POWL: Partially Ordered Workflow Language

Humam Kourani and Sebastiaan van Zelst

Fraunhofer FIT - Data Science and Artificial Intelligence, Sankt Augustin, Germany
{humam.kourani,sebastiaan.van.zelst}@fit.fraunhofer.de

Abstract. Process models are used to represent processes in order to support communication and allow for the simulation and analysis of the processes. Many real-life processes naturally define partial orders over the activities they are composed of. Partial orders can be used as a graph-like representation of process behavior. On the one hand, partially ordered graph representations allow us to easily model concurrent and sequential behavior between activities while ensuring simplicity and scalability. On the other hand, partial orders lack the support for typical process constructs such as choice and loop structures. Therefore, in this paper, we present a novel process modeling notation, i.e., the Partially Ordered Workflow Language (POWL). A POWL model is a partially ordered graph extended with control-flow operators for modeling choice and loop structures. A POWL model has a hierarchical structure; i.e, POWL models can be combined into a new model either using a control-flow operator or as a partial order. We propose an initial approach to demonstrate the feasibility of using POWL models for process discovery, and we evaluate our approach based on real-life data.

Keywords: POWL, process modeling, partial order, process tree

1 Introduction

A process model provides an illustration of a process that supports communication and allows for the simulation and analysis of the process. Process models can either be created by hand or discovered using process discovery techniques [2]. Organizations use information systems to track and record data about the execution of their processes, and this data is used for the discovery of process models. Process models might provide insights for organizations and allow them to analyze their processes in order to detect problems and bottlenecks. This can help to automate processes and to make better decisions.

Different modeling notations are used to model processes. *Petri nets* are a powerful modeling notation widely used to formally describe the behavior of processes. A sub-class of Petri nets, called *Workflow nets (WF-nets)*, is usually used to model *business processes*. WF-nets adhere to structural quality requirements; e.g., they define a clear notion for marking the start and end of processes. However, WF-nets may still suffer from behavioral quality issues. For instance, it is possible to construct a Workflow net with dead parts that can never be reached. WF-nets that do not suffer from such quality anomalies are called *sound*.

A *process tree* [16] is a hierarchical modeling notation, i.e., a mathematical tree, in which the leaves represent activities and the internal vertices represent control-flow operators for modeling behavioral dependencies between their children. Process trees represent a strict subset of WF-nets; i.e., any process tree can be transformed into a WF-net modeling the same behavior, but not all WF-nets can be modeled as process trees. Process trees are guaranteed to be sound by construction as they are limited to modeling hierarchical structures.

Partial orders are used as a representation of the execution order of activities for many real-life processes. In a partial order, some activities may have a strict order with respect to each other (e.g., activity "a" must happen before activity "b"), while other activities are concurrent (e.g. activities "b" and "c" may happen in any order). This reflects the reality of many business processes, where there may be multiple ways to accomplish a goal. Several partial-order-based modeling notations have been introduced, e.g., prime event structures [25] and conditional partial order graphs [22]. These notations allow us to model concurrency and sequential dependencies in an efficient and compact manner; however, none of them properly support cyclic process behavior, which is very common in practice. Moreover, in a partial order over activities, we assume all activities to be executed, and thus, modeling a choice is not supported.

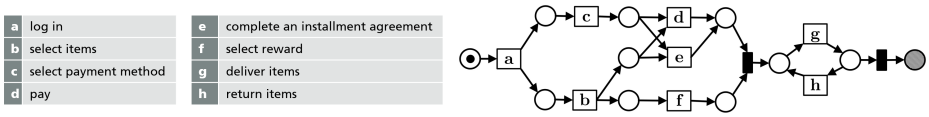
On the one hand, process trees fail to model non-hierarchical dependencies that can be easily described by a partial order. On the other hand, we cannot model loop or choice structures in a partially ordered graph. We propose a new modeling notation that combines hierarchical modeling notations with partial orders. We call our modeling language *Partially Ordered Workflow Language (POWL)*. A POWL model is a partially ordered graph extended with control-flow operators for modeling choice and loop structures; i.e, a POWL model is a hierarchical model that allows for defining partial orders over sub-models.

The remainder of the paper is structured as follows. We start with a motivating example in Section 2. We discuss related work in Section 3, and we briefly present preliminaries in Section 4. We define POWL models in Section 5, and we introduce an initial approach for the discovery of POWL models in Section 6. We evaluate our approach using real-life data in Section 7. Finally, we provide a brief summary of the paper and propose ideas for future work in Section 8.

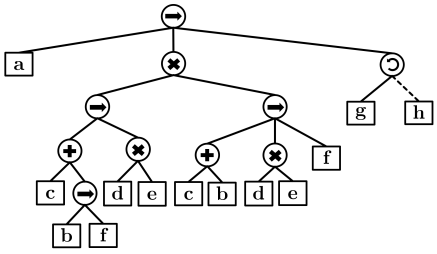
2 Motivation

In this section, we motivate our contribution based on a simple example.

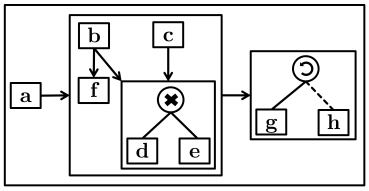
We consider a process for purchasing items from an online shop. The user starts an order by logging in to their account (a). Then, the user simultaneously selects the items to purchase (b) and sets a payment method (c). Afterward, the user either pays (d) or completes an installment agreement (e). After selecting the items, the user chooses between multiple options for a free reward (f). Since the reward value depends on the purchase value, this step is done after selecting the items, but it is independent of the payment activities. Finally, the items are delivered to the user (g). The user may exchange received items. The user can



(a) A labeled WF-net.



(b) A process tree.



(c) A POWL model. For simplicity, we only visualize the transitive reduction of the partial orders in the model.

Fig. 1: Process models.

return some items (h), and each time items are returned, a new delivery is made afterward. The WF-net shown in Figure 1a precisely models this process.

A process tree models hierarchical behavioral structures using the control-flow operators $\rightarrow$, $\times$, $\circlearrowleft$, and $+$. The $\rightarrow$ operator models a sequential execution of blocks; $\times$ models an exclusive choice; $+$ models concurrency; $\circlearrowleft$ models a do-redo loop between two blocks (i.e., the first block is executed once first, and every time the second block is executed it is followed by another execution of the first block). Figure 1b shows a process tree modeling the behavior of our example process. This tree contains a choice of two sub-tree over the same set of activities (b, c, d, e, and f). Process trees are limited to modeling hierarchical structures; i.e., without duplicating activities, a process tree cannot precisely model the dependencies between the activities b, c, d, e, and f.

Figure 1c shows a POWL model precisely modeling the behavior of our example process. The outer layer of the hierarchy is a partial order modeling a sequence between the activity sets $\{a\}$, $\{b, c, d, e, f\}$, and $\{g, h\}$. Another partial order is used to model the non-hierarchical dependencies between the activity sets $\{b\}$, $\{c\}$, $\{d, e\}$, and $\{f\}$. The process tree operators $\times$ and $\circlearrowleft$ are used to model the choice between d and e and the loop between g and h respectively.

Compared to the process tree and the WF-net, the POWL model has a simpler structure with fewer nodes and edges (i.e., no places and no duplication of activities). Moreover, the POWL model shows non-hierarchical dependencies without duplicating activities, while sub-models can still be easily identified in the hierarchy and the soundness guarantee is preserved.

3 Related Work

Different modeling notations are used among process mining tools and techniques. We refer to [3] for an overview of process modeling notations in process

mining. In [16], Leemans introduce the inductive mining framework and multiple process discovery approaches implementing it. The approach we propose for the discovery of POWL models extends the inductive mining framework.

Many ideas for combining different modeling notations have been proposed. In [4], a hybrid Petri net is defined as a Petri net extended with informal arcs connecting transitions. In [27], another type of hybrid process models is defined by combining imperative and declarative modeling languages.

Partial orders are used for data representation and process modeling. An overview of the use of partial orders in process mining is provided in [17]. In [21], Mannila et al. propose an approach for the discovery of frequent episodes, where an episode is defined as a partially ordered set of events. This approach is adapted in [15] to discover partially ordered sets of activities in event logs. In [14], the authors create partially ordered representations of activities and combine them into a workflow graph. In [13], the authors suggest an approach for generating prime event structures from event logs. A prime event structure [25] is a partially ordered graph enriched with a conflict relation. This approach is able to model choice due to the conflict relation; however, loops remain a major challenge for prime event structures. In [22], the authors present a method for deriving conditional partial order graphs from event logs. A conditional partial order graph [23] is a compact representation of a family of partial orders that is able to model choice structures, but it fails to capture cyclic behavior.

In [26], the authors introduce a flow language (BPEL) that allows for combining web service primitives using advanced control-flow constructs (including event handlers). BPEL additionally allows for imposing an execution order over primitives executed in parallel using control links. BPEL is a powerful language for implementing web services; however, BPEL is very complex for end users and can be viewed as a programming language rather than a modeling language [5].

4 Preliminaries

In this section, we present basic preliminaries that ease this paper's readability.

$\mathbb{N}=\{1,2,3,...\}$ denotes the set of natural numbers. We use $\mathbb{N}_{odd}=\{1,3,5,...\}$ to denote the set of odd numbers, and we use $\mathbb{N}_{even}=\{2,4,6,...\}$ to denote the set of even numbers.

$\mathcal{P}(X)=\{X'\subseteq X\}$ denotes the powerset of a set X. For n sets $X_1,...,X_n$, we define the *n-ary Cartesian product* as the set $X_1\times...\times X_n=\{(x_1,...,x_n) \mid x_i\in X_i \text{ for } 1\leq i\leq n\}$. An *n-ary relation* over $X_1,...,X_n$ is a subset of the n-ary Cartesian product $X_1\times...\times X_n$.

Let X and Y be two sets, and $f\colon X\to Y$ be a function. f is *injective* if $\underset{x,x'\in X}{\forall} f(x)=f(x')\Rightarrow x=x'$. f is *surjective* if $\underset{y\in Y}{\forall}\ \underset{x\in X}{\exists} f(x)=y$. f is *bijective* if it is injective and surjective. We use $\mathcal{B}(X,Y)$ to denote the set of all bijective functions from X to Y. A *multi-set* generalizes the notion of a set and allows for multiple occurrences of the same element. The order of occurrences of the elements in a multi-set is irrelevant. We define a multi-set M over a set X as a function $M\colon X\to\mathbb{N}\cup\{0\}$. We write a multi-set as $M=[{x_1}^{c_1},...,{x_n}^{c_n}]$ where

$M(x_i)=c_i$ for $1\leq i\leq n$ (for $x\in X$ with $M(x)=1$, we omit the superscript; in case $M(x)=0$, we omit x). We use $\mathcal{M}(X)$ to denote the set of all multi-sets over X.

A *sequence* is an ordered collection of elements. We define a sequence over a set X as a function $\sigma\colon \{1,\ldots,n\}\to X$, and we write $\sigma=\langle\sigma(1),...,\sigma(n)\rangle$. We use $|\sigma|=n$ to denote the length of σ and X^* to denote the set of all sequences over X. We use $\sigma_1\cdot\sigma_2$ to denote the *concatenation* of two sequences σ_1 and σ_2, e.g., $\langle x_1\rangle\cdot\langle x_2,x_1\rangle=\langle x_1,x_2,x_1\rangle$. We overload notation and, for two sets of sequences L_1 and L_2, we write $L_1\cdot L_2=\{\sigma_1\cdot\sigma_2 \mid \sigma_1\in L_1\wedge\sigma_2\in L_2\}$. We use $\sigma{\uparrow_Y}$ to denote the *projection* of a sequence σ on a set Y. For example, $\langle x_1,x_2,x_1\rangle{\uparrow_{\{x_1,x_3\}}}=\langle x_1,x_1\rangle$.

Let $\prec\subseteq X\times X$ be a 2-ary relation over a set X. For $(x_1,x_2)\in X\times X$, we write $x_1\prec x_2$ to denote that $(x_1,x_2)\in\prec$, and we write $x_1\not\prec x_2$ to denote that $(x_1,x_2)\notin\prec$. $\prec$ is a *strict partial order* if it is *irreflexive* (i.e., $x\not\prec x$ for all $x\in X$) and *transitive* (i.e., if $x_1\prec x_2$ and $x_2\prec x_3$, then $x_1\prec x_3$)[1]. In the remainder of the paper, we use the term *partial order* to refer to a strict partial order. We refer to $\rho=(X,\prec)$ as a *partially ordered set (poset)*. The language of ρ is defined as the set of sequences $\mathcal{L}(\rho)=\{\sigma\in X^* \mid \exists_{f\in\mathcal{B}(\{1,\ldots,|\sigma|\},X)} \forall_{1\leq i\leq|\sigma|} \sigma(i)=f(i)\wedge \forall_{1\leq j\leq|\sigma|} f(i)\prec f(j)\Rightarrow i<j\}$. The *transitive reduction* of $\prec$ is defined as $\prec^-=\{(x_1,x_3)\in\prec \mid \nexists_{x_2\in X} x_1\prec x_2\wedge x_2\prec x_3\}$. We use $\mathit{\Pi}(X)$ to denote the set of all posets over X. Let X and Y be two sets, $\rho=(X,\prec)$ be a poset, and $\gamma\colon X\to Y$ be a *labeling* function. The triple $\rho'=(X,\prec,\gamma)$ is called a *labeled partial order* over X and Y. The language of ρ' is defined as the set of sequences $\mathcal{L}(\rho')=\{\langle\gamma(\sigma(1)),...,\gamma(\sigma(|\sigma|))\rangle \mid \sigma\in\mathcal{L}(\rho)\}$. We use $\mathit{\Pi}(X,Y)$ to denote the set of all labeled partial orders over X and Y.

We use Σ to denote the universe of activities, and we use $\tau\notin\Sigma$ to denote the *silent activity* (τ is also referred to as the *unobservable activity*).

5 POWL Language

In this section, we introduce the Partially Ordered Workflow Language (POWL). We define POWL models, their semantics, and an approach for transforming POWL models into sound WF-nets.

A POWL model is a partially ordered graph representation of a process, extended with control-flow operators for modeling choice and loop structures. We define three types of POWL models. The first type is the *base case* consisting of a single activity. For the second type, we use the existing process tree operators $\times$ and $\circlearrowleft$ (defined in [16]) to combine multiple POWL models into a new model. We use the operator $\times$ to model an exclusive choice of $n\geq 2$ POWL models and the operator $\circlearrowleft$ to model a do-redo loop of two POWL models. The third type of POWL models is defined as a poset of $n\geq 2$ POWL models. We interpret unconnected nodes in a poset to be concurrent and connections between nodes as sequential dependencies. Figure 1c shows an example POWL model.

Definition 1 (POWL Model). *A POWL model is recursively defined as follows:*

[1] Irreflexivity and transitivity imply *asymmetry*; i.e., if $x_1\prec x_2$, then $x_2\not\prec x_1$.

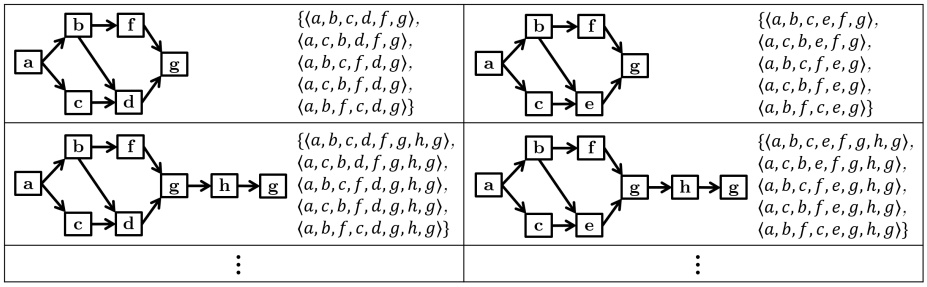


Fig. 2: Translation of the POWL model shown in Figure 1c into a set of labeled partial orders. For simplicity, we only show the labels of the transitions. We also show the language of each partial order (i.e., as a set of activity sequences).

- *Any activity* $a \in \Sigma \cup \{\tau\}$ *is a POWL model.*
- *Let* ψ_1 *and* ψ_2 *be two POWL models.* $\circlearrowleft(\psi_1, \psi_2)$ *is a POWL model.*
- *Let* $P = \{\psi_1, ..., \psi_n\}$ *be a set of* $n \geq 2$ *POWL models.*
  - $\times(\psi_1, ..., \psi_n)$ *is a POWL model.*
  - *A poset* $\rho = (P, \prec) \in \Pi(P)$ *is a POWL model.*

We use Ψ to denote the universe of POWL models. We define the execution semantics for POWL models. Since a partially ordered set of POWL models is a POWL model, we define the semantics of POWL models in terms of partial orders as well. However, choice and loop structures cannot be described using a single partial order. Hence, we define the semantics of a POWL model by transforming it into a set of labeled partial orders over a set of newly generated nodes (we call them *transitions*), and we use activities as labels. Figure 2 shows the result of applying this transformation on the POWL model shown in Figure 1c.

For the base case (i.e., a single activity), the POWL model is transformed into a single partial order with a transition having the corresponding activity as a label. For a silent activity, we create an empty labeled partial order. For the operator $\times$, the language is defined as the union of the languages of the sub-models. For the operator $\circlearrowleft$, we combine labeled partial orders from the languages of the sub-models such that the first order is from the do-part and each order from the redo-part is followed by an order from do-part. When combining these orders, we replace every transition from the orders of the languages of the sub-models by a new transition having the same label. For a poset of POWL models, labeled partial orders are generated by combining orders from the languages of the sub-models such that the partial order of the sub-models is preserved.

Definition 2 (Partial Order Semantics). *Let* $\mathcal{T}$ *be the universe of transitions.* $\Gamma : \Psi \to \mathcal{P}(\Pi(\mathcal{T}, \Sigma))$ *is a function recursively defined to transform a POWL model into a set of labeled partial orders as follows.*

- *For* $a \in \Sigma$, $\Gamma(a) = \{(\{t\}, \emptyset, (t, a))\}$ *where* $t \in \mathcal{T}$ *is a new transition.*

- $\Gamma(\tau)=\{(\emptyset,\emptyset,\emptyset)\}$.
- *Let* $P=\{\psi_1,...,\psi_n\}$ *be a set of* $n\geq 2$ *POWL models.*
  - $\Gamma(\times(\psi_1,...,\psi_n))=\bigcup_{1\leq i\leq n}\Gamma(\psi_i)$.
  - $\Gamma(\circlearrowleft(\psi_1,\psi_2))=$
    $\bigcup_{n\in\mathbb{N}_{odd}}\{(T,\prec,\gamma)\mid \exists_{(T_1,\prec_1,\gamma_1)\in\Gamma(\psi_{\tilde{1}}),...,(T_n,\prec_n,\gamma_n)\in\Gamma(\psi_{\tilde{n}}),f\in\mathcal{B}(\bigcup_{1\leq i\leq n}T_i,T)}$
    $\forall_{1\leq i\leq n,t_i\in T_i}\Big(\gamma_{\tilde{i}}(t_i)=\gamma(f(t_i))$
    $\wedge\ \forall_{1\leq j\leq n,t_j\in T_j} f(t_i)\prec f(t_j)\Leftrightarrow((i=j\wedge t_i\prec_i t_j)\vee i<j)\Big)\}$
    where $T\subseteq\mathcal{T}$ *refers to a set of new transitions and* $\tilde{i}$ *refers to the transformation of an index* $i\in\mathbb{N}$ *defined as:* $\tilde{i}=\begin{cases}1 & \text{if } i\in\mathbb{N}_{odd},\\ 2 & \text{if } i\in\mathbb{N}_{even}.\end{cases}$
  - *For a poset* $\rho=(P,\prec)$, $\Gamma(\rho)=$
    $\{(T,\prec',\gamma)\mid \exists_{(T_1,\prec_1,\gamma_1)\in\Gamma(\psi_1),...,(T_n,\prec_n,\gamma_n)\in\Gamma(\psi_n)} T=\bigcup_{1\leq i\leq n}T_i$
    $\wedge\ \forall_{1\leq i\leq n,t_i\in T_i}\Big(\gamma(t_i)=\gamma_i(t_i)$
    $\wedge\ \forall_{1\leq j\leq n,t_j\in T_j} t_i\prec' t_j\Leftrightarrow((i=j\wedge t_i\prec_i t_j)\vee\psi_i\prec\psi_j)\Big)\}$.

After transforming a POWL model into a set of labeled partial orders, we can derive the set of activity sequences that can be generated by the model. We overload notation by defining the language of a POWL model $\psi\in\Psi$ as the set of activity sequences $\mathcal{L}(\psi)=\{\sigma\in\Sigma^*\mid \exists_{\rho=(T,\prec,\gamma)\in\Gamma(\psi)}\sigma\in\mathcal{L}(\rho)\}$.

Similar to process trees, POWL models can be recursively transformed into WF-nets. The transformation approach is schematically presented in Figure 3. The generated workflow net is guaranteed to be sound; the soundness can be proven by the composition theorem ([1, Theorem 3]).

6 Discovery of POWL Models

In this section, we demonstrate the feasibility of using POWL models in process discovery by extending the base inductive miner [16] to mine for POWL models.

6.1 Event Log

Organizations use information systems to track and record information about the execution of their processes. Data can be stored in different forms. In process discovery, we assume data to be provided in the form of an *event log*. We define an *event log* $L\in\mathcal{M}(\Sigma^*)$ as a multi-set of activity sequences. A *trace* $\sigma\in L$ is a sequence of activities that represents the execution of a single process instance.

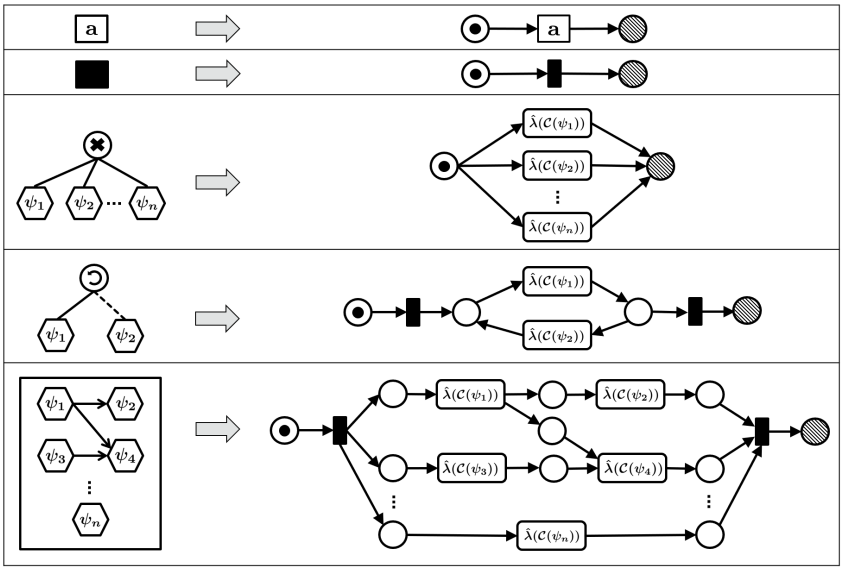


Fig. 3: POWL to WF-net converter $\mathcal{C}$. For a WF-net W, we use $\hat{\lambda}(N)$ to denote the Petri net that results after removing the source and sink places of W.

Let $L \in \mathcal{M}(\Sigma^*)$ be an event log. $\Sigma_L = \{a \in \Sigma \mid \exists_{\sigma \in L, 1 \leq i \leq |\sigma|} \sigma(i) = a\}$ denotes the set of activities that occur in L. We use $L_{\triangleright} = \{a \in \Sigma_L \mid \exists_{\sigma \in L} \sigma(1) = a\}$ to denote the set of *start activities* and $L_{\square} = \{a \in \Sigma_L \mid \exists_{\sigma \in L} \sigma(|\sigma|) = a\}$ to denote the set of *end activities*. The *directly-follows graph (DFG)* is a 2-ary relation $\mapsto_L \subseteq \Sigma_L \times \Sigma_L$ that captures direct successions between activities; i.e., $a \mapsto_L b$ iff $\exists_{\sigma \in L, 1 \leq i < |\sigma|} \sigma(i) = a \wedge \sigma(i+1) = b$. The *eventually-follows graph (EFG)* $\rightsquigarrow_L \subseteq \Sigma_L \times \Sigma_L$ captures direct and indirect successions between activities; i.e., $a \rightsquigarrow_L b$ iff $\exists_{\sigma \in L, 1 \leq i < j \leq |\sigma|} \sigma(i) = a \wedge \sigma(j) = b$.

$L_1 = [\langle a, b, c \rangle^3, \langle a, b, d \rangle^2]$ is an example event log. This event log consists of five traces with $\Sigma_{L_1} = \{a, b, c, d\}$, $L_{1\triangleright} = \{a\}$, $L_{1\square} = \{c, d\}$, $\mapsto_{L_1} = \{(a, b), (b, c), (b, d)\}$, and $\rightsquigarrow_{L_1} = \{(a, b), (a, c), (a, d), (b, c), (b, d)\}$.

6.2 Inductive Miner

The inductive miner [16] is one of the leading approaches in process discovery. It provides formal guarantees such as soundness, perfect fitness (i.e., it discovers a model that covers all behavior recorded in the log), and rediscoverability of certain process structures. There are several variants of the inductive miner (e.g., for handling incompleteness or infrequent behavior). In this paper, we extend the base variant of the inductive miner that assumes a noise-free event log and returns a model that perfectly fits the input event log.

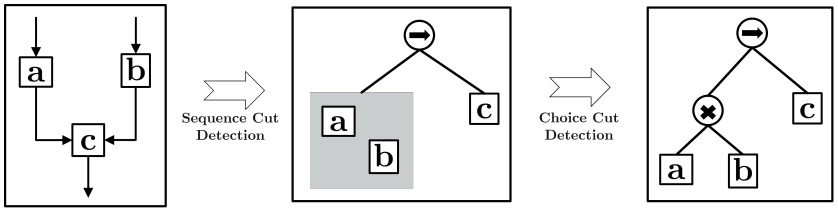


Fig. 4: Two steps of process tree cut detection: $\rightarrow$ and $\times$ cuts.

The inductive miner is a recursive top-down approach. The algorithm tries to detect a *cut*, i.e., it tries to detect a behavioral pattern in the directly-follows graph and a partitioning of the activities according to this pattern. The inductive miner supports four cuts corresponding to the four operators of process trees, and it recursively generates a process tree based on the detected cuts.

Definition 3 (Process Tree Cut). *Let $L \in \mathcal{M}(\Sigma^*)$ be an event log. A process tree cut $(\oplus, A_1, ..., A_n)$ of L is tuple of a control-flow operator $\oplus \in \{\rightarrow, \times, +, \circlearrowleft\}$ and a partitioning of the activities into $n \geq 2$ subsets; i.e., $\Sigma_L = A_1 \cup ... \cup A_n$ and $A_i \cap A_j = \emptyset$ for $1 \leq i < j \leq n$.*

After detecting a cut, the event log is projected into the different groups of the partitioning, creating several sub-logs. The same approach is then recursively applied to all sub-logs until a *base case* of the recursion is reached. A base case is defined as an event log whose activity set consists of a single activity. A base case can be easily transformed into a process tree: either into a single node or using the operators $\times$ or $\circlearrowleft$ to model an optional activity or a self-loop.

For the formal description of the different steps of the inductive miner, we refer to [16]. Figure 4 shows an example directly-follows graph and two steps of process tree cut detection based on it. First, a sequence cut $(\rightarrow, \{a, b\}, \{c\})$ is detected in the initial directly-follows graph; i.e., a sequential dependency between these groups of activities is discovered. The event log is then projected into the two groups of activities, creating sub-logs. The second sub-log is a base case. For the first sub-log, a choice cut $(\times, \{a\}, \{b\})$ is detected, and again, two sub-logs are generated. Both sub-logs are base cases, and the algorithm terminates returning the process tree $\rightarrow(\times(a, b), c)$.

If neither a base case nor a cut is detected, then the inductive miner invokes a *fall-through function*. This function always returns a cut that might correspond to an under-fitting model (i.e., a model that does not precisely capture the behavior recorded in the log), but it allows for continuing the recursion. For example, the fall-through function might return a concurrency cut between an activity that occurs exactly once in every trace and the rest of the activities. All steps of the algorithms are fitness-preserving [16]; i.e., all traces in the input event log are guaranteed to be included in the language of the generated model.

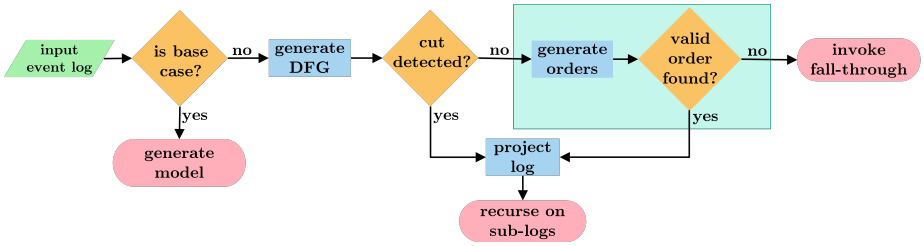


Fig. 5: Approach for the discovery of POWL models.

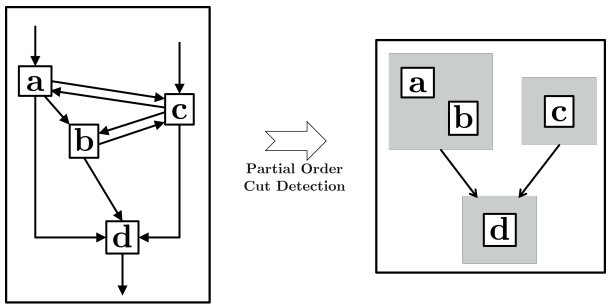


Fig. 6: Partial order cut detection.

6.3 Partial Order Cut

We adapt the inductive miner to discover POWL models instead of process trees. If the algorithm fails to detect a base case or a process tree cut, we mine for partial orders before invoking the fall-through function; we generate partial orders over all possible partitionings of activities, and we validate these orders using certain rules. If a *valid* order is found, then the event log is projected on the partitioning of activities and the recursion continues on the sub-logs; otherwise, the fall-through function is invoked. An overview of the different steps of our approach for the discovery of POWL models is shown in Figure 5.

We define a *partial order cut* as a partitioning of the activities and a partial order over the partitioning. Since a partial order is transitive, we use the eventually-follows graph instead of the directly-follows graph for the detection of partial order cuts (we discuss the detection step in Section 6.4). Figure 6 shows an example eventually-follows graph with a partial order cut detected based on it. This cut consist of a partitioning of activities $P=\{\{a,b\},\{c\},\{d\}\}$ and a partial order $\prec$ defined by two ordering relations $\{a,b\}\prec\{d\}$ and $\{c\}\prec\{d\}$.

Definition 4 (Partial Order Cut). *Let $L\in\mathcal{M}(\Sigma^*)$ be an event log. A partial order cut of L is a poset $\rho=(\{A_1,...,A_n\},\prec)$ over a partitioning of the activities into $n\geq 2$ subsets; i.e., $\Sigma_L=A_1\cup...\cup A_n$ and $A_i\cap A_j=\emptyset$ for $1\leq i<j\leq n$.*

Our approach tries to detect a process tree cut before mining for a partial order. In case a sequence or a concurrency cut is detected, we transform it into

a partial order cut since POWL models do not support the operators $\rightarrow$ and $+$. A concurrency is modeled as a poset with an empty ordering relation; i.e., we transform $(+, A_1, ..., A_n)$ into the poset $(\{A_1, ..., A_n\}, \emptyset)$. A sequence is modeled as poset using the sequential order of the nodes; i.e., we transform $(\rightarrow, A_1, ..., A_n)$ into the poset $(\{A_1, ..., A_n\}, \prec)$ where $A_i \prec A_j$ iff $1 \leq i < j \leq n$.

The base inductive miner only detects cuts that preserve perfect fitness [16]; i.e., all traces observed in the event log are guaranteed to be included in the language of the generated model. Similarly, we ensure perfect fitness for our approach.

Definition 5 (Fitness-Preserving Partial Order Cut). *Let $L \in \mathcal{M}(\Sigma^*)$ be an event log and $\rho = (P, \prec)$ be a partial order cut of L. ρ is fitness-preserving iff for any $A_1, A_2 \in P$: $A_1 \prec A_2 \Rightarrow \{\sigma \uparrow_{A_1 \cup A_2} \mid \sigma \in L\} \subseteq {A_1}^* \cdot {A_2}^*$.*

6.4 Detection of Partial Order Cut

Our approach mines for a partial order cut before invoking the fall-through function. We use a brute-force approach that generates all possible partitionings of activities, and for each partitioning, we mine for a *valid* partial order. We define a valid partial order cut as a behavioral pattern in the eventually-follows graphs that corresponds to a partial order over the partitioning of activities.

A valid order contains an ordering edge between two groups of activities if and only if all activities of the first group are eventually followed by all activities of the second group and none of the activities of the second group is eventually followed by an activity of the first group. Moreover, two groups are not connected through any ordering edges if and only if they are concurrent. We define two groups to be concurrent if every activity of each group is eventually following all activities of the other group. Finally, we ensure that groups with no preceding groups with respect to the order contain start activities and groups with no succeeding groups with respect to the order contain end activities.

Definition 6 (Valid Partial Order Cut). *Let $L \in \mathcal{M}(\Sigma^*)$ be an event log and $\rho = (P, \prec)$ be a partial order cut of L. ρ is valid if the following conditions hold for all $A_i, A_j \in P$; $A_i \neq A_j$:*

1. $(A_i \prec A_j \wedge A_j \not\prec A_i)$ *iff* $\forall_{a_i \in A_i, a_j \in A_j} a_i \rightsquigarrow_L a_j \wedge a_j \not\rightsquigarrow_L a_i$.
2. $(A_i \not\prec A_j \wedge A_j \not\prec A_i)$ *iff* $\forall_{a_i \in A_i, a_j \in A_j} (a_i \rightsquigarrow_L a_j \wedge a_j \rightsquigarrow_L a_i)$.
3. *if* $\nexists_{A_k \in P} A_k \prec A_i$, *then* $A_i \cap L_{\triangleright} \neq \emptyset$.
4. *if* $\nexists_{A_k \in P} A_i \prec A_k$, *then* $A_i \cap L_{\square} \neq \emptyset$.

The partial order cut shown in Figure 6 is valid. Note that if a valid partial order cut over a partitioning of activities exists, then it is unique (the first condition of Definition 6 uniquely defines a relation). Moreover, a valid partial order cut is fitness-preserving; i.e., the partial order cut detection step is fitness-preserving.

Theorem 1. *Let $L \in \mathcal{M}(\Sigma^*)$ be an event log and $\rho = (P, \prec)$ be a valid partial order cut of L. ρ is fitness-preserving.*

Proof. Let $A_i, A_j \in P$ with $A_i \prec A_j$. Then, $A_j \not\prec A_i$ since $\prec$ is asymmetric.
$\Rightarrow \quad \forall_{a_i \in A_i, a_j \in A_j} \; a_i \rightsquigarrow_L a_j \wedge a_j \not\rightsquigarrow_L a_i.$
$\Rightarrow \quad \nexists_{\sigma \in L, 1 \leq k < l \leq |\sigma|} \; \sigma(k) \in A_j \wedge \sigma(l) \in A_i.$
$\Rightarrow \quad \{\sigma \uparrow_{A_i \cup A_j} \mid \sigma \in L\} \subseteq \{\sigma_i \cdot \sigma_j \mid \sigma_i \in {A_i}^* \wedge \sigma_j \in {A_j}^*\} = {A_i}^* \cdot {A_j}^*.$ □

6.5 Discussion: Scalability, Fitness, Maximality

Our approach serves as a proof of concept to demonstrate the feasibility of using POWL models for process discovery. The step of partial order cut detection needs to be improved in terms of efficiency. We use a brute force approach for the step of partial order cut detection. Our approach generates all possible partitionings of activities until a valid order over one of these partitionings is found. For a large number of activities, this step becomes very time-consuming unless a partial order cut is detected in an early stage. A possible improvement for future work is to exploit the eventually-follows graph to dynamically prune the search space instead of generating all partitionings of activities.

The inductive mining framework guarantees perfect fitness for the generated models if all steps of the discovery are fitness-preserving [16, Corollary 4.2]. Our approach extends the base inductive miner (IM) by adding the step of partial order cut detection. All steps of IM are fitness-preserving [16], and we mine for valid partial order cuts, which are also fitness-preserving (Theorem 1). Therefore, our approach is guaranteed to discover fitting models.

Precision is another criterion used to assess the quality of process discovery approaches. A precise process model is a model that does not allow for behavior not observed in the log. Process discovery approaches aim at creating a balance between fitness and precision. As our approach guarantees perfect fitness, our goal is to maximize precision by discovering a model that allows for less behavior as possible. In Definition 6, we defined valid partial order cuts by exploiting the eventually-follows graph. This definition ensures the uniqueness of a valid cut for a given partitioning of activities. However, a general notion of maximality among different partitionings is missing. Currently, we only maximize the size of the partitioning; we generate the partitioning of maximal size first and try to detect a valid partial order cut over it, then we decrease the size of the partitioning gradually. For future work, we would like to have a stronger notion of maximality for valid partial order cuts over different partitionings.

7 Evaluation

We implemented our approach for the discovery of POWL models in PM4Py (`http://pm4py.org/`), and we evaluate it using real-life event logs. We compare our approach (IM_P) with the base inductive miner (IM) and a more advanced

Table 1: Evaluation results. We highlighted differences in precision for models discovered by IM_P compared to IM: increases in green and decreases in red.

Event log	#Act.	Time (sec)				Precision				Fitness				simplicity			
		IM	IM_P	IM_C	SM	IM	IM_P	IM_C	SM	IM	IM_P	IM_C	SM	IM	IM_P	IM_C	SM
BPI 2017	8	1.54	1.22	6.03	8.85	0.37	0.68	0.27	0.56	1	1	1	1	0.67	0.74	0.62	0.65
BPI 2017	12	18.02	5.66	13.79	16.86	0.23	0.34	0.22	0.34	1	1	1	1	0.65	0.68	0.64	0.57
BPI 2018	8	25.79	16.31	11.48	45.14	0.35	0.32	0.29	0.29	1	1	0.99	1	0.65	0.63	0.64	0.54
BPI 2018	12	59.75	112.29	17.25	55.36	0.2	0.21	0.27	0.19	1	1	0.98	1	0.61	0.63	0.63	0.48
BPI 2019	8	2.93	2.95	8.5	13.48	0.62	0.78	0.78	0.73	1	1	1	1	0.64	0.67	0.67	0.54
BPI 2019	12	5.56	3.3	13.15	13.46	0.55	0.7	0.7	0.7	1	1	1	1	0.64	0.64	0.65	0.49
Dom. Decl.	8	0.08	0.26	0.5	0.57	0.4	0.4	0.39	0.9	1	1	1	1	0.65	0.66	0.65	0.57
Dom. Decl.	12	0.13	38.16	0.98	0.61	0.5	0.54	0.37	0.84	1	1	1	1	0.61	0.67	0.62	0.59
Int. Decl.	8	0.05	0.08	0.4	0.54	0.5	0.53	0.59	0.66	1	1	1	1	0.67	0.71	0.69	0.54
Int. Decl.	12	0.12	0.31	1.6	0.68	0.47	0.51	0.53	0.56	1	1	0.98	0.89	0.65	0.69	0.69	0.38
Travel Permit	8	0.17	0.2	1.05	1.58	0.51	0.51	0.68	0.56	1	1	0.96	0.92	0.67	0.67	0.71	0.44
Travel Permit	12	0.45	280.02	0.79	0.91	0.33	0.35	0.6	0.41	1	1	0.95	0.99	0.65	0.67	0.68	0.49
Travel Costs	8	0.04	0.19	0.12	0.19	0.43	0.39	0.35	0.55	1	1	1	0.99	0.63	0.68	0.67	0.56
Travel Costs	12	0.15	276.87	0.22	0.28	0.23	0.35	0.24	0.54	1	1	1	0.83	0.58	0.68	0.63	0.38
Pay. Request	8	0.05	0.19	0.27	0.43	0.75	0.75	0.29	0.91	1	1	0.9	1	0.63	0.68	0.66	0.55
Pay. Request	12	0.09	41.44	0.5	0.5	0.49	0.49	0.38	0.82	1	1	1	1	0.6	0.67	0.64	0.55
Sepsis	8	0.08	0.1	0.07	0.21	0.51	0.51	0.51	0.42	1	1	1	1	0.64	0.65	0.64	0.5
Sepsis	12	0.2	0.14	0.16	0.3	0.34	0.35	0.4	0.31	1	1	1	1	0.64	0.65	0.63	0.47
Fine	8	0.49	0.71	9.21	5.35	0.76	0.76	0.76	0.91	1	1	1	1	0.66	0.67	0.68	0.56
Fine	11	0.69	7.13	17.52	5.49	0.58	0.58	0.78	0.92	1	1	1	1	0.62	0.63	0.64	0.51
Hosp. Billing	8	0.59	0.69	9.09	10.67	0.78	0.78	0.6	0.9	1	1	1	1	0.67	0.67	0.64	0.56
Hosp. Billing	12	0.99	1.55	14.17	5.61	0.6	0.6	0.46	0.86	1	1	1	1	0.65	0.66	0.62	0.53

variant of the inductive miner that handles incompleteness (IM_C) [16]. We additionally apply another state-of-the-art discovery approach: the split miner (SM) [6]. Since both IM and IM_P guarantee perfect fitness, we set the filtering threshold of the split miner to 0; for the other parameters, we use the default values.

We transform the discovered models into WF-nets, and we assess their quality using three conformance-checking metrics implemented in PM4Py: fitness [7], precision [24], and simplicity [8]. Fitness quantifies how well the discovered model reproduces the behavior recorded in the event log. Precision quantifies the degree to which the model is restricted to the behavior recorded in the event log. The simplicity metric implemented in PM4Py evaluates a model as simple if it has a low average degree of arcs (i.e., a low number of arcs per place or transition).

We use multiple real-life event logs for the evaluation. We use an event log that records sepsis cases from a hospital [19], an event log of a system managing road traffic fines [18], an event log for a hospital billing system [20], BPI Challenge 2017 [9], BPI Challenge 2018 [12], BPI Challenge 2019 [10], and the five event logs of the BPI Challenge 2020 [11]: Request For Payment, Prepaid Travel Costs, Travel Permit Data, International Declarations, and Domestic Declarations. We filter the event logs to only keep the most frequent activities using two values for this filter: 8 and (at most) 12 activities.

The results of the evaluation are shown in Table 1. We report the time required for discovering each model and the obtained conformance-checking scores.

On the one hand, IM_P led to better time performance than the other approaches in some cases (e.g., BPI Challenge 2017). On the other hand, IM_P was more time-consuming in other cases. Compared to IM, the time increased from 0.17 seconds to 0.2 seconds for the travel permit log with 8 activities and from 0.45 seconds to 280.02 seconds for the log with 12 activities. This shows how increasing the number of activities can dramatically worsen the time performance.

These results were expected as discussed in Section 6.5. IM_P serves as proof of concept, and it needs to be improved in terms of scalability in future work.

As expected, both IM and IM_P led to perfectly fitting models, while for IM_C and SM, we observe lower fitness values in some cases. As discussed in Section 6.5, IM_P preserves the fitness guarantee of IM as the step of partial order cut detection is fitness-preserving.

In general, the three variants of the inductive miner led to simpler models than the split miner. Our approach achieved the highest simplicity score on average (0.67), while SM achieved the lowest score on average (0.52). Note that these scores only evaluate the simplicity of the discovered models after transforming them into WF-nets; i.e., we are not evaluating the simplicity of the three different types of models (IM and IM_C produce process trees [16], IM_P produces POWL models, and SM produces BPMN models [6]).

We observe that precision varies among the different event logs. On the one hand, SM led to significantly higher precision values and lower simplicity values compared to the inductive miner in many cases (e.g., the fine management logs). On the other hand, we observe cases where the inductive miner performed better in terms of both precision and simplicity (e.g., the sepsis cases log). By comparing the three variants of the inductive miner with each other, we observe that IM_P led to the highest precision on average. In Table 1, we highlighted differences in precision between the models discovered by IM and IM_P as IM_P extends IM aiming at improving precision.

In general, our approach led to more precise models than IM. For some of these cases, this is due to the handling of incompleteness. Our approach uses the eventually-follows graph instead of the directly-follows graph for detecting partial order cuts. This design decision helps to handle incompleteness in the event log since the directly-follows graph is a subset of the eventually-follows graph; i.e., if the event log is incomplete and some connections are missing in the directly-follows graph, these connections might still be present in the eventually-follows graph. For instance, the precision of the BPI Challenge 2019 log with 12 activities increased from 0.55 to 0.7. However, the POWL model discovered by IM_P does not contain any structures that cannot be captured by a process tree. It models the same behavior of the process tree discovered by IM_C.

Figure 7 shows the POWL model discovered by IM_P for the BPI Challenge 2017 event log with 8 activities. The POWL model achieved a precision of 0.68 compared to 0.37 achieved by the model discovered by the base inductive miner. IM_P discovers local dependencies between activities IM fails to discover. For example, IM_P discovered a simple sequential relation between the activities "A_Concept" and "A_Accepted". This simple sequence cannot be discovered by IM as it only represents a local dependency; i.e., it does not correspond to a global process tree cut covering all activities. The POWL model shows a non-hierarchical structure that cannot be captured using a process tree (i.e., the partial order over {A_Concept}, {A_Accepted}, {O_Create Offer, O_Created}, {W_Complete application}, and {W_Call after offers, A_Complete}).

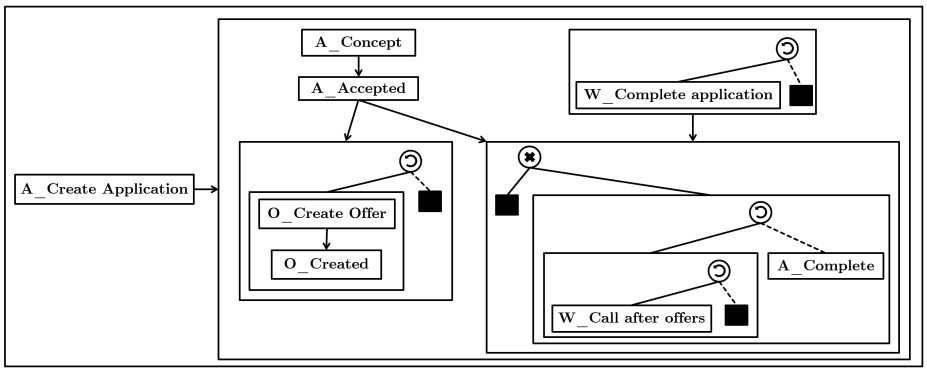


Fig. 7: POWL model discovered for the BPI Challenge 2017 event log.

Although IM_P led to more precise models compared to IM in most cases, we observe some exceptions. For instance, for the BPI Challenge 2018 event log with 8 activities, the precision decreased from 0.35 to 0.32. This is an example where invoking the fall-through function led to better results than detecting a partial order. In order to continue the recursion, the fall-through function returned a concurrency cut between an activity that occurs in every trace at most once and the rest of the activities.

To sum up, our evaluation shows that our approach discovers structures that cannot be captured by a process tree, and it leads to high precision and simplicity in general. However, the approach needs to be improved in terms of scalability.

8 Conclusion

Different modeling notations are used to model processes. Partial orders provide a compact representation of concurrent systems, but they are not able to represent cyclic or choice behavior. Process trees use control-flow operators for modeling processes as mathematical trees, but they are limited to hierarchical structures. A POWL model is a partially ordered graph representation, extended with control-flow operators for modeling choice and loop. POWL models can be converted into sound Workflow nets. We proposed an approach to demonstrate the feasibility of using POWL models in process discovery. We evaluated our approach using real-life event logs, and the evaluation showed that our approach is able to discover dependencies that cannot be captured by a process tree.

We propose multiple ideas for future work. First, our approach needs to be improved in terms of scalability. Moreover, it is possible to develop other types of approaches for the discovery of POWL models. Our approach is based on the base inductive miner; i.e., it is a top-down recursive approach. Developing a bottom-up approach and comparing it with the top-down approach is an interesting idea for future work. Moreover, we can develop a discovery approach for POWL models that exploits life-cycle information in event logs where each event has

a duration (i.e., each event has a start timestamp and an end timestamp). We usually assume event logs to be totally ordered; i.e., we define a trace as a sequence of activities. However, event logs might also be partially ordered. We suggest developing an approach for the discovery of POWL models from partially ordered event logs. Finally, the idea of combining different modeling notations to create new types of process models is not restricted to POWL models. This idea can be applied to combine other types of process models.

References

1. van der Aalst, W.M.P.: Workflow verification: Finding control-flow errors using petri-net-based techniques. In: Business Process Management, Models, Techniques, and Empirical Studies. LNCS, vol. 1806, pp. 161–183. Springer (2000)
2. van der Aalst, W.M.P.: Business process simulation revisited. In: Enterprise and Organizational Modeling and Simulation - 6th International Workshop, held at CAiSE 2010. Selected Papers. LNBIP, vol. 63, pp. 1–14. Springer (2010)
3. van der Aalst, W.: Process Mining - Data Science in Action, Second Edition. Springer (2016)
4. van der Aalst, W.M.P., De Masellis, R., Di Francescomarino, C., Ghidini, C., Kourani, H.: Discovering hybrid process models with bounds on time and complexity: When to be formal and when not? Information Systems **116**, 102214 (2023)
5. van der Aalst, W.M.P., Dumas, M., ter Hofstede, A.H.M., Russell, N., Verbeek, H.M.W., Wohed, P.: Life after BPEL? In: Formal Techniques for Computer Systems and Business Processes, EPEW 2005 and WS-FM 2005, Proceedings. LNCS, vol. 3670, pp. 35–50. Springer (2005)
6. Augusto, A., Conforti, R., Dumas, M., Rosa, M.L., Polyvyanyy, A.: Split miner: automated discovery of accurate and simple business process models from event logs. Knowl. Inf. Syst. **59**(2), 251–284 (2019)
7. Berti, A., van der Aalst, W.M.P.: Reviving token-based replay: Increasing speed while improving diagnostics. In: van der Aalst, W.M.P., Bergenthum, R., Carmona, J. (eds.) Proceedings of the International Workshop on Algorithms & Theories for the Analysis of Event Data, Satellite event of Petri Nets 2019 and ACSD 2019. CEUR Workshop Proceedings, vol. 2371, pp. 87–103. CEUR-WS.org (2019)
8. Blum, F.R.: Metrics in process discovery. Tech. Rep. TR/DCC-2015-6, Computer Science Department, Universidad de Chile, Chile (2015)
9. van Dongen, B.: BPI Challenge 2017 (2017)
10. van Dongen, B.: BPI Challenge 2019 (2019)
11. van Dongen, B.: BPI Challenge 2020 (2020)
12. van Dongen, B., Borchert, F.: BPI Challenge 2018 (2018)
13. Dumas, M., García-Bañuelos, L.: Process mining reloaded: Event structures as a unified representation of process models and event logs. In: Application and Theory of Petri Nets and Concurrency - 36th International Conference, Proceedings. LNCS, vol. 9115, pp. 33–48. Springer (2015)
14. Golani, M., Pinter, S.S.: Generating a process model from a process audit log. In: Business Process Management, International Conference, Proceedings. LNCS, vol. 2678, pp. 136–151. Springer (2003)
15. Leemans, M., van der Aalst, W.M.P.: Discovery of frequent episodes in event logs. In: Data-Driven Process Discovery and Analysis - 4th International Symposium, Revised Selected Papers. LNBIP, vol. 237, pp. 1–31. Springer (2014)

16. Leemans, S.J.J.: Robust Process Mining with Guarantees - Process Discovery, Conformance Checking and Enhancement, LNBIP, vol. 440. Springer (2022)
17. Leemans, S.J., van Zelst, S.J., Lu, X.: Partial-order-based process mining: a survey and outlook. Knowl Inf Syst (2022)
18. de Leoni, M.M., Mannhardt, F.: Road Traffic Fine Management Process (2015)
19. Mannhardt, F.: Sepsis Cases - Event Log (2016)
20. Mannhardt, F.: Hospital Billing - Event Log (2017)
21. Mannila, H., Toivonen, H., Verkamo, A.I.: Discovery of frequent episodes in event sequences. Data Min. Knowl. Discov. **1**(3), 259–289 (1997)
22. Mokhov, A., Carmona, J.: Event log visualisation with conditional partial order graphs: from control flow to data. In: Proceedings of the International Workshop on Algorithms & Theories for the Analysis of Event Data, Satellite event of Petri Nets 2015 and ACSD 2015. CEUR Workshop Proceedings, vol. 1371, pp. 16–30. CEUR-WS.org (2015)
23. Mokhov, A., Yakovlev, A.: Conditional partial order graphs: Model, synthesis, and application. IEEE Trans. Computers **59**(11), 1480–1493 (2010)
24. Munoz-Gama, J., Carmona, J.: A fresh look at precision in process conformance. In: Hull, R., Mendling, J., Tai, S. (eds.) BPM 2010, Hoboken, NJ, USA. Proceedings. LNCS, vol. 6336, pp. 211–226. Springer (2010)
25. Nielsen, M., Plotkin, G.D., Winskel, G.: Petri nets, event structures and domains, part I. Theor. Comput. Sci. **13**, 85–108 (1981)
26. Ouyang, C., Verbeek, E., van der Aalst, W.M.P., Breutel, S., Dumas, M., ter Hofstede, A.H.M.: Formal semantics and analysis of control flow in WS-BPEL. Sci. Comput. Program. **67**(2-3), 162–198 (2007)
27. Slaats, T., Schunselaar, D.M.M., Maggi, F.M., Reijers, H.A.: The semantics of hybrid process models. In: On the Move to Meaningful Internet Systems: OTM 2016 Conferences - Confederated International Conferences: CoopIS, C&TC, and ODBASE 2016, Proceedings. LNCS, vol. 10033, pp. 531–551 (2016)