Folding Partially Ordered Runs

Robin Bergenthum, Sebastian Mauser

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

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

1 Introduction

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

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

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

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

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

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

2 Partially Ordered Runs of EPCs

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

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

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

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

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

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

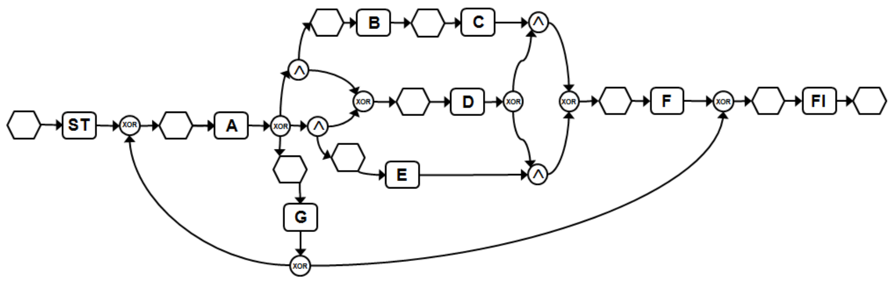

Fig. 1. EPC.

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

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

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

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

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

A marking of an EPC is formally defined as follows.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3 Folding Algorithm

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

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

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

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

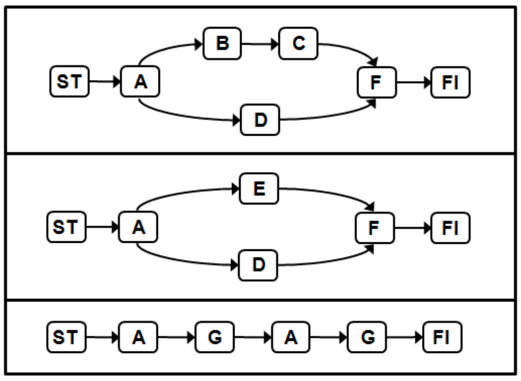

Fig. 2. Set of LPOs.

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

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

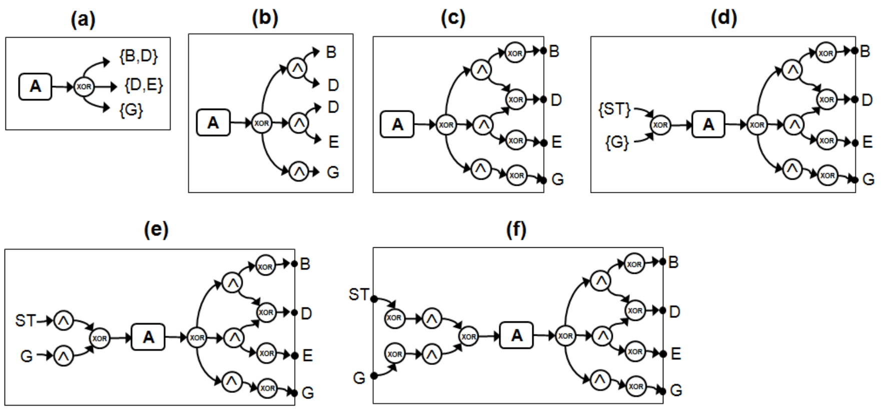

Fig. 3. Construction of a building block.

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

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

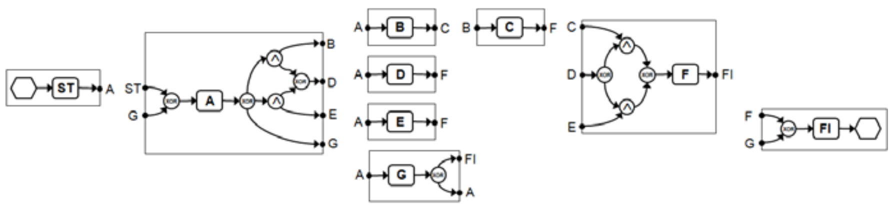

Fig. 4. Building blocks (inadequate connectors are already removed).

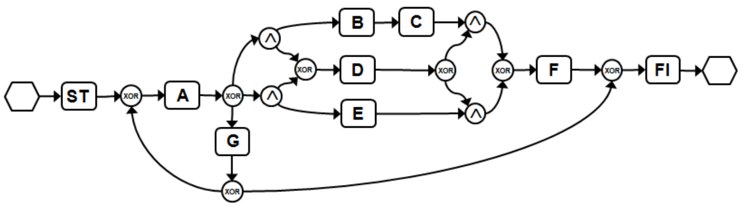

Fig. 5. Resulting EPC.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

4 Conclusion

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

References

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