} = (V, \lhd, l)$, where $\lhd$ is the set of skeleton arcs, i.e. $\lhd = \{(v, v') \mid v < v' \wedge \nexists v'' : v < v'' < v'\}$.

Let $lpo = (V, <, l)$ and $lpo' = (V, <', l)$ be labeled partial orders. If $< \subseteq <'$ holds, lpo' is a sequentialisation of lpo. If $V = V_1 \dot\cup \ldots \dot\cup V_n$ and $<' = \bigcup_{i<j} V_i \times V_j$ hold, we call the sequence $l(V_1)\,\ldots\,l(V_n)$ a transition step sequence of lpo.

The sequence $\{$*grind beans*, *turn off*$\}\{$*unlock*$\}\{$*empty strainer*, *get water using coffee-pot*$\}\{$*fill strainer*$\}\{$*assemble and turn on*$\}$ is a transition step sequence of the labeled partial order of Figure 1.

Definition 6. *Let wn be a marked workflow net. A labeled partial order lpo is enabled in wn if all transition step sequences of lpo are enabled in the initial marking of wn.*

Figure 1, Figure 2, and Figure 4 depict Hasse-diagrams of labeled partial orders enabled in the initial marking of the workflow net depicted in Figure 3.

3 Folding Algorithm

We introduce a folding algorithm to construct a workflow net from a specification which is a set of labeled partial orders. Each partial order corresponds to a run of the business process. Events model occurrences of activities, arcs model dependencies between events, and unordered events can occur concurrently. Our revised folding algorithm is based on the folding algorithm presented in [1]. We add an initial and a final event to every labeled partial order, before reducing every order to its Hasse-diagram. From these we deduce a neighborhood relation on the set of labels. We define a set of preceding and succeeding label sets for each label. Every label of the specification can, of course, occur multiple times, even in one labeled partial order.

Definition 7. *Let $lpo = (V, <, l)$ be a labeled partial order, let $(V, \lhd, l)$ be its Hasse-diagram, and let T be a set of labels with $l(V) \subset T$. Let $e \in V$ be an event, denote $pred(e) = \{l(e')|e' \lhd e\}$ the set of preceding labels, and denote $succ(e) = \{l(e')|e \lhd e'\}$ the set of succeeding labels.*

Let L be a set of labeled partial orders. Let $t \in T$ be a label, denote $predset(t) = \{pred(e)|(V, <, l) \in L, e \in V, l(e) = t\}$ the set of preceding label sets, and denote $succset(t) = \{succ(e)|(V, <, l) \in L, e \in V, l(e) = t\}$ the set of succeeding label sets.

To construct a workflow net from a specification, we construct a transition for every label and connect transitions according to the corresponding preceding and succeeding label sets. For every transition we will define a so called building block. The center of each building block is the transition, surrounded by three layers of connectors. Next to the transition is a layer of two *xor*-connectors, because each transition can have multiple preceding and multiple succeeding label sets. For each of these sets, there is an *and*-connector on the second layer, because each set can have multiple labels. If succeeding or preceding label sets share labels, there is a *xor*-connector on the third layer. We define a building block as follows:

Definition 8. *Let L be a set of labeled partial orders and T be its set of labels. For each label $t \in T$ we define a workflow net structure $wn^t = (\{t\}, P^t, C^t_{xor}, C^t_{and}, F^t)$ called building block of t. The sets P^t, C^t_{xor}, and C^t_{and} are defined as follows:*

$$\begin{aligned}C^t_{xor} = \{&xor^t_{pre}, xor^t_{post}\} \cup \{xor^t_{pre,t'}|t' \in X, X \in predset(t)\} \cup \\ &\{xor^t_{post,t'}|t' \in X, X \in succset(t)\},\end{aligned}$$

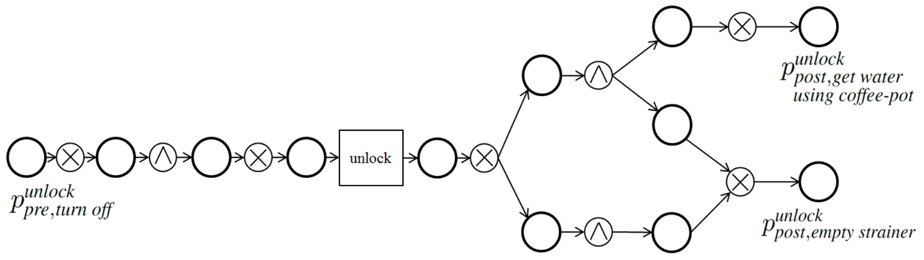
$$C^t_{and} = \{and^t_{pre,X}|X \in predset(t)\} \cup \{and^t_{post,X}|X \in succset(t)\},$$

$$\begin{aligned}P^t = \{&p^t_{pre}, p^t_{post}\} \cup \\ &\{p^t_{pre,X}|X \in predset(t)\} \cup \{p^t_{post,X}|X \in succset(t)\} \cup \\ &\{p^t_{pre,t',X}|t' \in X, X \in predset(t)\} \cup \{p^t_{post,X,t'}|t' \in X, X \in succset(t)\} \cup \\ &\{p^t_{pre,t'}|t' \in X, X \in predset(t)\} \cup \{p^t_{post,t'}|t' \in X, X \in succset(t)\}.\end{aligned}$$

The set of arcs F^t is defined as follows:

$$\begin{aligned}F^t = &\{(xor^t_{pre}, p^t_{pre}), (p^t_{pre}, t), (t, p^t_{post}), (p^t_{post}, xor^t_{post})\} \cup \\ &\{(and^t_{pre,X}, p^t_{pre,X}) | X \in predset(t)\} \cup \\ &\{(p^t_{pre,X}, xor^t_{pre}) | X \in predset(t)\} \cup \\ &\{(xor^t_{post}, p^t_{post,X}) | X \in succset(t)\} \cup \\ &\{(p^t_{post,X}, and^t_{post,X} | X \in succset(t)\} \cup \\ &\{(xor^t_{pre,t'}, p^t_{pre,t',X}) | t' \in X, X \in predset(t)\} \cup \\ &\{(p^t_{pre,t',X}, and^t_{pre,X}) | t' \in X, X \in predset(t)\} \cup \\ &\{(and^t_{post,X}, p^t_{post,X,t'}) | t' \in X, X \in succset(t)\} \cup \\ &\{(p^t_{post,X,t'}, xor^t_{post,t'}) | t' \in X, X \in succset(t)\} \cup \\ &\{(p^t_{pre,t'}, xor^t_{pre,t'}) | t' \in X, X \in predset(t)\} \cup \\ &\{(xor^t_{post,t'}, p^t_{post,t'}) | t' \in X, X \in succset(t)\}.\end{aligned}$$

Figure 5 depicts the building block of label *unlock*. There are two events labeled by *unlock* in Figure 1 and Figure 2. Both events have the same set of preceding labels, i.e. $\{$*turn off*$\}$, but have different sets of succeeding labels. According to these sets there is one *xor*-connector and two *and*-connectors right behind transition *unlock*. Since both sets share the label *empty strainer*, the control flow is joined with $xor^{unlock}_{post,\ empty\ strainer}$ in front of the outgoing interface place $p^{unlock}_{post,\ empty\ strainer}$.



unlock
$p^{unlock}_{pre,turn\ off}$
$p^{unlock}_{post,get\ water\ using\ coffee\text{-}pot}$
$p^{unlock}_{post,empty\ strainer}$

Fig. 5. The building block of label unlock.

We call a building block a compressed building block if all superfluous connectors and the corresponding places are removed. A connector is superfluous if it has one ingoing and one outgoing arc. As an example, Figure 6 depicts the set of compressed building blocks of our coffee brewing process. Note that, before building these blocks, an event labeled with start, and an event labeled with stop, are added to every labeled partial order. In this figure, we depict transition *start* by a big black dot, transition *stop* by a circle with a dot. We hide most places and only sketch interface places by small dots labeled by the corresponding preceding or succeeding labels. On the top left of Figure 6, we depict the building block of label *start*. Next to this block, there is the compressed version of the building block depicted in Figure 5. We merge all compressed building blocks depicted in Figure 6 to get the workflow net depicted in Figure 3.

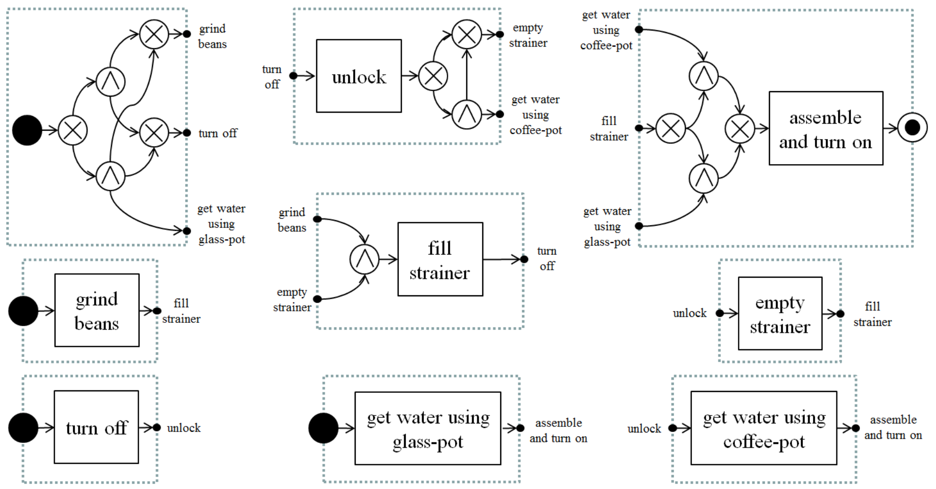


Fig. 6. Compressed building blocks.

Algorithm 1 Folding

```
1:  input: Specification $L$
2:  $L \leftarrow$ add an initial event labeled $start$ to every $lpo \in L$
3:  $L \leftarrow$ add a final event labeled $stop$ to every $lpo \in L$
4:  $H \leftarrow$ Hasse-diagrams of $L$
5:  $T \leftarrow$ Labels of $L$
6:  $B \leftarrow$ Building blocks of $T$
7:  $wn \leftarrow$ Merge all building blocks at matching interface places
8:  $wn \leftarrow$ Delete superfluous connectors from $wn$ by merging the preset and postset places
9:  $wn \leftarrow$ Remove place $p_{pre}^{start}$ and transition $start$
10: $wn \leftarrow$ Mark $p_{post}^{start}$ by one token
11: $wn \leftarrow$ Remove place $p_{post}^{stop}$ and transition $stop$
12: return $wn$
```

Algorithm 1 implements the folding procedure. The input is a set of labeled partial orders. In Line 2 and Line 3 we add two additional events, one labeled *start* and one labeled *stop*, to every labeled partial order. We extend each partial order so that the *start* events are earlier than every other event of their partial order and the *stop* events are later than every other event of their partial order. These new events will result in two additional building blocks responsible for starting and ending runs of the workflow net model. In Line 4, we reduce the specifications to Hasse-diagrams and collect the set of labels (including *start* and *stop*). In Line 6, according to Definition 8, we build a building block for every label and in Line 7, we merge all building blocks at matching interface places, i.e. for every pair of labels we merge all places $p_{post,t'}^{t}$ and $p_{pre,t}^{t'}$. In Line 8, we delete superfluous connectors. In addition, we remove the xor-connector in front of transition *start* and the xor-connector right behind transition *stop*. We remove connectors by merging preset and postset places and we also delete corresponding arcs.

In Line 9, we delete transition *start* and place p_{pre}^{start} in its preset. Thereby, place p_{pre}^{start} becomes the initial place of the workflow net and we mark this place by one token. In Line 11, we remove transition *stop* and place p_{post}^{stop} in its preset to get the final result of the folding procedure. Altogether, Algorithm 1 constructs a workflow net enabled to execute every labeled partial order of the specification. For the proof we refer the reader to [1] but state the following theorem.

Theorem 1. *Let L be a set of labeled partial orders and construct a workflow net wn from L using Algorithm 1. Each labeled partial order $lpo \in L$ is enabled in wn.*

4 Revised Folding Algorithm

In this section we will introduce a revised folding algorithm to construct a workflow net from a specification. Folding is very efficient and generates an intuitive workflow net, but for most examples the workflow net is able to execute additional runs. This is reasonable if the specification is incomplete. However, if we assume that the specification is complete, additional behavior should not be included in the business process model. In the following, we detect and deal with additional behavior introduced within the folding procedure.

During the folding procedure, Hasse-diagrams define sets of preceding and succeeding transitions but sometimes, considering only these dependencies is insufficient. In a business process an early decision can easily determine later alternatives.

We consider Figure 1 and Figure 2 as an example. There is only one event labeled *get water using coffee-pot*. The building block of *get water using coffee-pot* has one preceding label set, i.e. $\{$*unlock*$\}$. According to Figure 2, the transitions *get water using glass-pot*, *turn off*, and *unlock* can occur. This prefix enables *get water using coffee-pot* by the occurrence of *unlock*. This results in the Hasse-diagram depicted in Figure 4.

As stated above, the Hasse-diagrams of the specification define preceding and succeeding label sets to construct building blocks. Considering these sets defined by the transitive relation of the labeled partial orders constructs a workflow net with minimal additional behavior. The occurrence of any transition in this net is conditioned by the occurrence of all transitions corresponding to the complete history of a corresponding event. Obviously, this leads to an unreadable workflow net with a huge number of connectors and arcs.

Our aim is to identify so-called dependency diagrams, a compromise between the Hasse-diagrams and the partial orders, in order to modify the specification thus that additional behavior of a folded model is restricted as far as possible. The fewer dependencies we add, the smaller is the constructed workflow net.

Definition 9. *Let $(V, <, l)$ be a labeled partial order, let $(V, \lhd, l)$ be its Hasse-diagram, and denote T the set of labels. Let (D, E) be a pair of sets of labels, we denote $\lhd^{[D,E]} = \lhd \cup \{(e, e')|e < e', l(e) \in D, l(e') \in E\}$ the dependency relation of $<$ with regard to (D, E). We call $(V, \lhd^{[D,E]}, l)$ the dependency diagram of $(V, <, l)$ with regard to (D, E).*

Of course, $\lhd^{[\emptyset,\emptyset]} = \lhd$ and $\lhd^{[T,T]} =<$, i.e. every dependency diagram is some tradeoff between the Hasse-diagram and the labeled partial order.

Our revised folding algorithm starts by constructing a workflow net from a set of Hasse-diagrams. If this workflow net contains additional behavior, an enabled but not specified partial order is generated. We modify the set of Hasse-diagrams to get a set of dependency diagrams so that folding these diagrams leads to a net that does not enable the additional partial order. We repeat this procedure to get a workflow net that has no additional behavior if such a workflow net exists. Let us now take a closer look at every step of the revised folding algorithm.

We construct the initial workflow net using Algorithm 1. We generate the behavior of this model by an unfolding procedure. We calculate a so-called branching process [21, 20, 22] describing all enabled labeled partial orders. It is easy to calculate such branching processes for workflow nets because they only branch at xor-splits. Moreover, we do not calculate the complete (maybe infinite) behavior of the workflow net but stop generating the behavior as soon as we construct not specified behavior. If there is not specified behavior, there is at least one so-called wrong continuation. A wrong continuation is a labeled partial order enabled in the workflow net and not part of the specification. Removing one event from a wrong continuation yields a specified partial order. Wrong continuations were originally defined in the area of synthesis of Petri nets from step sequences [9] and for synthesizing Petri nets from partial orders [11].

Definition 10. *Let L be a specification and let T be the set of labels. A labeled partial order $(V, <, l) \notin L$ is called a wrong continuation if there is an event $e \in V$ so that $(V \backslash \{e\}, <|_{(V\backslash\{e\})\times(V\backslash\{e\})}, l|_{V\backslash\{e\}}) \in L$ holds.*

We consider Figure 4 as an example. The events *grind beans*, *turn off*, *unlock*, *get water using coffee-pot*, and *get water using glass-pot* form a wrong continuation.

In the last step of the revised folding algorithm, we modify the specification to exclude a wrong continuation. The main idea is to extend the preceding and succeeding label sets appropriately, before restarting the folding procedure.

Definition 11. *Let $L = \{(V_1, <_1, l_1), \ldots, (V_n, <_n, l_n)\}$ be a specification and $L^{\lhd} = \{(V_1, \lhd_1, l_1), \ldots, (V_n, \lhd_n, l_n)\}$ be its set of Hasse-diagrams. Denote T the set of labels. Let $(V_w, <_w, l_w)$ be a wrong continuation and $(V_w, \lhd_w, l_w)$ be its Hasse-diagram.*

Let (D, E) be a pair of label sets and let l, l' be two labels. We call l dependent on l' if for all $(V, <, l) \in L, v \in V, l(v) = l$: $l' \in \{l(v') | v' \lhd^{[D,E]} v\}$. We denote $M^{[D,E]}(l')$ the set of all labels that depend on l'.

(D, E) is called disabling pair of $(V_w, <_w, l_w)$ if one of the following conditions holds:

(a) There is an $e' \in V_w$ so that there is no $(V_i, <_i, l_i) \in L, e \in V_i, l_i(e) = l_w(e')$: $\{l_i(v) | v \lhd_i^{[D,E]} e\} \subseteq \{l_w(v) | v <_w e'\}$ holds.

(b) There is an $e' \in V_w$ so that there is no $(V_i, <_i, l_i) \in L, e \in V_i, l_i(e) = l_w(e')$: $\{l_i(v) | e \lhd_i^{[D,E]} v\} \supseteq \{l_w(v) | e' \lhd_w^{[D,E]} v\} \cap M^{[D,E]}(l_w(e'))$ holds.

A disabling pair (D, E) defines a modification of a specification. This modification yields a set of dependency diagrams. Every dependency diagram includes the Hasse-diagram and extends this diagram by all transitive arcs leading from labels in D to labels in E. We denote the resulting specification by $L^{[D,E]}$.

Theorem 2. *Let L be a specification, lpo_w be a wrong continuation, and (D, E) be a disabling pair of lpo_w. If we construct a workflow net wn from $L^{[D,E]}$ using Algorithm 1, lpo_w is not enabled in wn.*

Proof. Either (a) or (b) of Definition 11 holds.

If (a) holds, there is an event $e' \in V_w$ for which no event $e \in V_i$, $(V_i, <_i, l_i) \in L$, $l_i(e) = l_w(e')$ exists so that $\{l_i(v) | v \lhd_i^{[D,E]} e\} \subseteq \{l_w(v) | v <_w e'\}$ holds.

Algorithm 1 will build and-connectors related to preceding label sets in the building block of $l(e')$. For every and-connector there is a choice of $e \in V_i$, $(V_i, <_i, l_i) \in L$, $l_i(e) = l_w(e')$ so that the and-connector is related to $\{l_i(v) | v \lhd_i^{[D,E]} e\}$. $\{l_i(v) | v \lhd_i^{[D,E]} e\}$ is not included in $\{l_w(v) | v <_w e'\}$. After the occurrence of $\{l_w(v) | v <_w e'\}$ the and-connector is not enabled. The same holds for every other preceding and-connector of the building block of $l_w(e')$. e' can not occur after the occurrence of its prefix. lpo_w is not enabled in wn.

If (b) holds, there is an event $e' \in V_w$ for which no event $e \in V_i$, $(V_i, <_i, l_i) \in L$, $l_i(e) = l_w(e')$ exists so that $\{l_i(v) | e \lhd_i^{[D,E]} v\} \supseteq \{l_w(v) | e' \lhd_w^{[D,E]} v\} \cap M^{[D,E]}(l_w(e'))$ holds.

Algorithm 1 will build and-connectors related to succeeding label sets in the building block of $l(e')$. For every and-connector there is a choice of $e \in V_i$, $(V_i, <_i, l_i) \in L$, $l_i(e) = l_w(e')$ so that the and-connector is related to $\{l_i(v) | e \lhd_i^{[D,E]} v\}$. $\{l_w(v) | e' \lhd_w^{[D,E]} v\} \cap M^{[D,E]}(l_w(e'))$ is not included in $\{l_i(v) | e \lhd_i^{[D,E]} v\}$. The occurrence of this and-connector will not enable all actions in $\{l_w(v) | e' \lhd_w^{[D,E]} v\} \cap M^{[D,E]}(l_w(e'))$, but every such action depends on the occurrence of $l(e')$. The same holds for every other and-connector of the building block $l(e')$. When executing lpo_w in wn there is at least one action missing a token from the building block $l(e')$. lpo_w is not enabled in wn.

Both conditions (a) and (b) suppress the executability of a wrong continuation in a workflow net representing the dependencies introduced from the disabling pair. As an example, we consider the wrong continuation depicted in Figure 4. A disabling pair of label sets is $(\{$*start*$\}, \{$*get water using coffee-pot*$\})$. In the Hasse-diagram depicted in Figure 1, the succeeding label set B_1 of $start$ according to this disabling pair is $\{$*grind beans*, *turn off*, *get water using coffee-pot*$\}$. In other words, *get water using coffee-pot* is added to the original succeeding label set. The succeeding label set B_2 of $start$ of Figure 2 stays unchanged. In Figure 4 the succeeding label set W of $start$ according to the disabling pair is $\{$*grind beans*, *turn off*, *get water using glass-pot*, *get water using coffee-pot*$\}$. This set W is not covered by B_1 or B_2 so that condition (b) of Definition 11 holds. The wrong continuation is not enabled if the additional dependency between *start* and *get water using coffee-pot* is considered when constructing corresponding building blocks. Altogether, if we add one transitive arc to the Hasse-diagram depicted in Figure 1 (from *start* to *get water using coffee-pot*) and apply Algorithm 1, we construct a workflow net which is not able to execute the Hasse-diagram depicted in Figure 4. In this example, the resulting workflow net (depicted in Figure 7) behaves exactly as specified.

Algorithm 2 implements the revised folding procedure. The input is a set of labeled partial orders. In Line 3 and Line 4, we invoke Algorithm 1. While the result of Algorithm 1 has additional behavior, we calculate a wrong continuation in Line 6. We

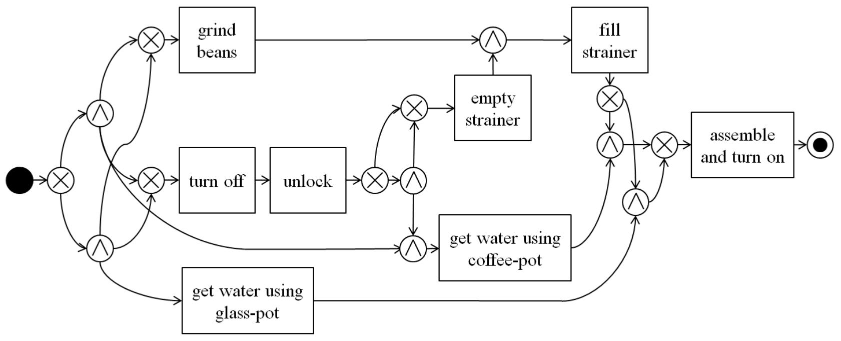


Fig. 7. A business process model of our coffee brewing process.

Algorithm 2 Revised Folding

```
 1: input: Specification $L$
 2: $W \leftarrow \emptyset$
 3: $H \leftarrow$ Hasse-diagrams of $L$
 4: $wn \leftarrow$ Folding of $H$
 5: while $L(wn) \supset (L \cup W)$ do
 6:     $w \leftarrow$ a wrong continuation of $wn$
 7:     if there is a disabling pair $(D, E)$ of $w$ then
 8:         $H \leftarrow$ expand $H$ by all arcs of $L$ in $(D \times E)$
 9:         $wn \leftarrow$ Folding of $H$
10:     else
11:         $W \leftarrow W \cup \{w\}$
12: return $wn$
```

construct a disabling pair (if such a pair exists) or add the wrong continuation to a set W. W contains all wrong continuations which cannot be excluded from a workflow net including the specified behavior. In Line 8, we update the set of Hasse-diagrams by constructing a set of dependency diagrams. We fold again to get a new workflow net still including the specified behavior (and W), but excluding the wrong continuations (Line 9). Just like Algorithm 1, Algorithm 2 constructs a workflow net able to execute every labeled partial order of the specification. Furthermore, Algorithm 2 excludes not specified behavior whenever possible.

The runtime of the new algorithm consists of three parts. It is the sum of the runtime of the folding procedures, the runtime of the unfolding procedures (to check for wrong continuations), and the runtime of the calculations of disabling pairs. Every folding procedure is fast. For every event we compute the preceding and succeeding labels and build the corresponding connectors in the workflow net. The worst case complexity of the unfolding procedure is in exponential time. However, the average runtime, where a workflow net has a reasonable level of concurrent activities, is still fast and is determined by the number of xor-split connectors. The most time consuming part is to find a disabling pair. Altogether, the presented algorithm can be slow, especially if the workflow net has a lot of wrong behavior and describes a lot of concurrency. But in this case, it is possible to stop the algorithm after each iteration and still have a reasonable result.

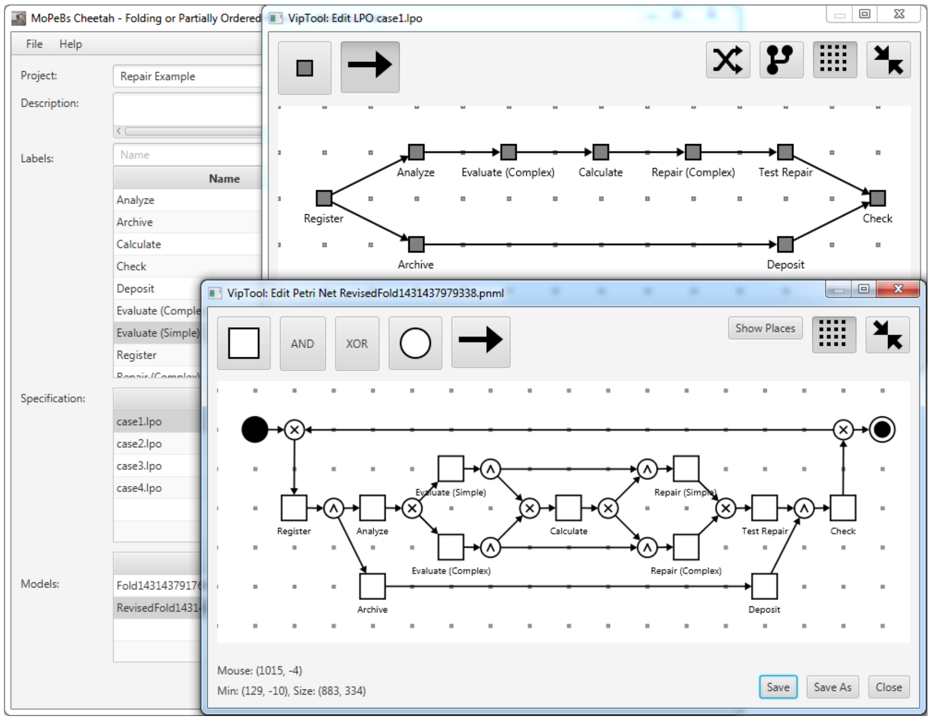


Fig. 8. Screenshot of folding results in MoPeBs

The presented revised folding approach is implemented and available in our tool called MoPeBs Cheetah. MoPeBs Cheetah is a lightweight editor showcasing the revised folding algorithm plug-in of our tool set VipTool [23]. VipTool supports various algorithms related to partially ordered behavior of Petri nets [24, 25]. Figure 8 depicts a screenshot of MoPeBs Cheetah. MoPeBs Cheetah (including examples for the folding algorithm and the revised folding algorithm) is available at https://www.fernuni-hagen.de/sttp/forschung/mopebs.shtml.

5 Conclusion

We recapitulated a folding algorithm to generate a workflow net from a specification. The specification is a set of labeled partial orders. The presented algorithm generates an intuitive model by representing the direct dependencies included in the specification. The generated workflow net is able to execute all specified runs. Moreover, this algorithm usually rounds off the specification, i.e. additional runs which are similar to the specified labeled partial orders are executable in the generated workflow net as well. This is reasonable in cases where the specification is considered to be incomplete.

Reusing the folding algorithm, we introduced a revised folding approach. Starting with an initial model, this iterative approach is able to discover transitive dependencies in the specification which yield a more precise process model. The size of the generated model heavily depends on the number of wrong continuations but for most examples, the generated results are readable as well. The generated workflow net can easily be translated into an EPC, BMPN-model, YAWL-model or an Activity Diagram. These are often used for practical applications. Using an interactive version of the revised folding approach, it is easy to validate the specification while generating a process model. We can add reasonable wrong continuations to the specification while excluding unwanted behavior. All in all, in contrast to other process mining algorithms, the revised folding approach provides perfect control over the language of the generated business process model.

References

[1] Bergenthum, R.; Mauser, S.: Folding Partially Ordered Runs. *Proc. of workshop Application of Region Theory (ART) 2011* (Desel, J.; Yakovlev, A. eds.), CEUR 725, 52–62

[2] Mayr, H. C.; Kop, C.; Esberger, D.: Business Process Modeling and Requirements Modeling. *Proc. of International Conference on the Digital Society (ICDS) 2007*, IEEE, Los Alamitos

[3] Oestereich, B.: Objektorientierte Geschäftsprozessmodellierung und modellgetriebene Softwareentwicklung. *HMD-Praxis Wirtschaftsinformatik 241*, dpunkt.verlag, 27–33

[4] Weske, M.: Business Process Management: Concepts, Languages, Architectures. Springer, Berlin, 2012

[5] Glinz, M.: Improving the quality of requirements with scenarios. *Proc. of World Congress on Software Quality 2000*, JUSE, Yokohama, 55–60

[6] Mayr, H. C.; Kop, C.: A User Centered Approach to Requirements Modeling. *Proc. of Modellierung 2002*, (Glinz, M.; Müller-Luschnat, G. eds.), LNI P-12, 75–86

[7] Desel, J.: From Human Knowledge to Process Models. *Information Systems and e-Business Technologies 2008*, (Kaschek, R.; Kop, C.; Steinberger, C.; Fliedl, G. eds.) LNBIP 5, 84–95

[8] 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 & Knowledge Engineering 47(2)*, (Chen, P. P. ed.), Elsevier, 2003, Philadelphia, 237–267

[9] Darondeau, P.: Region Based Synthesis of P/T-Nets and its Potential Applications. *Proc. of Petri Nets 2000*, (Nielsen, M.; Simpson, D. eds.), LNCS 1825, 16–23

[10] Badouel, E.; Darondeau, P.: Theory of regions. *Lectures on Petri Nets I: Basic Models*, (Reisig, W.; Rozenberg G. eds.), LNCS 1491, 529–586

[11] Bergenthum, R.; Desel, J.; Lorenz, R.; Mauser, S.: Synthesis of Petri Nets from Term Based Representations of Infinite Partial Languages. Fundamenta Informaticae (95), 187–217

[12] Reisig, W.: Petrinetze: Modellierungstechnik, Analysemethoden, Fallstudien. *Leitfäden der Informatik*, Vieweg+Teubner 2010

[13] Scheer, A.-W.: ARIS – Vom Geschäftsprozess zum Anwendungssystem. Springer, Berlin, 2002

[14] White, S. A.: Introduction to BPMN. IBM Cooperation, 2004

[15] van der Aalst, W. M. P.; ter Hofstede, A. H. M.: YAWL: Yet Another Workflow Language. (Shasha, D.; Vossen, G. eds.) *Information Systems 30(4)*, Elsevier, 2005, 245–275

[16] International Organization for Standardization: Information technology – Object Management Group Unified Modeling Language – Part 1: Infrastructure, ISO 19505-1:2012, 2012

[17] International Organization for Standardization: Information technology – Object Management Group Unified Modeling Language – Part 2: Superstructure, ISO 19505-2:2012, 2012

[18] Kiehn, A.: On the Interrelation Between Synchronized and Non-Synchronized Behaviour of Petri Nets. (Dassow, J.; Reichel, B. eds.) *Journal of Information Processing and Cybernetics 24(1-2)*, Otto von Guericke Universität, 1988, 3–18

[19] Vogler, W.: Modular Construction and Partial Order Semantics of Petri Nets. LNCS 625, 1992

[20] Goltz, U.; Reisig, W.: Processes of Place/Transition-Nets. (Diaz, J. ed.) *Automata, Languages and Programming*, LNCS 154, 1983, 264–277

[21] Goltz, U.; Reisig, W.: The Non-Sequential Behaviour of Petri Nets. (Meyer, A. R. ed.) *Information and Control 57(2)*, Elsevier 154, 1983, 125–147

[22] Bergenthum, R.; Mauser, S.; Lorenz, R.; Juhás, G.: Unfolding Semantics of Petri Nets Based on Token Flows. *Information and Control 57(2)*, (Niwiński, D.; Son Nguyen, H. eds.), Fundamenta Informaticae 94(3), 2009, 331–360

[23] Desel, J.; Juhás, G.; Lorenz, R.; Neumair, C.:: Modelling and Validation with VipTool. *Proc. of Business Process Management*, (van der Aalst, W. M. P.; Weske, M. eds.), LNCS 2678, 2003, 380–389

[24] Bergenthum, R.; Mauser, S.: Synthesis of Petri Nets from Infinite Partial Languages with VipTool. *Proc. of workshop Algorithmen und Werkzeuge für Petrinetze 2008*, (Lohmann, N.; Wolf, K. eds.), CEUR 380, 2003, 81–86

[25] Bergenthum, R.; Desel, J.; Juhás, G.; Lorenz, R.: Can I Execute My Scenario In Your Net? VipTool Tells You!. *Proc. of Petri Nets and Other Models of Concurrency 2006*, (Donatelli, S.; Thiagarajan, P.S. eds.), LNCS 4024, 2006, 381–390