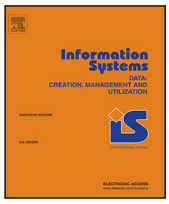
Information Systems 128 (2025) 102493

Contents lists available at ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/is



Discovering partially ordered workflow models

Humam Kourani [a,b,*], Sebastiaan J. van Zelst [c], Daniel Schuster [a,b], Wil M.P. van der Aalst [a,b]

[a] Fraunhofer Institute for Applied Information Technology FIT, Data Science and Artificial Intelligence, Schloss Birlinghoven, Sankt Augustin, 53757, Germany
[b] RWTH Aachen University, Process and Data Science, Ahornstraße 55, Aachen, 52074, Germany
[c] Celonis Labs GmbH, Theresienstraße 6, Munich, 80333, Germany

ARTICLE INFO

Keywords:
Process modeling
Partial order
Process discovery
Partially ordered workflow language

ABSTRACT

In many real-world scenarios, processes naturally define partial orders over their constituent tasks. Partially ordered representations can be exploited in process discovery as they facilitate modeling such processes. The Partially Ordered Workflow Language (POWL) extends partially ordered representations with control-flow operators to support modeling common process constructs such as choice and loop structures. POWL integrates the hierarchical nature of process trees with the flexibility of partially ordered representations, opening up significant opportunities in process discovery. This paper presents and compares various approaches for the automated discovery of POWL models. We investigate the effects of applying varying validity criteria to partial orders, and we propose methods for incorporating frequency information to improve the quality of the discovered models. Additionally, we propose alternative visualizations for POWL models, offering different approaches that may be useful in various contexts. The discovery approaches are evaluated using various real-life data sets, demonstrating the ability of POWL models to capture complex process structures.

1. Introduction

Process models serve as visual representations of processes, facilitating communication and enabling simulation and analysis of these processes. Organizations employ information systems to track and record data about the execution of their processes, and these data can be used for the discovery of process models. Discovered models can help gain insights into the underlying processes, empowering organizations to streamline and automate their processes, ultimately leading to more informed and effective decision-making. This can significantly contribute to the overall performance and efficiency of an organization.

Workflow nets (WF-nets) [1] are a widely used process modeling language for the formal representation of business processes. While traditional process discovery techniques often produce WF-nets, many of these techniques utilize other modeling notations as intermediate process representations. For instance, hierarchical modeling languages, such as *process trees* [2], are commonly employed in process discovery due to their *soundness* guarantee. Soundness refers to meeting certain quality requirements, e.g., a sound model has no dead parts that can never be reached.

Process trees fall short in accurately capturing non-hierarchical structures that are common in real-world scenarios. This limitation is particularly evident in processes involving concurrent tasks with complex interdependencies, where traditional models used in process discovery either oversimplify reality or become overly complex.

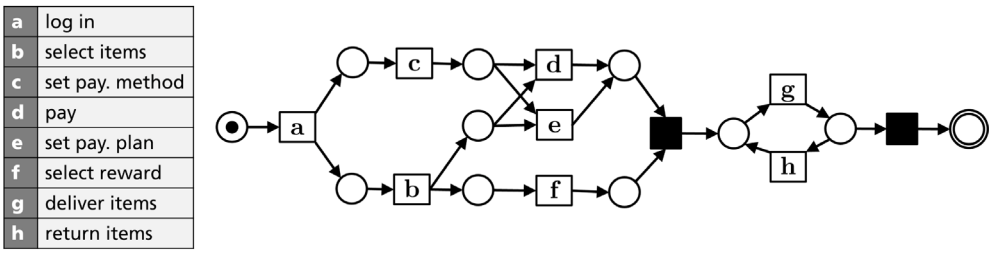
To illustrate this point, consider the process of online shopping illustrated as a *WF-net* in Fig. 1(a). This process involves non-hierarchical dependencies between selecting the items (b), setting the payment method (c), the payment activities (d and e), and selecting a complimentary reward depending on the total purchase amount (f). Process trees would either duplicate activities to capture this behavior (e.g., Fig. 1(b)) or oversimplify the process, failing to represent the actual workflow accurately (e.g., the tree in Fig. 1(c) allows for selecting the reward and/or completing the payment before selecting the items).

Partial orders allow for a compact representation of concurrent systems and support adding execution order restrictions between activities. In a partial order, certain activities are mandated to follow a specific order, while other activities can be executed in parallel. The non-hierarchical dependencies in our example can be modeled as a partial order as shown in Fig. 1(d).[1] However, partial orders lack support for cyclic and choice structures. The partial order visualized in Fig. 1(d) does not show the choice between payment and setting a payment plan, and it only captures cases with exactly two deliveries (instead of a loop of returning items for exchange followed by a new delivery each time).

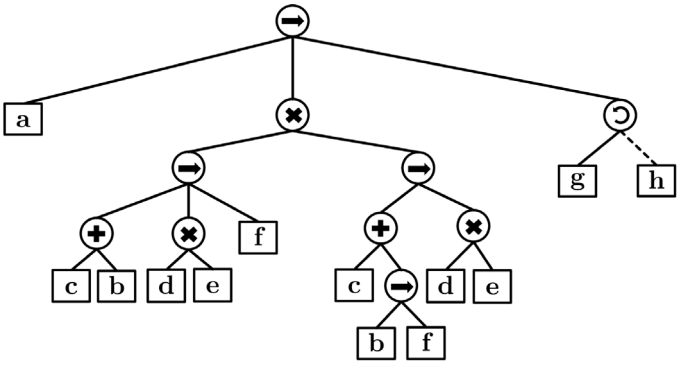
* Corresponding author at: Fraunhofer Institute for Applied Information Technology FIT, Data Science and Artificial Intelligence, Schloss Birlinghoven, Sankt Augustin, 53757, Germany.
E-mail address: humam.kourani@fit.fraunhofer.de (H. Kourani).

[1] For the sake of simplicity, we only visualize the transitive reduction of partial orders.

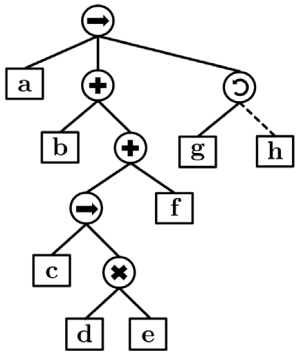
https://doi.org/10.1016/j.is.2024.102493
Received 19 January 2024; Received in revised form 18 September 2024; Accepted 18 November 2024
Available online 25 November 2024



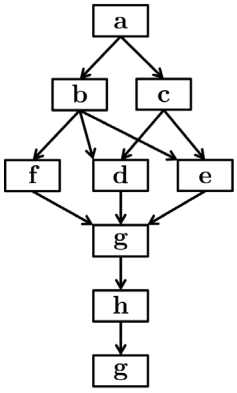
(a) A labeled Workflow net precisely modeling the behavior.



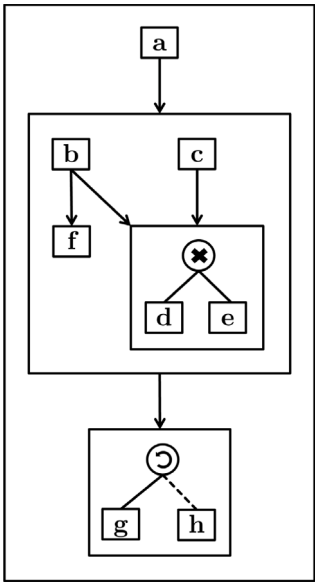
(b) A process tree with duplicated activity labels.



(c) An imprecise process tree with unique activity labels.



(d) A graph representation of a partial order that only captures cases with two deliveries and does not show the choice between d and e.



(e) A POWL model precisely modeling the behavior.

Fig. 1. Process models modeling a process for purchasing items in an online store.

Our proposed modeling notation, named Partially Ordered Workflow Language (POWL) [3], overcomes these challenges by combining the hierarchical structure of process trees with the flexibility of partial orders. A POWL model is a partially ordered graph extended with control-flow operators for modeling choice and loop structures. A POWL model can also be viewed as a process tree extended to support combining sub-models as partial orders. Fig. 1(e) shows a POWL model representing the behavior of our example process. The uppermost layer of the model is a partial order that sets the sequential order of the activity sets $\{a\}$, $\{b, c, d, e, f\}$, and $\{g, h\}$. Additionally, another partial order models the non-hierarchical dependencies between the activity sets $\{b\}$, $\{c\}$, $\{d, e\}$, and $\{f\}$. The process tree operators $\times$ and $\circlearrowleft$ are utilized to represent the exclusive choice between d and e, and the loop between the activities g and h, respectively.

This paper builds upon our previous work in [3,4] on exploiting POWL models in process discovery. First, we investigate the effect of using varying degrees of validity for partial orders; we introduce a discovery approach that accepts a wider range of partial orders compared to our previous methods. Second, we enhance the adaptability of our approach to real-world data complexities by introducing two methods for integrating frequency-based filtering in the discovery of POWL models.

POWL is able to represent non-hierarchical structures while preserving the soundness guarantee of process trees. This makes POWL an ideal modeling language for process discovery. The contribution of this paper also includes proving the soundness guarantee of POWL models. Beyond process discovery, we propose alternative visualizations for POWL models, aiming to provide flexible options that can aid interpretability in different scenarios.

Our contribution in this paper is guided by the following questions we investigate:

1. What is the effect of relaxing the strong validity requirements in the discovery of POWL models?
2. How can frequency-based filtering be incorporated into the discovery of POWL models, and what impact does it have on the quality of the resulting models?
3. What alternative visualizations of POWL models can improve their interpretability in different scenarios?
4. Can we prove the soundness guarantee of POWL models?

The remainder of the paper is structured as follows. We discuss related work in Section 2, and we present an overview of preliminaries in Section 3. The concept of POWL models is defined in Section 4, and an approach for transforming POWL models into sound workflow nets is presented in Section 5. In Section 6, we address the automated discovery of POWL models, and we propose methods for incorporating frequency-based filtering in the discovery in Section 7. We propose ideas for improving the visualization of POWL models in Section 8. Our discovery approaches are evaluated on real-life data in Section 9, and the paper concludes with a summary and ideas for future research in Section 10.

2. Related work

In the field of process mining, partial orders play a crucial role in both data representation and process modeling. A detailed examination of their use in process mining is found in [5]. Process execution variants for partially ordered event data are defined in [6]. The authors in [7] present three methods for aggregating partially ordered sets of events to generate Petri nets. Another method for deriving Petri nets from languages characterized by partial orders is introduced in [8]. The concept of frequent episodes, as partially ordered sets of events, was introduced in [9], and this idea was further adapted to event logs in [10]. Workflow graphs incorporating partially ordered activities are discussed in [11]. Moreover, [12] proposes methods for the discovery of prime event structures, where a prime event structure is defined as a partially ordered graph enriched with a conflict relation [13]. In [14], the authors present a method for generating conditional partial order graphs from event logs. A conditional partial order graph [15] represents a family of partial orders, allowing for modeling choice structures. Both prime event structure and conditional partial order graphs are limited in modeling cyclic behavior.

The creation of hybrid process models, combining different modeling languages, is an area of active exploration. Hybrid Petri nets, as described in [16], extend traditional Petri nets by introducing informal arcs to connect transitions. Another type of hybrid process models, blending imperative and declarative modeling languages, is discussed in [17].

In our previous work [3], we introduced POWL models, which combine elements of process trees with partially ordered graphs, presenting a novel perspective in process modeling. In [18], BPEL (Business Process Execution Language) is introduced as a flow language designed for modeling web services. Similar to POWL, BPEL allows for combining primitives through advanced control-flow constructs and additionally imposing an execution order of concurrently running primitives using control links. While BPEL serves as a potent tool for web service implementation, its complexity often renders it more akin to a programming language, posing challenges for end users.

Various methods for the discovery of process models from event data have been introduced and refined. For an overview of process discovery techniques prevalent in process mining, we refer to [19,20]. In our previous work [3], we introduced an initial approach that demonstrates the feasibility of using POWL models in process discovery. Subsequently, we refined this approach in [4] to ensure scalability on large data sets by mining for partial orders that strictly conform with the data. Both proposed approaches extend the inductive mining framework detailed in [2]. Our work on POWL models in [3,4] situates itself within the broader context of advancing process discovery by marking a step towards more compact and comprehensive process models.

3. Preliminaries

This section introduces fundamental preliminaries to facilitate the understanding of the paper.

3.1. Basic notations

We define the set of natural number $\mathbb{N} = \{1, 2, 3, \dots\}$, the set of odd numbers $\mathbb{N}_{odd} = \{1, 3, 5, \dots\}$, and the set of even numbers $\mathbb{N}_{even} = \{2, 4, 6, \dots\}$. For any set X, its *powerset* is $\mathcal{P}(X) = \{X' \subseteq X\}$. For n sets $X_1, \dots, X_n$, their *n-ary Cartesian product* is $X_1 \times \dots \times X_n = \{(x_1, \dots, x_n) \mid x_i \in X_i \text{ for } 1 \leq i \leq n\}$. An *n-ary relation* is a subset of the n-ary Cartesian product.

Let $\prec \subseteq X \times X$ be a binary relation (i.e., 2-ary relation) over a set X. We use $x_1 \prec x_2$ to indicate that $(x_1, x_2) \in \prec$ and $x_1 \not\prec x_2$ to indicate that $(x_1, x_2) \notin \prec$. We define the *transitive closure* of $\prec$ as $\prec^+ = \{(a, b) \in X \times X \mid \exists n \in \mathbb{N};\ x_0, x_1, \dots, x_n \in X\ (a = x_0 \wedge b = x_n \wedge \forall 1 \leq i \leq n\ (x_{i-1} \prec x_i))\}$.

For a set X, a *partitioning* of X into $n \in \mathbb{N}$ subsets is a set of subsets $P = \{X_1, \dots, X_n\}$ such that $X = X_1 \cup \dots \cup X_n$ and $X_i \cap X_j = \emptyset$ for $1 \leq i < j \leq n$. We use $Part_n(X)$ to denote the set of all partitionings of X into n subsets, and we define $Part(X) = \bigcup_{1 \leq n \leq |X|} Part_n(X)$.

A function $f : X \to Y$ is called *injective* if $\forall x, x' \in X\ (f(x) = f(x') \Rightarrow x = x')$, and f is called *surjective* if $\forall y \in Y \exists x \in X\ (f(x) = y)$. The function f is *bijective* if it is both injective and subjective.

A *multi-set* generalizes the notion of a set by tracking the frequencies of its elements. A multi-set over a set X is expressed as $M = [x_1{}^{c_1}, \dots, x_n{}^{c_n}]$ where $x_1, \dots, x_n \in X$ are the elements of M (denoted as $x_i \in M$ for $1 \leq i \leq n$) and $M(x_i) = c_i$ is the frequency of x_i for $1 \leq i \leq n$. We use $\mathcal{M}(X)$ to denote the set of all multi-sets over X.

A sequence over a set X is defined as function $\sigma : \{1, \dots, n\} \to X$, and we express it as $\sigma = \langle \sigma(1), \dots, \sigma(n) \rangle$. The length of σ is denoted by $|\sigma| = n$, and the set of all sequences over X is denoted by X^*. The concatenation of two sequences σ_1 and σ_2 is expressed as $\sigma_1 \cdot \sigma_2$, e.g., $\langle x_1 \rangle \cdot \langle x_2, x_1 \rangle = \langle x_1, x_2, x_1 \rangle$. For two sets of sequences L_1 and L_2, we write $L_1 \cdot L_2 = \{\sigma_1 \cdot \sigma_2 \mid \sigma_1 \in L_1 \wedge \sigma_2 \in L_2\}$. The projection of a sequence σ on a set Y is denoted as $\sigma{\restriction_Y}$, e.g., $\langle x_1, x_2, x_1 \rangle{\restriction_{\{x_1, x_3\}}} = \langle x_1, x_1 \rangle$.

A *strict partial order* over a set X is binary relation that is *irreflexive* ($x \not\prec x$ for all $x \in X$) and *transitive* ($x_1 \prec x_2 \wedge x_2 \prec x_3 \Rightarrow x_1 \prec x_3$). Irreflexivity and transitivity imply *asymmetry* as well ($x_1 \prec x_2 \Rightarrow x_2 \not\prec x_1$). The *transitive reduction* of a strict partial order $\prec$ is uniquely defined as $\prec^- = \{(x_1, x_3) \in \prec \mid \nexists x_2 \in X\ (x_1 \prec x_2 \wedge x_2 \prec x_3)\}$.

In this paper, we use the term *partial order* to refer to a strict partial order. For a partial order $\prec$ over a set X, we call $\rho = (X, \prec)$ a *partially ordered set (poset)*. We denote the set of all posets over a set X by $\Pi(X)$. The language of a poset $\rho = (X, \prec)$ is the set of sequences $\mathcal{L}(\rho) = \{\sigma \in X^* \mid \exists f : \{1, \dots, |\sigma|\} \to X\ (\text{f is bijective} \wedge \forall 1 \leq i \leq |\sigma|\ (\sigma(i) = f(i) \wedge \forall 1 \leq j \leq |\sigma|\ (f(i) \prec f(j) \Rightarrow i < j)))\}$.

For a poset $\rho = (X, \prec)$ and a *labeling* function $\gamma : X \to Y$, the triple $\rho' = (X, \prec, \gamma)$ is called a *labeled partial order* over X and Y. The language of ρ' is the set of sequences $\mathcal{L}(\rho') = \{\langle \gamma(\sigma(1)), \dots, \gamma(\sigma(|\sigma|)) \rangle \mid \sigma \in \mathcal{L}(\rho)\}$. We use $\Pi(X, Y)$ to denote the set of all labeled partial orders over X and Y.

Let us consider an example poset $\rho = (X, \prec)$ with $X = \{x, y, z\}$ and a partial order defined such that $x \prec y$ and $x \prec z$. The language of ρ is $\mathcal{L}(\rho) = \{\langle x, y, z \rangle, \langle x, z, y \rangle\}$. Consider a labeling function $\gamma : X \to Y$ where $Y = \{a, b\}$ defined by $\gamma(x) = a$, $\gamma(y) = b$, and $\gamma(z) = b$; i.e., $\gamma = \{(x, a), (y, b), (z, b)\}$. Then $\rho' = (X, \prec, \gamma)$ is a labeled partial order over X and Y. The language of ρ' is $\mathcal{L}(\rho') = \{\langle a, b, b \rangle\}$.

3.2. Event log

We use Σ to denote the universe of activities. An *event log* $L \in \mathcal{M}(\Sigma^*)$ is a multi-set of sequences of activities. A *trace* $\sigma \in L$ is a sequence of activities representing the execution of a single process instance. For an event log $L \in \mathcal{M}(\Sigma^*)$, we define the following notations:

- $\Sigma_L = \{a \in \sigma \mid \sigma \in L\}$ is the set of activities in L.
- $L_{\triangleright} = \{\sigma(1) \mid \sigma \in L\}$ is the set of *start activities* in L.
- $L_{\square} = \{\sigma(|\sigma|) \mid \sigma \in L\}$ is the set of *end activities* in L.
- $\mapsto_L \subseteq \Sigma_L \times \Sigma_L$ is the *Directly-Follows Graph (DFG) relation* defined as follows: $a \mapsto_L b$ iff $\exists \sigma \in L, 1 \leq i < |\sigma|\ (\sigma(i) = a \ \wedge\ \sigma(i+1) = b)$. It indicates direct succession relationships between activities in L.
- $\rightsquigarrow_L \subseteq \Sigma_L \times \Sigma_L$ is the *Eventually-Follows Graph (EFG) relation* defined as follows: $a \rightsquigarrow_L b$ iff $\exists \sigma \in L, 1 \leq i < j \leq |\sigma|\ (\sigma(i) = a \ \wedge\ \sigma(j) = b)$. It indicates both direct and indirect succession relationships between activities in L.

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

3.3. Workflow net

A Petri net is a directed bipartite graph composed of *places* (visualized as circles) and *transitions* (visualized as boxes); i.e., there are no edges connecting two places or two transitions. A *labeled Petri net* is a Petri net with a labeling function mapping transitions into activities. Transitions represent instances of activities and places are added to model causal relations between activities. Multiple transitions can have the same label to model multiple occurrences of the same activity. We use $\tau \notin \Sigma$ to denote the *silent activity*. The silent activity can be used, for example, to model a choice between executing or skipping an activity.

Definition 1 (*Petri Net*[2])**.** Let T be a set of transitions and P be a set of places characterized by the transitions they connect, i.e., $P \subseteq \mathcal{P}(T) \times \mathcal{P}(T)$. The tuple $N = (P,T)$ is a Petri net. For a labeling function $l : T \to \Sigma \cup \{\tau\}$, $N = (P,T,l)$ is called a labeled Petri net.

Places hold *tokens* that are consumed and produced by *firing transitions*. A transition is considered *enabled* if each of its preceding places has at least one token. An enabled transition can *fire* (i.e., the activity can be executed). Firing a transition consumes one token from each of its preceding places and produces a token in each of its succeeding places. A *marking* is a multi-set of places indicating the number of tokens in each place to represent the state of the Petri net.

Let $N = (P,T,l)$ be a labeled Petri net. We define the following notations:

- For two markings $M, M' \in \mathcal{M}(P)$, we use $(N,M)[t\rangle(N,M')$ to denote that $t \in T$ is enabled in M and firing t results in the marking M'.
- For a sequence of transitions $\sigma = \langle t_1, \dots, t_n\rangle \in T^*$, $(N,M) \xrightarrow{\sigma} (N,M')$ denotes that there is a sequence of markings $M_0, M_1, \dots, M_n$ such that $M_0 = M$, $M_n = M'$, and $(N,M_i)[t_{i+1}\rangle(N,M_{i+1})$ for $0 \leq i < n$.
- $N_{\triangleright} = \{t \in T \mid \nexists (T_1,T_2) \in P\ (t \in T_2)\}$ is the set of *start transitions* of N, i.e., a start transition is not preceded by any place.
- $N_{\square} = \{t \in T \mid \nexists (T_1,T_2) \in P\ (t \in T_1)\}$ is the set of *end transitions* of N, i.e., an end transition is not followed by any place.

[2] A Petri net is generally defined as a triple (P,T,F) with disjoint sets of vertices T and P and a set of edges F. We omit F, and we fully characterize places by the transitions they connect, restricting ourselves to Petri nets with no duplicate places that connect the same sets of transitions.

Consider an example labeled Petri net $N = (\{p_1,p_2\}, \{t_1,t_2,t_3,t_4\}, l)$ with two places $p_1 = (\{t_1\},\{t_2,t_3\})$ and $p_2 = (\{t_2,t_3\},\{t_4\})$. In this Petri net, $N_{\triangleright} = \{t_1\}$ is the set of start transitions, and $N_{\square} = \{t_4\}$ is the set of end transitions. Starting from the marking $[p_1^2, p_2]$, executing the sequence of transitions $\langle t_2,t_3,t_4\rangle$ leads to the marking $[p_2^2]$, i.e., $(N,[p_1^2,p_2]) \xrightarrow{\langle t_2,t_3,t_4\rangle} (N,[p_2^2])$.

A *system net* is a Petri net with an initial and a final marking. The initial marking specifies the initial state of the process before executing any activity, while the final marking is reached at the end of a process execution. A *Workflow net (WF-net)* is a special type of system net such that every place/transition in the net is on a path from a unique source place to a unique sink place.

Definition 2 (*Workflow Net*)**.** Let (P,T,l) be a labeled Petri net and $s, f \in P$ be two places such that $s \neq f$. The tuple $W = (P,T,l,s,f)$ is a labeled workflow net if the following conditions hold:

- s is the unique source place; i.e., $\{s\} = \{(T_1,T_2) \in P \mid T_1 = \emptyset\}$.
- f is the unique sink place; i.e., $\{f\} = \{(T_1,T_2) \in P \mid T_2 = \emptyset\}$.
- All places and transitions are on a path from s to f.

Fig. 1(a) shows an example labeled WF-net. All places and transitions are on a path from the unique source place (marked by a token) to the unique sink place (marked by a double border).

Let $W = (P,T,l,s,f)$ be a labeled WF-net. We use $\lambda(W) = (P,T,l)$ to denote the *underlying labeled Petri net* of W; we use $\hat{\lambda}(W) = (P',T,l)$ with $P' = P \backslash \{s,f\}$ to denote the *reduced labeled Petri net of* W. The *language* of W is defined as $\mathcal{L}(W) = \{\langle l(\sigma_1), \dots, l(\sigma_n)\rangle \restriction_{\Sigma} \mid \sigma = \langle \sigma_1, \dots, \sigma_n\rangle \in T^* \ \wedge\ (\lambda(W), s) \xrightarrow{\sigma} (\lambda(W), f)\}$. We use $\mathcal{W}$ to denote the universe of labeled WF-nets.

Petri nets and similar graph-based process modeling notations may suffer from some quality anomalies. For instance, it is possible to construct a labeled WF-net with an unreachable final marking or with transitions that are not enabled in any reachable marking. Such models having undesirable behavioral properties are called *unsound*. We call a labeled WF-net *sound* if starting from the initial marking (1) each transition can be enabled at some point, (2) the final marking can always be reached, and (3) no place can have multiple tokens at the same time.

3.4. Inductive miner

Process discovery is an essential aspect of process mining, where the goal is to derive a process model from an event log. The event log, serving as the input, contains detailed records of the actual execution of processes within an organization. The goal of process discovery is to effectively translate the event log into a coherent process model that describes the behavior of the underlying process.

The inductive miner (IM) [2] is one of the leading approaches in process discovery. It is renowned for offering strong assurances such as the soundness of the discovered model, the ability to discover a model that encapsulates all the behaviors reported in the log, and the rediscoverability of specific process structures. The inductive miner has various adaptations, for example, to address incomplete or infrequent behaviors.

The inductive miner uses *process trees* to model processes. A process tree is a hierarchical modeling notation that captures behavioral dependencies between blocks of activities using control-flow operators. The operator $\rightarrow$ is used to demonstrate the sequential execution of different blocks; $\times$ indicates an exclusive choice; concurrency is depicted by $+$; and $\circlearrowleft$ illustrates a do-redo loop between two blocks (that is, the do-part is executed initially, and every completed execution of the redo-part triggers another execution of the do-part).

The inductive miner employs a recursive, top-down strategy. The algorithm tries to identify a *cut*, which involves recognizing a behavioral pattern within the event log and splitting the activities based on

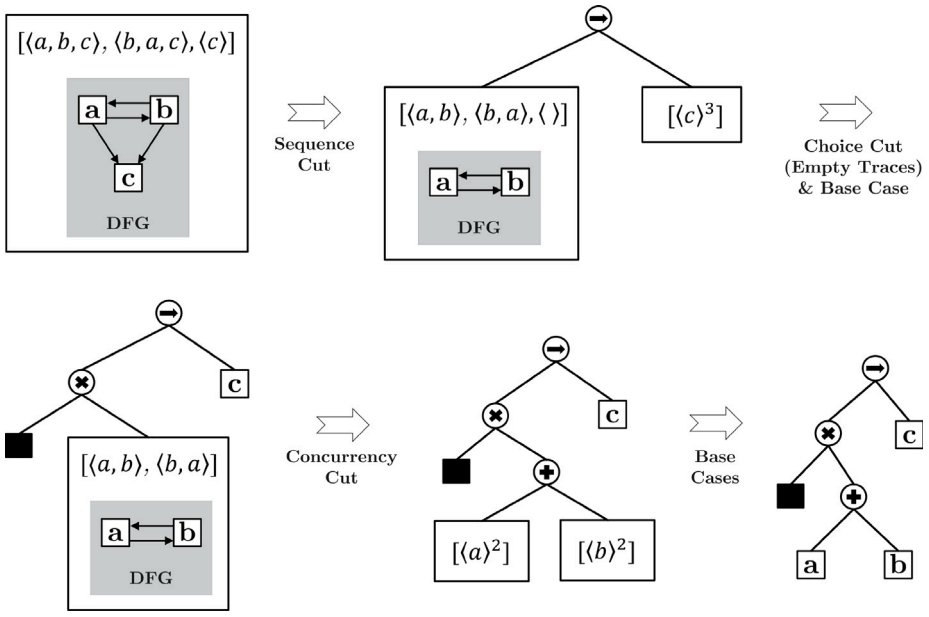


Fig. 2. Application of the inductive miner on an example event log.

this pattern. The inductive miner recognizes four types of cuts that correspond to the four main operators in process trees. Formally, a process tree cut over an event log is a tuple $(\oplus, A_1, \dots, A_n)$ of a control-flow operator $\oplus \in \{\rightarrow, \times, +, \circlearrowleft\}$ and a partitioning of the activities of L into $n \geq 2$ subsets ($n = 2$ for $\circlearrowleft$).

The inductive miner iteratively constructs a process tree based on the identified cuts. After detecting a cut, the event log is projected into the different subsets of activities, creating several smaller sub-logs. This method is then recursively applied to each sub-log until the process reaches a *base case*, which is an empty event log or an event log that consists of a single trace variant of length 1. A base case can be trivially converted into a process tree that consists of a single activity or a silent activity.

The step of process tree cut detection is mainly based on the directly-follows graph. For instance, a choice cut is detected if no activity in any partitioning is directly followed by an activity from another partitioning. However, other factors are also taken into consideration. For instance, the inductive miner ensures that all start and end activities of the event log are in the do-part of a loop cut. Another example is the detection of empty traces within the event log. If empty traces exist, then the inductive miner returns a choice cut between τ and the activities of the log, and it filters out the empty traces from the projected log.

Fig. 2 illustrates the application of the inductive miner on the event log $L = [\langle a, b, c\rangle, \langle b, a, c\rangle, \langle c\rangle]$. Initially, the sequence cut $(\rightarrow, \{a, b\}, \{c\})$ is discovered, and the log is projected to create two sub-logs. The choice cut $\times(\tau, \{a, b\})$ is discovered for the first sub-log as it contains empty traces, while the second sub-log is a base case. After filtering out empty traces, the concurrency cut $(+, \{a\}, \{b\})$ is detected, indicating that a and b can be executed in parallel. This leads to the creation of two sub-logs that represent base cases, and the algorithm terminates returning the process tree $\rightarrow(\times(\tau, +(a, b)), c)$.

In cases where neither a base case nor a cut is identified, the inductive miner utilizes a *fall-through* function. This function invariably identifies a cut, potentially leading to an under-fitting model (i.e., a model that might not precisely represent the log's behavior), allowing for the continuation of the recursion. For instance, this function might return a concurrency cut between an activity occurring once in every trace and the rest of the activities. For a detailed description of the different steps of the inductive miner, we refer to [2].

4. POWL language

The partially ordered workflow language [3] extends partially ordered graphs with control-flow operators for modeling choice and loop structures. A single activity is a POWL model (*base case*). Multiple POWL models can be combined into a new model by using one of the control-flow operators $\times$ and $\circlearrowleft$ or by defining a partial order over the sub-models.

Definition 3 (*POWL Model [3]*)**.** A POWL model is recursively defined as follows:

- 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 $P = \{\psi_1, \dots, \psi_n\}$ be a set of $n \geq 2$ POWL models.
  - $\times(\psi_1, \dots, \psi_n)$ is a POWL model.
  - A poset $\rho \in \Pi(P)$ is a POWL model.

We use Ψ to denote the universe of POWL models. The semantics of a POWL model are defined by transforming it into a set of labeled partial orders, with activities as labels, over a set of newly generated nodes. We call these nodes *transitions* as they represent instances of activities. For example, the POWL model shown in Fig. 1(e) can be transformed into the set of labeled partial orders illustrated in Fig. 3.

A POWL model is recursively transformed into a set of partial orders. A single activity is transformed into a single partial order with a transition having the activity as a label or into an empty labeled partial order if the activity is silent. For the choice operator $\times$, we transform the POWL model into the union of the partially ordered sets generated for the sub-models. For $\circlearrowleft(\psi_1, \psi_2)$, the labeled partial orders of the sub-models are combined such that the first order is from ψ_1 and each order from ψ_2 is followed by another order from ψ_1. For the case of a partial order over a set of POWL models, the orders generated for the sub-models are combined such that the partial order of the sub-models is preserved.

Notation. We define the following auxiliary notations for Definition 4:

- For combining the orders in the loop case, we define an index transformation function $\tilde{i}$ mapping odd numbers into the do-part ($\tilde{i} = 1$ for $i \in \mathbb{N}_{odd}$) and even number into the redo-part ($\tilde{i} = 2$ for $i \in \mathbb{N}_{even}$).

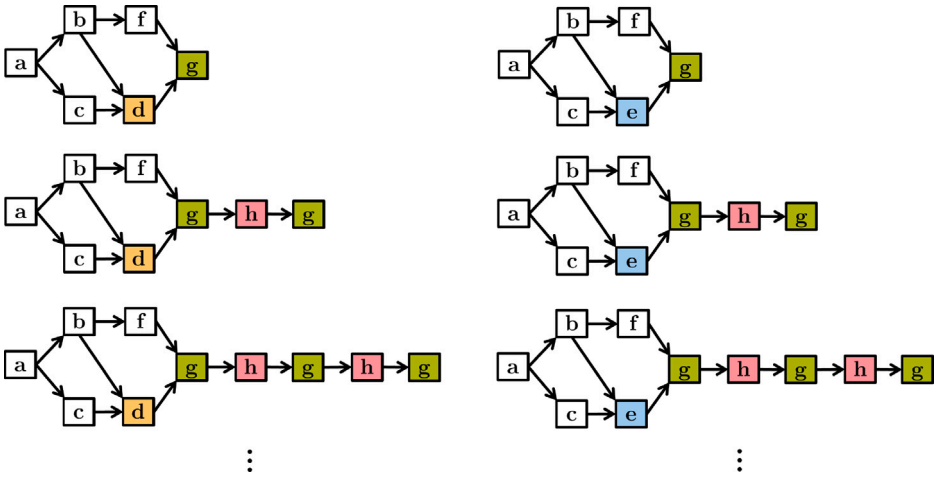


Fig. 3. A set of labeled partial orders modeling the behavior of the POWL model shown in Fig. 1(e). The choice is modeled by duplicating the order to cover the cases with the activity d (colored orange) on the left and the cases with the activity e (colored light blue) on the right. The cyclic behavior is modeled by duplicating the order to cover loops of different lengths (the do-part is colored green, while the redo-part is colored red). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

- Let $(T_1, \prec_1), \dots, (T_n, \prec_n)$ be n labeled posets of transitions. We use $\mathcal{T}_{\rightarrow}(\prec_1, \dots, \prec_n)$ to denote the set of all posets of new transitions $(T, \prec)$ such that a bijective functions $f : \bigcup_{1 \leq i \leq n} T_i \rightarrow T$ exists and the following condition holds for all $1 \leq i \leq n, 1 \leq j \leq n, t \in T_i, t' \in T_j$:

  $f(t) \prec f(t') \Leftrightarrow i < j \vee (i = j \wedge t \prec_i t')$.

Definition 4 (*POWL Semantics [3]*)**.** Let $\mathcal{T}$ be the universe of transitions. $\Gamma : \Psi \Rightarrow \mathcal{P}(\Pi(\mathcal{T}, \Sigma))$ 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 fresh transition.
- $\Gamma(\tau) = \{(\emptyset, \emptyset, \emptyset)\}$.
- Let $P = \{\psi_1, \dots, \psi_n\}$ be a set of $n \geq 2$ POWL models.[3]
  - $\Gamma(\times(\psi_1, \dots, \psi_n)) = \bigcup_{1 \leq i \leq n} \Gamma(\psi_i)$.
  - $\Gamma(\circlearrowleft(\psi_1, \psi_2)) = \bigcup_{n \in \mathbb{N}_{odd}} \{ (T, \prec, \gamma) \mid \exists (T_1, \prec_1, \gamma_1) \in \Gamma(\psi_1), \dots, (T_n, \prec_n, \gamma_n) \in \Gamma(\psi_{\tilde{n}}) \; ((T, \prec) \in \mathcal{T}_{\rightarrow}(\prec_1, \dots, \prec_n) \wedge \forall 1 \leq i \leq n, \; t \in T_i \; (\gamma_i(t) = \gamma(f(t)))) \}$.
  - For a poset $\rho = (P, \prec)$, $\Gamma(\rho) = \{ (T, \prec', \gamma) \mid \exists (T_1, \prec_1, \gamma_1) \in \Gamma(\psi_1), \dots, (T_n, \prec_n, \gamma_n) \in \Gamma(\psi_n) \; (T = \bigcup_{1 \leq i \leq n} T_i \wedge \forall 1 \leq i \leq n, t \in T_i \; (\gamma(t) = \gamma_i(t) \wedge \forall 1 \leq j \leq n, t' \in T_j \; (t \prec' t' \Leftrightarrow ((i = j \wedge t \prec_i t') \vee \psi_i \prec \psi_j)))) \}$.

The *language* of a POWL model is the set of activity sequences that can be generated by the model. After transforming a POWL model ψ into a set of labeled partial orders $\Gamma(\psi)$, we can derive its language $\mathcal{L}(\psi) = \{\sigma \in \mathcal{L}(\rho) \mid \rho \in \Gamma(\psi)\}$.

5. Transforming POWL models into workflow nets

POWL models can be recursively transformed into WF-nets. In this section, we present an approach for transforming a POWL model into a labeled WF-net modeling the same behavior, and we prove that the generated WF-net is sound. This shows that POWL models inherently guarantee soundness.

[3] Since we always use newly generated transitions, we assume the sets of generated transitions for the sub-models to be pairwise disjoint.

The POWL to WF-net converter is a recursively defined function $\mathcal{C} : \Psi \Rightarrow \mathcal{W}$ that transforms any POWL model into a WF-net. We define the base case in Definition 5. For the combination of multiple POWL models (using $\times$, $\circlearrowleft$, or a poset), the WF-net is generated as described in Definition 6, Definition 7, or Definition 8. The POWL to WF-net converter $\mathcal{C}$ is schematically presented in Fig. 4.

For the base case, the POWL to WF-net converter $\mathcal{C}$ maps an activity $a \in \Sigma \cup \{\tau\}$ into a WF-net with a single transition having the label a.

Definition 5 (*Activity to WF-Net Converter*)**.** Let $\psi = a \in \Sigma \cup \{\tau\}$. The converter $\mathcal{C}$ transforms ψ into the WF-net $\mathcal{C}(\psi) = (\{s, f\}, \{t\}, \{(t, a)\}, s, f)$ where t is a fresh transition, $s = (\emptyset, \{t\})$, and $f = (\{t\}, \emptyset)$.

Lemma 1. *Let $\psi = a \in \Sigma \cup \{\tau\}$ be a POWL model. $\mathcal{L}(\psi) = \mathcal{L}(\mathcal{C}(\psi))$.*

Proof. Both languages consist of a single trace: $\langle a \rangle$ in case $a \in \Sigma$ and $\langle\rangle$ in case $a = \tau$. □

A composition of POWL models (using $\times$, $\circlearrowleft$, or a poset) can be modeled as a WF-net by first recursively transforming all sub-models into WF-nets. Then, the reduced Petri nets of the WF-nets are derived (the reduced Petri net $\hat{\lambda}(W)$ is derived by removing the source and sink places of a workflow net W). These reduced Petri nets are combined by using additional places (including a unique source place and a unique sink place) and silent transitions. For the choice operator $\times$, we connect the source place to all start transitions, and we connect all end transitions to the sink place.

Definition 6 (*XOR to WF-Net Converter*)**.** Let $\psi_1, \dots, \psi_n$ be $n \geq 2$ POWL models and $\psi = \times(\psi_1, \dots, \psi_n)$. Let $N_i = (P_i, T_i, l_i) = \hat{\lambda}(\mathcal{C}(\psi_i))$ for $1 \leq i \leq n$ with pairwise disjoint transitions, i.e., $T_i \cap T_j = \emptyset$ for $1 \leq i < j \leq n$. The WF-net $\mathcal{C}(\psi) = (P, T, l, s, f)$ is generated as follows:

- $T = \bigcup_{1 \leq i \leq n} T_i$ with $l(t) = l_i(t)$ for $t \in T_i, 1 \leq i \leq n$.
- $P = \{s, f\} \cup \bigcup_{1 \leq i \leq n} P_i$ with $s = (\emptyset, \bigcup_{1 \leq i \leq n} N_{i\triangleright})$ and $f = (\bigcup_{1 \leq i \leq n} N_{i\square}, \emptyset)$.

Lemma 2. *Let $\psi = \times(\psi_1, \dots, \psi_n)$ be a POWL model. If $\mathcal{C}(\psi_i)$ is sound and $\mathcal{L}(\psi_i) = \mathcal{L}(\mathcal{C}(\psi_i))$ for all $1 \leq i \leq n$, then $\mathcal{L}(\psi) = \mathcal{L}(\mathcal{C}(\psi))$.*

Proof. In the initial marking $[s]$, all sub-nets are enabled, and one of them fires, consuming the token. After the termination of the firing sub-net, it produces a token in the final place, leading to the final marking $[f]$. Therefore, the language of N is the union of the languages

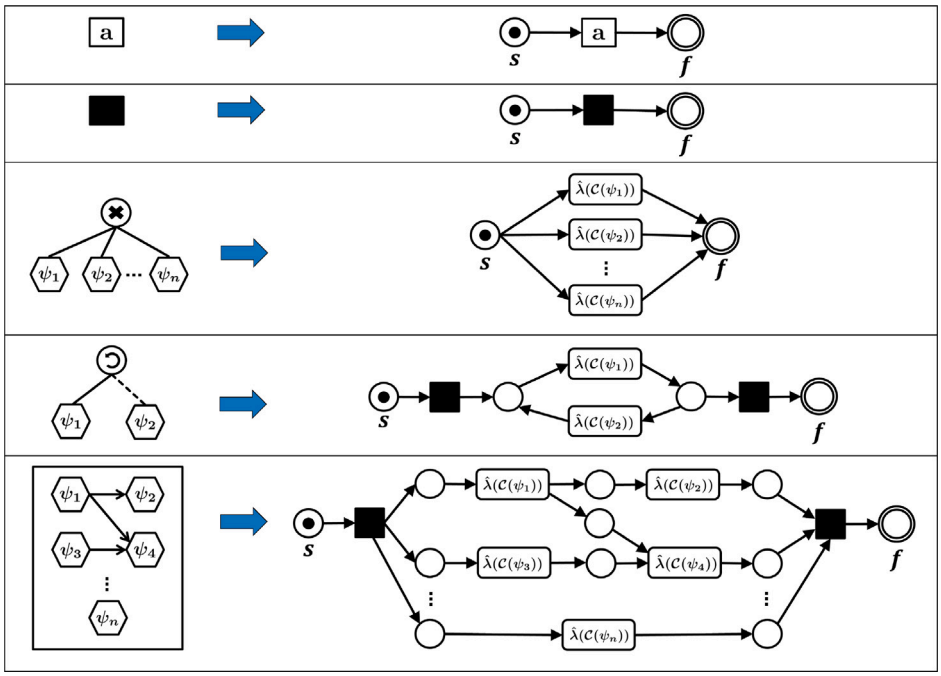


Fig. 4. POWL to WF-net converter C.

of $\psi_i, \dots, \psi_i$. This is the same language described by the POWL model $\times(\psi_1, \dots, \psi_n)$. □

We model a loop as a WF-net using two silent transitions and two places (in addition to the sink and source places), and we connect them as illustrated in Fig. 4, ensuring that the do-part in executed first and the redo-part is always followed by another execution of the do-part.

Definition 7 (*Loop to WF-Net Converter*)**.** Let ψ_1 and ψ_2 be two POWL models and $\psi = \circlearrowleft(\psi_1, \psi_2)$. Let $N_1 = (P_1, T_1, l_1) = \hat{\lambda}(C(\psi_1))$ and $N_2 = (P_2, T_2, l_2) = \hat{\lambda}(C(\psi_2))$ with $T_1 \cap T_2 = \emptyset$. The WF-net $C(\psi) = (P, T, l, s, f)$ is generated as follows:

- $T = T_1 \cup T_2 \cup \{t_{in}, t_{out}\}$ with transitions $t_{in}, t_{out} \notin T_1 \cup T_2$.
- $l(t_{in}) = l(t_{out}) = \tau$ and $l(t) = l_i(t)$ for $t \in T_i, 1 \leq i \leq 2$.
- $P = P_1 \cup P_2 \cup \{s, p_{do}, p_{redo}, f\}$ with $s = (\emptyset, \{t_{in}\})$, $p_{do} = (\{t_{in}\} \cup N_{2\square}, N_{1\triangleright})$,$p_{redo} = (N_{1\square}, N_{2\triangleright} \cup \{t_{out}\})$, and $f = (\{t_{out}\}, \emptyset)$.

Lemma 3. *Let $\psi = \circlearrowleft(\psi_1, \psi_2)$ be a POWL model. If $C(\psi_i)$ is sound and $\mathcal{L}(\psi_i) = \mathcal{L}(C(\psi_i))$ for all $1 \leq i \leq 2$, then $\mathcal{L}(\psi) = \mathcal{L}(C(\psi))$.*

Proof. In the initial marking $[s]$, only t_{in} is enabled. After firing t_{in}, we reach the marking $[p_{do}]$. Afterward, N_1 is enabled. After the termination of N_1, we reach the marking $[p_{redo}]$. In $[p_{redo}]$, there is a choice between firing t_{out} or N_2. If t_{out} is fired, we reach the final marking $[f]$. Otherwise, after the termination of N_2, we return to the marking $[p_{do}]$. In other words, every time N_2 is fired, N_1 is re-enabled after its termination. This is the same language described by the POWL model $\circlearrowleft(\psi_1, \psi_2)$. □

In order to transform a poset into a WF-net, we model an AND-split between the start models (using a place preceding each start model and a silent transition) and an AND-join between the end models (using a place following each end model and a silent transition). Finally, each edge of the transitive reduction of the partial order is modeled through a place connecting the end transitions of the first model with the start transitions of the second model. We only consider the transitive reduction of the partial order in order to avoid adding places that are implied by other places (i.e., places that do not add further behavioral restrictions).

Definition 8 (*Poset to WF-Net Converter*)**.** Let $\psi = (\{\psi_1, \dots, \psi_n\}, \prec)$ be a poset over a set of $n \geq 2$ POWL models. Let $N_i = (P_i, T_i, l_i) = \hat{\lambda}(C(\psi_i))$ for $1 \leq i \leq n$ with pairwise disjoint transitions, i.e., $T_i \cap T_j = \emptyset$ for $1 \leq i < j \leq n$. The WF-net $C(\psi) = (P, T, l, s, f)$ is generated as follows:

- $T = \{t_{split}, t_{join}\} \cup \bigcup_{1 \leq i \leq n} T_i$ with two transitions $t_{split}, t_{join} \notin \bigcup_{1 \leq i \leq n} T_i$.
- $l(t_{split}) = l(t_{join}) = \tau$ and $l(t) = l_i(t)$ for $t \in T_i, i \in \{1, \dots, n\}$.
- $s = (\emptyset, \{t_{split}\})$ and $f = (\{t_{join}\}, \emptyset)$.
- $N_{split} = \{N_j \mid 1 \leq j \leq n \wedge \nexists 1 \leq i \leq n\ (\psi_i \prec \psi_j)\}$.
- $P_{split} = \{(t_{split}, N_\triangleright) \mid N \in N_{split}\}$.
- $N_{join} = \{N_i \mid 1 \leq i \leq n \wedge \nexists 1 \leq j \leq n\ (\psi_i \prec \psi_j)\}$.
- $P_{join} = \{(N_\square, t_{join}) \mid N \in N_{join}\}$.
- $P_{order} = \{(N_{i\square}, N_{j\triangleright}) \mid i, j \in \{1, \dots, n\} \wedge \psi_i \prec^- \psi_j\}$.
- $P = \{s, f\} \cup P_{split} \cup P_{join} \cup P_{order} \cup \bigcup_{1 \leq i \leq n} P_i$.

Lemma 4. *Let $\psi = (\{\psi_1, \dots, \psi_n\}, \prec)$ be a poset over a set of POWL models. If $C(\psi_i)$ is sound and $\mathcal{L}(\psi_i) = \mathcal{L}(C(\psi_i))$ for all $1 \leq i \leq n$, then $\mathcal{L}(\psi) = \mathcal{L}(C(\psi))$.*

Proof. In the initial marking $[s]$, only t_{split} is enabled. After firing t_{split}, a token is produced in all places of P_{split}, i.e., all start sub-nets $N' \in N_{split}$ are enabled and they all can fire simultaneously. After the termination of a firing sub-net, it will produce a token in all of its succeeding places. Here we distinguish two cases: (a) a final sub-net $N' \in N_{join}$ will produce a token in one the places of P_{join} after its termination, while (b) other sub-nets $N' \notin N_{join}$ will produce tokens in the places of P_{order}. The places of P_{order} model the partial order; i.e., for each $\psi_i \prec^- \psi_j$, we have a place $p = (N_{i\square}, N_{j\triangleright}) \in P_{order}$ modeling the firing order between N_i and N_j. Each sub-net fires exactly once, and after the termination of all sub-nets, each place of P_{join} will contain one token. This enables t_{join}, and firing t_{join} will lead to the final marking $[f]$. To sum up, an accepted trace by N is an interleaving of n traces $\sigma_i \in \mathcal{L}(\psi_i)$ for $1 \leq i \leq n$ in such a way that the execution order of activities adheres to the partial order $\prec$. This is the same language modeled by the POWL model $\psi = (\{\psi_1, \dots, \psi_n\}, \prec)$. □

Next, we prove that the POWL to WF-net converter C generates a WF-net that is guaranteed to be sound (Theorem 1) and describes the same language as the POWL model (Theorem 2).

The composition of WF-nets preserves soundness; i.e., if we replace a transition in a sound WF-net W_1 by the *reduced Petri net* $\hat{\lambda}(W_2)$ of another sound WF-net, then the resulting WF-net is sound as well.

Notation. For two sets X_1 and X_2, we define $X_1|_{x \to X_2} = X_1$ if $x \notin X_1$, and $X_1|_{x \to X_2} = (X_1 \backslash \{x\}) \cup X_2$ if $x \in X_1$.

Lemma 5 (*WF-Net Compositionality [21]*). *Let W_1 and W_2 be sound WF-nets with $N_1 = \hat{\lambda}(W_1) = (P_1, T_1, l_1)$, $N_2 = \hat{\lambda}(W_2) = (P_2, T_2, l_2)$, $T_1 \cap T_2 = \emptyset$, and $t^+ \in T_1$. Let $W_3 = (P_3, T_3, l_3, s, f)$ be the WF-net obtained by replacing t^+ in W_1 by N_2, i.e., $T_3 = T_1 \cup T_2 \backslash \{t^+\}$, $l_3(t) = l_1(t)$ for $t \in T_1 \backslash \{t^+\}$, $l_3(t) = l_2(t)$ for $t \in T_2$, $P_3 = \{(X_1|_{t^+ \to N_{2\square}}, X_2|_{t^+ \to N_{2\triangleright}}) \mid (X_1, X_2) \in P_1\} \cup P_2$, $s = (\emptyset, N_{1\triangleright}|_{t^+ \to N_{2\triangleright}})$, and $f = (N_{1\square}|_{t^+ \to N_{2\square}}, \emptyset)$. W_3 is sound.*

Proof. See [21, Theorem 3]. □

Theorem 1 (*Soundness Guarantee of POWL to WF-Net Converter*). *Let $\psi \in \Psi$ be a POWL model. $C(\psi)$ is sound.*

Proof. We prove this theorem by induction. For the base case ($\psi = a \in \Sigma \cup \{\tau\}$), $C(a)$ is trivially sound. Let us assume that the lemma is proven for $n \geq 2$ POWL models $\psi_1, \ldots, \psi_n$. Let ψ be the composition of the POWL models as a poset or using one of the operator $\times$ and $\circlearrowleft$ (with $n = 2$ for $\circlearrowleft$). Let W' be a WF-net generated by replacing all sub-nets $\hat{\lambda}(C(\psi_1)), \ldots, \hat{\lambda}(C(\psi_n))$ in $W = C(\psi)$ by single transitions $t_1, \ldots, t_n$. W' is trivially sound since W' models an exclusive choice, a do-redo loop, or a strict partial order over $\{t_1, \ldots, t_n\}$. The soundness of W follows from the soundness of W' and Lemma 5. □

Theorem 2 (*Language Equivalence Guarantee of POWL to WF-Net Converter*). *Let $\psi \in \Psi$ be a POWL model. $\mathcal{L}(\psi) = \mathcal{L}(C(\psi))$.*

Proof. We prove this theorem by induction. The base case ($\psi = a \in \Sigma \cup \{\tau\}$) is proven by Lemma 1. Let us assume that the lemma is proven for $n \geq 2$ POWL models $\psi_1, \ldots, \psi_n$. The soundness of $C(\psi_i)$ for $1 \leq i \leq n$ is proven by Theorem 1. Therefore, the statement is proven by Lemma 2 for $\psi = \times(\psi_1, \ldots, \psi_n)$; by Lemma 3 for $\psi = \circlearrowleft(\psi_1, \psi_2)$; and by Lemma 4 for a poset $\psi = (\{\psi_1, \ldots, \psi_n\}, \prec)$. □

6. Discovery of POWL models

POWL models allow for an explicit representation of partial orders, providing a compact depiction of the flow of activities within processes. Their capability to represent complex, non-hierarchical relationships between activities makes them particularly suited for process discovery challenges.

In this section, we present several approaches for the discovery of POWL models from event logs. We extend the inductive miner (IM) with an additional step to mine for partial orders. This step takes place after attempting to discover process tree cuts and before triggering the fall-through function.

We introduce the following three approaches within the *POWL inductive mining framework*:

- **POWL Inductive Miner - Brute-Force** (PM_{bf}): Introduced in [3], this is the base approach that serves as a proof-of-concept to demonstrate the practicality of applying POWL models in process discovery. As illustrated in Fig. 5(a), PM_{bf} employs a brute-force approach that generates and validates partial orders over all possible partitionings of activities. This leads to scalability problems for event logs that contain a large number of activities.
- **POWL Inductive Miner - Maximal Order** (PM_{max}): This approach, illustrated in Fig. 5(b), was introduced in [4] to enhance efficiency by mining for a unique partial order, introducing the concepts of completeness and maximality. PM_{max} offers formal guarantees, ensuring that the discovered partial order is the most comprehensive representation of the event log's behavior.
- **POWL Inductive Miner - Dynamic Clustering** (PM_{clst}): This new approach relaxes the stringent validity requirements of the previous methods. It is designed to reduce dependence on the fall-through function by accommodating a wider spectrum of partial orders. As illustrated in Fig. 5(c), PM_{clst} mines for partial order by dynamically clustering subsets of activities, ensuring high scalability on large data sets. Additionally, this approach moves away from the reliance on process tree cuts for detecting sequences and concurrency (cf. Figs. 5(a) and 5(b)), and it directly identifies these structures as partial orders.

In the remainder of this section, we delve deeper into each of these three POWL inductive mining approaches.

6.1. POWL Inductive Miner - Brute-Force (PM_{bf})

As illustrated in Fig. 5(a), PM_{bf} extends the inductive miner to mine for partial orders. If no process tree cut is detected, then, before invoking the fall-through function, PM_{bf} mines for a partial order that fulfills certain validity criteria.

6.1.1. Validity of partial orders

We define a *partial order cut* as a partitioning of the activities of the event log into subsets, along with a partial order over this partitioning.

Definition 9 (*Partial Order Cut [3]*). Let $L \in \mathcal{M}(\Sigma^*)$ be an event log. A partial order cut of L is a poset $\rho = (P, \prec) \in \Pi(P)$ over a partitioning of the activities $P \in Part_n(\Sigma_L)$ into $n \geq 2$ subsets.

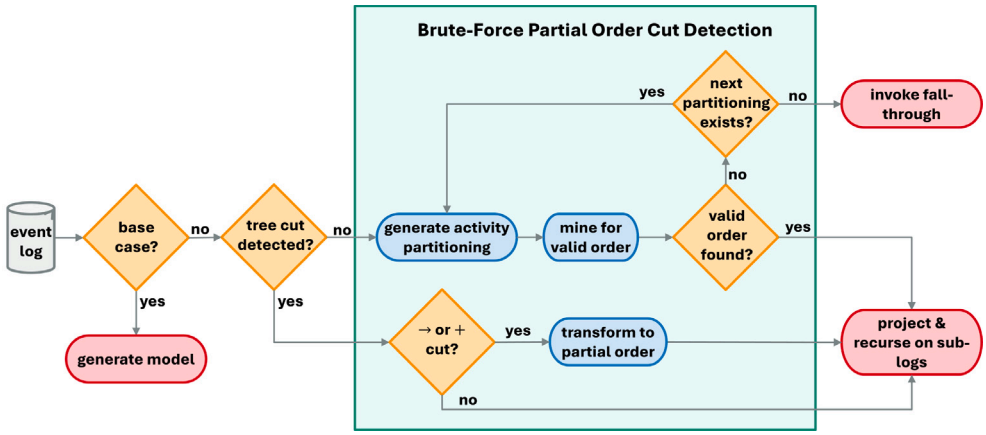
The key to our approach is the definition and identification of *valid* partial order cuts. We aim to ensure that the partial order accurately reflects the behavioral patterns observed in the event log. We define validity criteria based on the eventually-follows graphs and the start and end activities of the event log. We use the eventually-follows graph instead of the directly-follows graph because our goal is to discover a partial order (i.e., a transitive, irreflexive relation) capturing both direct and indirect dependencies between activities.

The validity of a partial order cut $(P, \prec)$ is defined based on multiple criteria. First, we ensure that the order conforms with the eventually-follows dependencies: $A_i \prec A_j$ if and only if all activities in A_i are eventually followed by all activities of A_j, while no activity in A_j is eventually followed by an activity from A_i. Second, two groups are concurrent if and only if every activity of each group eventually follows all activities of the other group. Finally, groups with no preceding groups in the order include start activities, while those without succeeding groups include end activities.

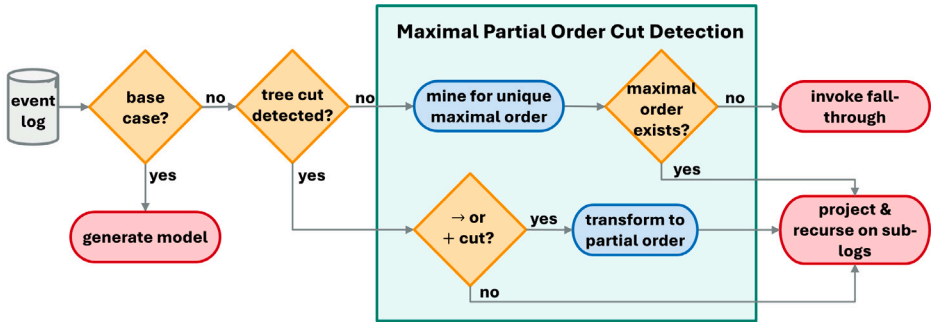
Definition 10 (*Valid Partial Order Cut [3]*). Let $\rho = (P, \prec)$ be a partial order cut of an event log $L \in \mathcal{M}(\Sigma^*)$. ρ is valid if the following conditions hold for all $A_i, A_j \in P$; $A_i \neq A_j$:

1. $(A_i \prec A_j \wedge A_j \not\prec A_i)$ iff $\forall a_i \in A_i, a_j \in A_j \ (a_i \nrightarrow_L a_j \wedge a_j \not\nrightarrow_L a_i)$.
2. $(A_i \not\prec A_j \wedge A_j \not\prec A_i)$ iff $\forall a_i \in A_i, a_j \in A_j \ (a_i \nrightarrow_L a_j \wedge a_j \nrightarrow_L a_i)$.
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 $A_i \cap L_\square \neq \emptyset$.

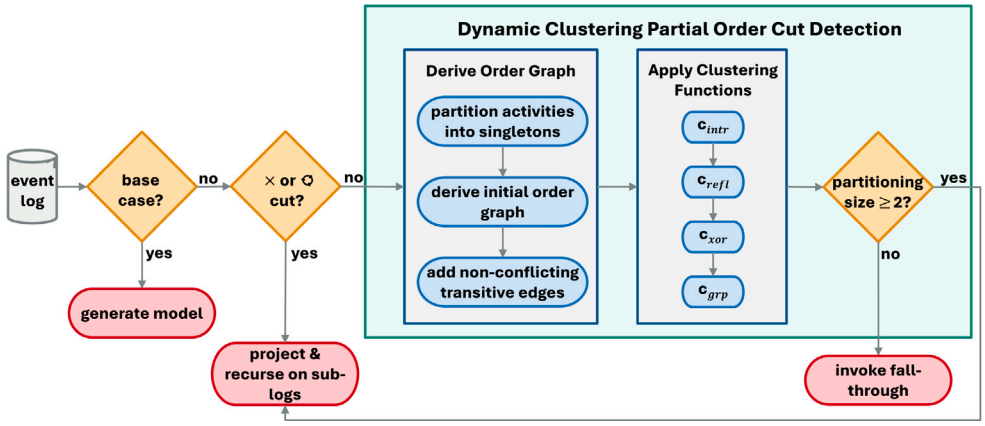
Note that the first condition of Definition 10 uniquely defines a binary relation. This means that for a given partitioning of activities, if a valid partial order cut exists, then it is unique.



(a) PM_{bf} generates and validates partial orders over all possible partitionings of activities after failing to discover a process tree cut.



(b) PM_{max} introduces a notion of maximality to mine for a unique partial order.



(c) PM_{clst} generates an order graph and dynamically clusters subsets of activities to derive a partial order. It detects sequences and concurrency within the discovered partial orders, not as process tree cuts.

Fig. 5. Overview of our three approaches for the discovery of POWL models.

6.1.2. Approach

The steps of the brute-force POWL inductive miner (PM_{bf}) are outlined in Alg.1. If a base case is detected, the algorithm returns a single activity or a silent activity (cf. lines 2–5). Afterward, the algorithm mines for a process tree cut. If a choice or loop cut is found, then the process tree operator × or ↻ is used to create a POWL model, and the event log is projected into the subsets of the partitioning to continue the recursion. If a sequence or concurrency cut is found instead, PM_{bf} transform it into a partial order cut. A concurrency cut is modeled as a poset with no ordering relation (cf. lines 11–12), while a sequence cut is transformed using the sequential order of the nodes (cf. lines 13–15). If no process tree cut is detected, PM_{bf} mines for a valid partial order cut. The step of partial order cut detection follows a brute-force approach (cf. lines 16–22). PM_{bf} generates partial orders over all possible partitionings of activities until a valid partial order cut is found. If a *valid* order is detected, then the event log is projected on the subsets of activities, and the recursion continues on the sub-logs; otherwise, the fall-through function is invoked. Note that we adapt the fall-through function [2] to return POWL models instead of process trees (similarly to the conversion of process tree cuts in lines 6–15).

Fig. 6 shows four partial order cuts generated for the event log $L = [\langle a, b, d\rangle, \langle a, b, c, b, d\rangle, \langle b, d\rangle, \langle b, c, b, d\rangle]$. The cuts are defined over different partitionings of the activities (cf. line 17 in Alg.1). The first cut ρ_1 violates the second and third validity conditions defined in Definition 10 (c is not eventually followed by a and c is not a start activity). The other three cuts are valid partial order cuts.

As asserted in [2], all steps of the base inductive miner are designed to be *fitness-preserving*, i.e., all traces in the input event log are included in the language of the discovered model. The brute-force POWL inductive miner ensures perfect fitness as well. The step of partial order

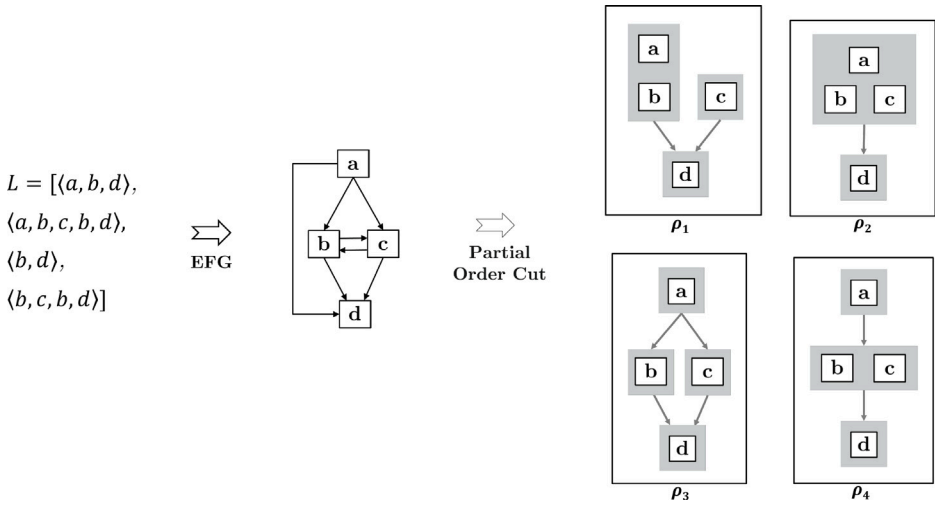


Fig. 6. An example event log and four partial order cuts detected for different partitionings of the activities. The first cut violates the validity requirements, while the other three cuts are valid.

Input: an event log L.
Output: a POWL model $PM_{bf}(L)$.

begin
  if $L = [\langle a \rangle^f]$ *for* $a \in \Sigma$ *and* $f \in \mathbb{N}$ **then**
    return a
  if $L = []$ **then**
    return τ
  if *a process tree cut* $\oplus \in \{\times, \rightarrow, +, \circlearrowleft\}$ *is detected* **then**
    $L_1, ..., L_n \leftarrow$ project L on the subsets of the partitioning
    $\psi_1, ..., \psi_n \leftarrow PM_{bf}(L_1), ..., PM_{bf}(L_n)$
    if $\oplus \in \{\times, \circlearrowleft\}$ **then**
      return $\oplus(\psi_1, ..., \psi_n)$
    if $\oplus = +$ **then**
      return $(\{\psi_1, ..., \psi_n\}, \emptyset)$
    if $\oplus = \rightarrow$ **then**
      $\prec \leftarrow \{(\psi_i, \psi_j) \mid 1 \leq i < j \leq n\}$
      return $(\{\psi_1, ..., \psi_n\}, \prec)$
  for $P \in Part(\Sigma_L)$ **do**
    $\prec = \{(A_i, A_j) \in P \times P \mid \forall a_i \in A_i, a_j \in A_j \ (a_i \nrightarrow_L a_j \wedge a_j \not\nrightarrow_L a_i)\}$
    if $(P, \prec)$ *is a valid partial order cut* **then**
      $L_1, ..., L_n \leftarrow$ project L on the subsets of $P = \{A_1, ..., A_n\}$
      $\psi_1, ..., \psi_n \leftarrow PM_{bf}(L_1), ..., PM_{bf}(L_n)$
      $\prec' \leftarrow \{(\psi_i, \psi_j) \mid i, j \in \{1, ..., n\} \ \wedge \ A_i \prec A_j\}$
      return $(\{\psi_1, ..., \psi_n\}, \prec')$
  $\psi \leftarrow$ FallThrough(L)
  return ψ

Algorithm 1: POWL Inductive Miner - Brute-Force (PM_{bf}). The algorithm takes an event log as input and returns a POWL model. The partial order cut detection step involves generating and validating partial order cuts over all possible partitionings of activities.

cut detection preserves the fitness guarantee as it only mines for valid partial order cuts.

Theorem 3 (*Fitness Guarantee of* PM_{bf}**).** *Let* $L \in \mathcal{M}(\Sigma^*)$ *be an event log and let* $PM_{bf}(L)$ *be the POWL model discovered using Alg.1. Every trace* $\sigma \in L$ *is included in the model's language, i.e.,* $\sigma \in \mathcal{L}(PM_{bf}(L))$.

Proof. We prove this theorem by induction on the recursive application of the algorithm Alg.1 as it constructs the POWL model.

(Base Case) If $L = [\langle a \rangle^f]$ for some $a \in \Sigma$ and $f \in \mathbb{N}$, then $PM_{bf}(L) = a$. The language of this model is $\mathcal{L}(a) = \{\langle a \rangle\}$, which includes the trace $\langle a \rangle$. If $L = []$, then no $\sigma \in L$ exists.

(Inductive Step) Assume the theorem holds for the outcome of any previously processed input by PM_{bf}. This assumption means that any sub-model generated by the algorithm for a sub-log already includes all corresponding traces in its language. As the algorithm processes L, we distinguish three cases based on the nature of the operations it performs:

- **In case a process tree cut is detected:**
  Process tree cuts are fitness-preserving [2]. To prove that all traces in L are included in the language of $PM_{bf}(L)$, we still need to show that transforming sequence and concurrency cuts into partial order cuts yields the same language:
  - The concurrency operator $+(\psi_1, \dots, \psi_n)$ interleaves sub-traces from the sub-models while maintaining the order within each sub-trace [2]. This language is equivalent to the language of the poset $(\{\psi_1, \dots, \psi_n\}, \emptyset)$. This poset has an empty partial order, which permits the interleaving of sub-traces without altering the internal order of each sub-trace.
  - The sequence operator $\rightarrow(\psi_1, \dots, \psi_n)$ concatenates sub-traces from the sub-models using the sequential order of the sub-models in the detected cut [2]. This language is the same as the language of the poset $\rho = (\{\psi_1, \dots, \psi_n\}, \{(\psi_i, \psi_j) \mid 1 \leq i < j \leq n\})$: $\mathcal{L}(\rho) = \mathcal{L}(\rightarrow(\psi_1, \dots, \psi_n)) = \{\sigma_1 \cdot ... \cdot \sigma_n \mid \sigma_i \in \mathcal{L}(\psi_i) \text{ for } 1 \leq i \leq n\}$.
- **In case a partial order cut is detected:**
  Let $\rho = (P, \prec)$ be the detected partial order cut, and let $A_i, A_j \in P$ be two subsets with $A_i \prec A_j$. $\Rightarrow \ \forall a_i \in A_i, a_j \in A_j \ (a_i \nrightarrow_L a_j \wedge a_j \not\nrightarrow_L a_i)$. $\Rightarrow \ \nexists \sigma \in L, 1 \leq k < l \leq |\sigma| \ (\sigma(k) \in A_j \wedge \sigma(l) \in A_i)$. $\Rightarrow \{\sigma \restriction_{A_i \cup A_j} \mid \sigma \in L\} \subseteq \{\sigma_i \cdot \sigma_j \mid \sigma_i \in {A_i}^* \wedge \sigma_j \in {A_j}^*\} = {A_i}^* \cdot {A_j}^*$.
  This shows that for any trace $\sigma \in L$, the activities within σ are ordered in such a way that the partial order $\prec$ over the subsets of

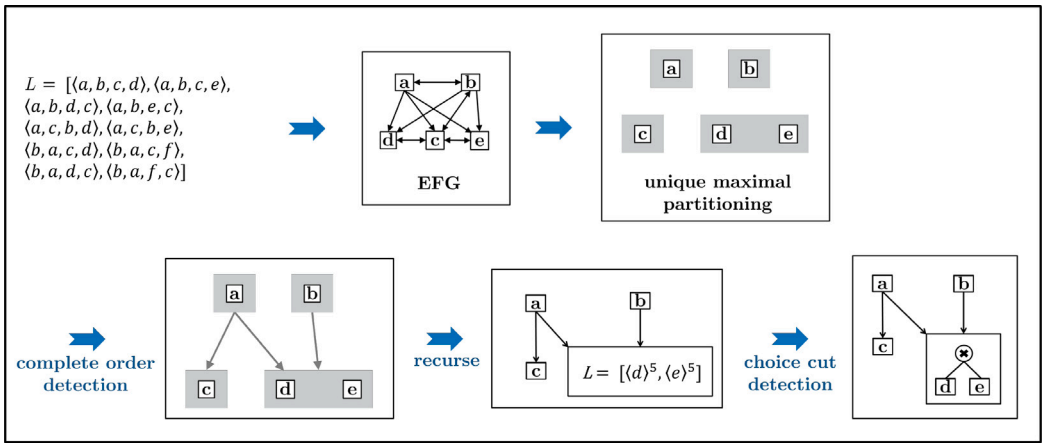


Fig. 7. Application of PM_{max} on an example event log.

the partitioning is preserved. Additionally, we know by induction that the projection of σ on each subset of activities is included in the language of the corresponding sub-model. Therefore, σ must be included in the language of $PM_{bf}(L)$.

- **In case the fall-through function is invoked:**
  All traces in L are included in the language of $PM_{bf}(L)$ since the fall-through function is fitness-preserving [2] and we proved that transforming sequence and concurrency cuts into partial orders yields the same language.

Thus, by induction, $PM_{bf}(L)$ can regenerate every trace in the event log L. □

6.2. POWL inductive miner - maximal order (PM_{max})

In [4], we refined the discovery of POWL models by introducing the concept of maximality for partial order cuts. As outlined in Fig. 5(b), PM_{max} mines for a unique partial order cut instead of validating partial orders over all possible partitionings of activities (cf. Fig. 5(a)). This approach enhances the efficiency of the partial order detection step by mining for a unique solution and it ensures that this solution provides existence and validity guarantees if any other solutions exist.

6.2.1. Completeness and maximality of partial orders

We define notions of *completeness* and *maximality* of partial order cuts. A partial order cut is complete if it captures all order restrictions derived from the eventually-follows graph. For instance, the valid cut ρ_2 in Fig. 6 is not complete because it does not capture the sequential dependency between activities a and b, while ρ_3 and ρ_4 are both complete.

Definition 11 (*Complete Partial Order Cut [4]*)**.** Let $\rho = (P, \prec)$ be a partial order cut of an event log $L \in \mathcal{M}(\Sigma^*)$. ρ is complete if for all $A_i, A_j \in P; a_i \in A_i; a_j \in A_j$: $A_i \prec A_j \Leftrightarrow (a_i \rightsquigarrow_L a_j \wedge a_j \not\rightsquigarrow_L a_i)$.

After establishing completeness, we further define maximality for complete cuts. We define a complete partial order cut to be *maximal* if no subsets of the partitioning can be merged without violating completeness. In other words, two activities are in the same subset of the partitioning if and only if they share the same sets of preceding and succeeding activities. In our example in Fig. 6, the third partial order cut ρ_3 is not maximal because $\{b\}$ and $\{c\}$ are both preceded by $\{a\}$ and succeeded by $\{d\}$. The activities b and c can be merged into the same group without violating completeness, resulting in the maximal partial order cut ρ_4.

Notation. For a binary relation $\prec \subseteq X \times X$ over a set X, we define the *pre-set* of $x \in X$ as $\bullet\prec x = \{y \in X \mid y \prec x \wedge x \not\prec y\}$ and the *post-set* of x as $x \prec \bullet = \{y \in X \mid x \prec y \wedge y \not\prec x\}$.

Definition 12 (*Maximal Partial Order Cut [4]*)**.** Let $\rho = (P, \prec)$ be a complete partial order cut of an event log $L \in \mathcal{M}(\Sigma^*)$. ρ is maximal if for all $A_i, A_j \in P$; $a_i \in A_i$; $a_j \in A_j$: $(A_i = A_j) \Leftrightarrow (\bullet\rightsquigarrow_L a_i = \bullet\rightsquigarrow_L a_j \wedge a_i \rightsquigarrow_L \bullet = a_j \rightsquigarrow_L \bullet)$.

6.2.2. Approach

The maximal POWL inductive miner (PM_{max}) follows the same steps outlined in Alg. 1, but it restricts the search space to mine for a maximal partial order cut. Instead of generating and validating partial order cuts over all possible partitionings of the activities (cf. lines 16), PM_{max} utilizes Definition 12 to uniquely define a partitioning of the activities. It only mines for a valid partial order cut over this partitioning. Since PM_{max} follows the same steps as PM_{bf}, it preserves the fitness guarantee.

Theorem 4 (*Fitness Guarantee of PM_{max}*)**.** *Let $L \in \mathcal{M}(\Sigma^*)$ be an event log and let $PM_{max}(L)$ be the POWL model discovered using the maximal POWL inductive miner. Every trace $\sigma \in L$ is included in the model's language, i.e., $\sigma \in \mathcal{L}(PM_{max}(L))$.*

Proof. The proof is analogous to Theorem 3. □

Besides ensuring the efficiency of the algorithm, mining for a maximal partial order cut has several advantages over the brute-force approach. First, grouping nodes increases the simplicity of the graph representation of a partial order (fewer arcs are required). Moreover, it enables the discovery of more structures in the next iterations of the algorithm. In our example in Fig. 6, by grouping b and c together, we enable the discovery of a loop cut between these two activities in the next iteration. Finally, we proved in [4] that mining for a maximal cut provides the following three guarantees:

1. **Uniqueness Guarantee:** if a maximal partial order cut exists, then it is unique.
2. **Existence Guarantee:** if a complete partial order cut exists, then the maximal partial order cut exists as well.
3. **Validity Guarantee:** if a complete partial order cut exists and is valid, then the maximal partial order cut exists and is valid as well.

Fig. 7 illustrates the application of PM_{max} on an example event log. In the first iteration, the algorithm mines for a maximal partial order cut after failing to detect a base case or a process tree. It generates the

unique partitioning of the activities as defined in Definition 12. The activities d and e are grouped into the same subset because they share the same pre-set and post-set with respect to the eventually-follows graph. The algorithm detects a complete partial order cut over this partitioning, which also conforms with the validity requirements of Definition 11. The discovered maximal partial order cut is returned, and the algorithm continues the recursion to discover a choice cut between d and e in the next iteration.

6.3. POWL Inductive Miner - Dynamic Clustering (PM_{clst})

In Section 6.2, we introduced PM_{max} as an extension to PM_{bf} that aims to improve the efficiency of the algorithm. It avoids generating and validating partial orders over all possible partitionings of activities by restricting the search space to only mine for complete partial order cuts. In this section, we introduce an alternative approach (PM_{clst}) that also prioritizes efficiency. Rather than restricting the search space, PM_{clst} allows for more flexibility in the discovery of partial orders. It discovers partial orders that do not fully meet the strong validity requirements we defined in Definition 10. The efficiency of this approach is ensured by the dynamic strategy employed to generate partial orders by clustering subsets of activities. The new approach additionally aims to reduce reliance on the fall-through function by accepting a broader range of partial orders.

The step of partial order cut detection in PM_{clst} is illustrated in Fig. 5(c). It starts by partitioning the activities into singletons (i.e., into subsets of size 1) and building a binary relation over this partitioning (called *order graph*) to capture the order restrictions between activities. Then, the generated order graph undergoes a series of validation and refinement steps by applying a sequence of *clustering functions*. These clustering functions dynamically merge subsets of activities that violate transitivity (c_{intr}), violate irreflexivity (c_{refl}), are mutually exclusive (c_{xor}), or share the same preceding and succeeding nodes (c_{grp}).

6.3.1. Order graph

For every partitioning of activities P, we uniquely define an *order graph* ($\twoheadrightarrow_P$) that captures the order restrictions between activities in the eventually-follows graphs such that $A_1 \twoheadrightarrow_P A_2$ if at least one activity in A_1 is eventually followed by an activity from A_2, and none of the activities of A_2 is eventually followed by an activity from A_1. Additionally, we extend the graph by the missing transitive relationships that do not conflict with the eventually-follows graph. In other words, if there is an indirect path in the initial order graph from A_1 to A_2 and A_2 is never eventually followed by A_1, then we add (A_1, A_2) to the order graph.

Definition 13 (*Order Graph*)**.** Let $L \in \mathcal{M}(\Sigma^*)$ be an event log and $P \in Part(\Sigma_L)$ be a partitioning of the activities of L. The order graph of P is a binary relation $\twoheadrightarrow_P \subseteq P \times P$ uniquely defined as follows:

- $\twoheadrightarrow_{pre} = \{(A_1, A_2) \in (P \times P) \mid \rightsquigarrow_L \cap (A_1 \times A_2) \neq \emptyset \wedge \rightsquigarrow_L \cap (A_2 \times A_1) = \emptyset\}$.
- $\twoheadrightarrow_P = \{(A_1, A_2) \in \twoheadrightarrow_{pre}{}^+ \mid \rightsquigarrow_L \cap (A_2 \times A_1) = \emptyset\}$.

We use $OG_L = \{\twoheadrightarrow_P \mid P \in Part(\Sigma_L)\}$ to denote the set of all order graphs of an event log L. The order graph is not necessarily a partial order. Fig. 8 shows four order graphs derived for different partitionings of the activities of the event log $L = [\langle a, b, c\rangle, \langle c, d, b\rangle]$. The green arc from a to d in $\twoheadrightarrow_1$ is added by transitivity because d is never eventually followed by a. The order graphs $\twoheadrightarrow_1$ and $\twoheadrightarrow_3$ are not partial orders because the nodes colored in red violate transitivity. We cannot resolve intransitivity in both graphs because (b, c) is in the eventually-follows graph.

We define a notion of *weak validity* for partial order cuts as a relaxed version of the strong validity definition (Definition 10). The first two conditions of Definition 10 ensure high conformance between a valid partial order and the eventually-follows graph. We define a partial order cut to be weakly valid if it fully conforms with the order graph instead. This weakens the conditions of validity to account for potential incompleteness in the event log. We also discard the third and fourth validity conditions of Definition 10. The motivation behind these two conditions is to allow for the discovery of loop cuts, i.e., if a subset of activities has no start and end activities, then it is highly likely a part of the redo part in a loop. However, since we only mine for a partial order cut after failing to discover a loop cut, discovering a partial order violating these conditions might still lead to better results than invoking the fall-through function.

Definition 14 (*Weakly Valid Order*)**.** Let $\rho = (P, \prec)$ be a partial order cut of an event log $L \in \mathcal{M}(\Sigma^*)$. ρ is weakly valid if $\prec = \twoheadrightarrow_P$.

6.3.2. Clustering functions

After building the order graph over the partitioning of activities into singletons, PM_{clst} applies a series of clustering functions. For an order graph $\twoheadrightarrow_P$ of a partitioning of activities P, a clustering function merges subsets of activities in P that meet specific requirements and returns the order graph of the refined partitioning.

Definition 15 (*Clustering Function*)**.** Let $L \in \mathcal{M}(\Sigma^*)$ be an event log. A clustering function $c : OG_L \to OG_L$ maps an order graph $\twoheadrightarrow_P \in OG_L$ of a partitioning of the activities P into the order graph $\twoheadrightarrow_{P'} \in OG_L$ of another partitioning P'. P' is generated by merging subsets from P based on a specific clustering condition, i.e., if the clustering condition is met for two subsets $A_1, A_2 \in P$, then there must exist $A \in P'$ such that $(A_1 \cup A_2) \subseteq A$.

We define the *clustering condition* C of a clustering function c as the set of pairs of activity subsets that need to be merged together, i.e., the clustering function merges $A_1, A_2 \in P$ into the same group if $(A_1, A_2) \in C$. As illustrated in Fig. 5(c), PM_{clst} uses four concrete clustering functions (c_{intr}, c_{refl}, c_{xor}, and c_{grp}), and we use C_{intr}, C_{refl}, C_{xor}, and C_{grp} to denote their corresponding clustering conditions.

The first two clustering functions aim to transform the order graph into a partial order by resolving any violations of the transitivity and irreflexitivity requirements. If the order graph contains an indirect path between two subsets of activities that are not directly connected, then c_{intr} merges these two subsets into the same group as they violate transitivity. Similarly, if the order graph directly connects two subsets of activities in both directions, then c_{refl} merges them into the same group as they violate irreflexitivity. The third clustering function (c_{xor}) enables the detection of exclusive choice structures in the next iterations (as $\times$-cuts) by merging subsets sharing no eventually-follows connections between each other. The fourth clustering function (c_{grp}) merges subsets of activities sharing the same preceding and succeeding nodes unless this grouping will lead to a single cluster. The goal of c_{grp} is to enable the detection of loop structures in the next iterations and to increase the simplicity of the partial order (grouping nodes sharing the same preceding and succeeding nodes into the same cluster decreases the number of arcs required to visualize the partial order).

Definition 16 (*Clustering Conditions*)**.** Let $L \in \mathcal{M}(\Sigma^*)$ be an event log, $P \in Part(\Sigma_L)$ be a partitioning of the activities of L, and $\twoheadrightarrow$ be the order graph of P. A clustering condition is a subset $C \subseteq (P \times P)$. We define four clustering conditions:

1. $C_{intr} = \{(A_1, A_2) \in P \times P \mid A_1 \not\twoheadrightarrow A_2 \ \wedge \ A_1 \twoheadrightarrow^+ A_2\}$.
2. $C_{refl} = \{(A_1, A_2) \in P \times P \mid A_1 \twoheadrightarrow A_2 \ \wedge \ A_2 \twoheadrightarrow A_1\}$.
3. $C_{xor} = \{(A_1, A_2) \in P \times P \mid A_1 \not\twoheadrightarrow A_2 \ \wedge \ A_2 \not\twoheadrightarrow A_1 \ \wedge \ (\rightsquigarrow_L \cap (A_1 \times A_2) = \emptyset) \ \wedge \ (\rightsquigarrow_L \cap (A_2 \times A_1) = \emptyset)\}$.
4. $C_{grp} = \{(A_1, A_2) \in P \times P \mid \bullet \twoheadrightarrow A_1 = \bullet \twoheadrightarrow A_2 \ \wedge \ A_1 \twoheadrightarrow \bullet = A_2 \twoheadrightarrow \bullet \ \wedge \ (\bullet \twoheadrightarrow A_1 \cup A_1 \twoheadrightarrow \bullet) \neq \emptyset\}$.

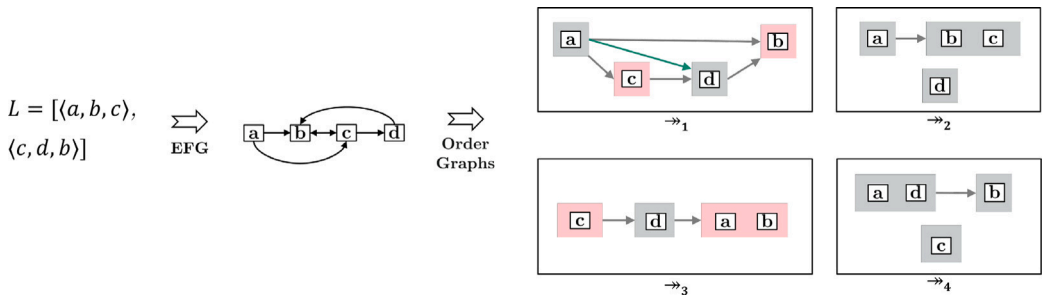


Fig. 8. An eventually-follows graph and four order graphs derived based on it for different partitionings of the activities. The green arcs are derived by transitivity. Missing transitive relationships between the red nodes are not added as they conflict with the eventually-follows graph. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

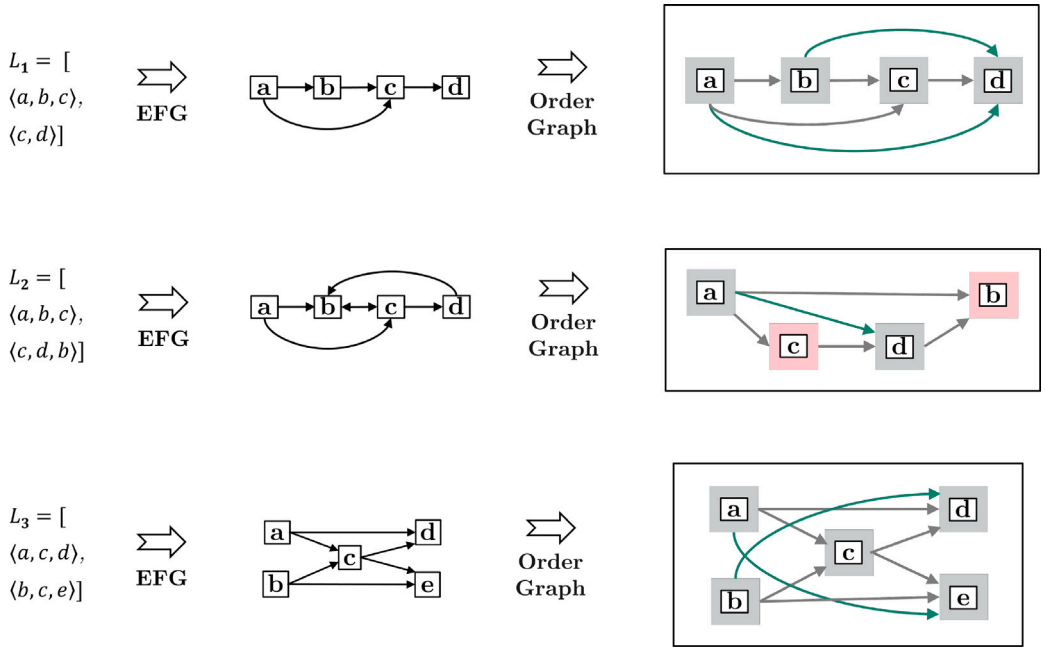


Fig. 9. Three eventually-follows graphs and the derived order graphs over the initial partitioning of activities into singletons. The green arcs are derived by transitivity. The missing transitive relationship (c, b) in the second order graph conflicts with the eventually-follows graph. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

6.3.3. Discussion: Detection of sequences and concurrency

Our previous approaches (PM_{bf} and PM_{max}) both use the process tree cuts $\rightarrow$ and $+$ to mine for sequences and concurrency, and the results are transformed into partial orders as a post-processing step. PM_{clst} moves away from the reliance on the sequence and concurrency process tree cuts because these constructs can naturally be represented as partial orders.

Without the process tree concurrency cut, PM_{max} will not be able to identify true concurrency. In a maximal partial order cut, if two activities share the same pre-set and post-set with respect to the eventually-follows graph, then they must be in the same subset within the partitioning. This prevents the detection of true concurrency, relying on detecting a concurrency cut in the next iteration. In PM_{clst}, the fourth clustering function (c_{grp}) only groups subsets of activities that share the same preceding and succeeding nodes together if this grouping does not result in a single cluster. This enables the detection of true concurrency.

Addressing the elimination of the sequence cut is more challenging. The maximal cut detection approach in PM_{max} will fail to discover sequences if the data is incomplete. This is primarily because PM_{max} relies on the eventually-follows graph, unlike the base inductive miner which uses the transitive closure of the directly-follows graph to detect sequence cuts. For example, the first eventually-follows graph shown in Fig. 9 is derived from the event log $L_1 = [\langle a, b, c\rangle, \langle c, d\rangle]$. Utilizing $\mapsto_L^+$ allows for the discovery of a process tree sequence cut $(\rightarrow, \{a\}, \{b\}, \{c\}, \{d\})$. However, PM_{max} is not able to identify an equivalent partial order cut since there is no direct eventually-follows relationships from a and b to d. Deciding whether it is more advantageous to identify such incomplete sequences or to resort to the fall-through function is not straightforward and may vary depending on specific cases. For PM_{clst}, we decided to enable the detection of incomplete sequences by extending the initial order graph with missing transitive connections (the green arcs). However, such missing transitive connections are only added to the order graph as long as adding them does not violate the perfect fitness guarantee. For example, let us consider the order graph derived for the second event log $L_2 = [\langle a, b, c\rangle, \langle c, d, b\rangle]$ in Fig. 9. The missing transitive connection (c, b) is not added to the order graph because it conflicts with the eventually-follows relationship (b, c). Adding (c, b) to the order graph will violate the fitness of the trace $\langle a, b, c\rangle$.

Another challenge in sequence detection is the modeling of long-term dependencies. Let us consider the third event log $L_3 = [\langle a, c, d\rangle, \langle b, c, e\rangle]$ in Fig. 9. The underlying process begins with a choice between a and b, followed by c, and then a non-free choice between d and e that depends on the previous choice. If a was selected before, then d is executed; if b was selected before, then e is executed. Both POWL models and process trees are not able to accurately represent such

Input: an event log L.
Output: a POWL model $PM_{clst}(L)$.
1 **begin**
2 **if** $L = [\langle a \rangle^f]$ *for* $a \in \Sigma$ *and* $f \in \mathbb{N}$ **then**
3 **return** a
4 **if** $L = []$ **then**
5 **return** τ
6 **if** *a process tree cut* $\oplus \in \{\times, \circlearrowleft\}$ *is detected* **then**
7 $L_1, ..., L_n \leftarrow$ project L on subsets
8 **return** $\oplus(PM_{clst}(L_1), ..., PM_{clst}(L_n))$
9 **else**
10 $P \leftarrow \{\{a\} \mid a \in \Sigma_L\}$
11 **for** $c \in \langle c_{intr}, c_{refl}, c_{xor}, c_{grp} \rangle$ **do**
12 $P \leftarrow c(P)$
13 **if** $|P| > 1$ **then**
14 $L_1, ..., L_n \leftarrow$ project L on the subsets of $P = \{A_1, ..., A_n\}$
15 $\psi_1, ..., \psi_n \leftarrow PM_{clst}(L_1), ..., PM_{clst}(L_n)$
16 $\prec' \leftarrow \{(\psi_i, \psi_j) \mid i, j \in \{1, ..., n\} \wedge A_i \twoheadrightarrow_P A_j\}$
17 **return** $(\{\psi_1, ..., \psi_n\}, \prec')$
18 $\psi \leftarrow$ FallThrough(L)
19 **return** ψ

Algorithm 2: POWL Inductive Miner - Dynamic Clustering (PM_{clst}). The algorithm takes an event log as input and returns a POWL model. The partial order cut detection step involves applying the clustering functions to dynamically cluster subsets of activities.

long-term dependencies. The optimal process tree for this scenario is $\rightarrow(\times(a, b), c, \times(d, e))$, but this sequence cannot be deduced solely from the eventually-follows graph, as a is never eventually followed by e. Extending the order graph with the missing transitive relationships allows PM_{clst} to discover such sequences.

6.3.4. Approach

The dynamic clustering POWL inductive miner (PM_{clst}) is outlined in Alg. 2. It starts by mining for a base case (cf. lines 2–5). If no base case is detected, then PM_{clst} mines for a choice or loop process tree cut (cf. lines 6–8). If no choice or loop cut is detected, PM_{clst} mines for a partial order cut by dynamically clustering subsets of activities (cf. lines 10–12). It starts by partitioning the activities into singletons, then it applies all clustering functions. If the clustering step results in a single group containing all activities, then the fall-through function is invoked. Otherwise, the event log is projected into the subsets of the partitioning, and the algorithm continues the recursion and returns a poset that conforms with the order graph (cf. lines 13–17). The order graph corresponds to a partial order cut because irreflexivity and transitivity are ensured by c_{intr} and c_{refl}.

Fig. 10 illustrates an example application of PM_{clst} on the event log $L = [\langle a, b, c, d, f \rangle, \langle a, c, b, d, f \rangle, \langle a, b, c, f, d \rangle, \langle a, c, b, f, d \rangle, \langle b, c, e, f \rangle, \langle c, b, e, f \rangle, \langle b, c, f, e \rangle, \langle c, b, f, e \rangle]$. The initial order graph is extended with (a, e) by transitivity because this relationship does not conflict with the eventually-follows graph. The resulting order graph is irreflexive and transitive. The clustering functions c_{xor} and c_{grp} are then applied, merging b and c into the same group and d, e, and f into another group. A partial order modeling a sequence between the activity sets $\{a\}$, $\{b, c\}$, and $\{d, e, f\}$ is discovered. In the next iteration, c_{xor} clusters d and e into the same group because they are never eventually followed by each other. A concurrency between $\{b\}$ and $\{c\}$ and another concurrency between $\{d, e\}$ and $\{f\}$ are discovered and modeled as empty partial orders. In the final iteration, the choice between d and e is discovered and modeled as $\times(\{d\}, \{e\})$.

Relaxing the condition of validity into weak validity preserves the fitness guarantee of the step of partial order cut detection.

Theorem 5 (*Fitness Guarantee of PM_{clst}*)**.** *Let $L \in \mathcal{M}(\Sigma^*)$ be an event log and let $PM_{clst}(L)$ be the POWL model discovered using Alg. 2. Every trace $\sigma \in L$ is included in the model's language, i.e., $\sigma \in \mathcal{L}(PM_{clst}(L))$.*

Proof. We prove this theorem by induction.

(Base Case) If $L = [\langle a \rangle^f]$ for some $a \in \Sigma$ and $f \in \mathbb{N}$, then $PM_{clst}(L) = a$. The language of this model is $\mathcal{L}(a) = \{\langle a \rangle\}$, which includes the trace $\langle a \rangle$. If $L = []$, then no $\sigma \in L$ exists.

(Inductive Step) Assume the theorem holds for all sub-models. We distinguish three cases:

- **In case a process tree cut is detected:**
  All traces in L are included in the language of $PM_{clst}(L)$ since choice and loop cuts are fitness-preserving [2].
- **If the clustering step results in a partitioning P of size $|P| > 1$:**
  The algorithm returns a poset that corresponds to the order graph of P (cf. lines 16–17 of Alg. 2). Let $A_1, A_2 \in P$ be two subsets with $A_1 \twoheadrightarrow_P A_2$.
  $\Rightarrow \nrightarrow_L \cap (A_2 \times A_1) = \emptyset$.
  $\Rightarrow \nexists \sigma \in L, 1 \leq k < l \leq |\sigma| \ (\sigma(k) \in A_2 \wedge \sigma(l) \in A_1)$.
  $\Rightarrow \{\sigma \restriction_{A_1 \cup A_2} \mid \sigma \in L\} \subseteq \{\sigma_i \cdot \sigma_j \mid \sigma_i \in {A_1}^* \wedge \sigma_j \in {A_2}^*\} = {A_1}^* \cdot {A_2}^*$.
  This shows that for any trace $\sigma \in L$, the activities within σ are ordered in such a way that the returned partial order over the sub-models is preserved. Additionally, we know by induction that the projection of σ on each subset of activities is included in the language of the corresponding sub-model. Therefore, σ must be included in the language of $PM_{clst}(L)$.
- **In case the fall-through function is invoked:**
  All traces in L are included in the language of $PM_{clst}(L)$ since the fall-through function is fitness-preserving [2] and we proved in Theorem 3 that transforming sequence and concurrency cuts into partial orders yields the same language.

Thus, by induction, $PM_{clst}(L)$ can regenerate every trace in the event log L. □

7. Incorporating frequencies in the discovery of POWL models

The three approaches we proposed for the discovery of POWL models (Section 6) all provide a perfect fitness guarantee, i.e., all traces in the event log are guaranteed to fit the generated POWL model. This guarantee is beneficial for cases where the event log is assumed to be noise-free. However, real-life data often contains noise. Moreover, ensuring a balance between fitness, precision, and simplicity is crucial in process discovery. Attempting to discover a process model that perfectly fits the event log might lead to complex models that are hard to interpret and/or imprecise models (models that allow for behavior not observed in the log).

In this section, we propose methods for incorporating frequency-based filtering in the discovery of POWL models. By integrating frequency information, we aim to refine process discovery to better match empirical data, acknowledging that some behavior patterns are more frequent and hence potentially more significant than others. We extend PM_{clst} with the following frequency-based filtering methods:

- **Eventually-Follows Frequency Filtering:** As shown in Fig. 11(a), this method extends the partial order cut detection step. It incorporates the relative frequencies of eventually-follows connections in the generation of the order graph.

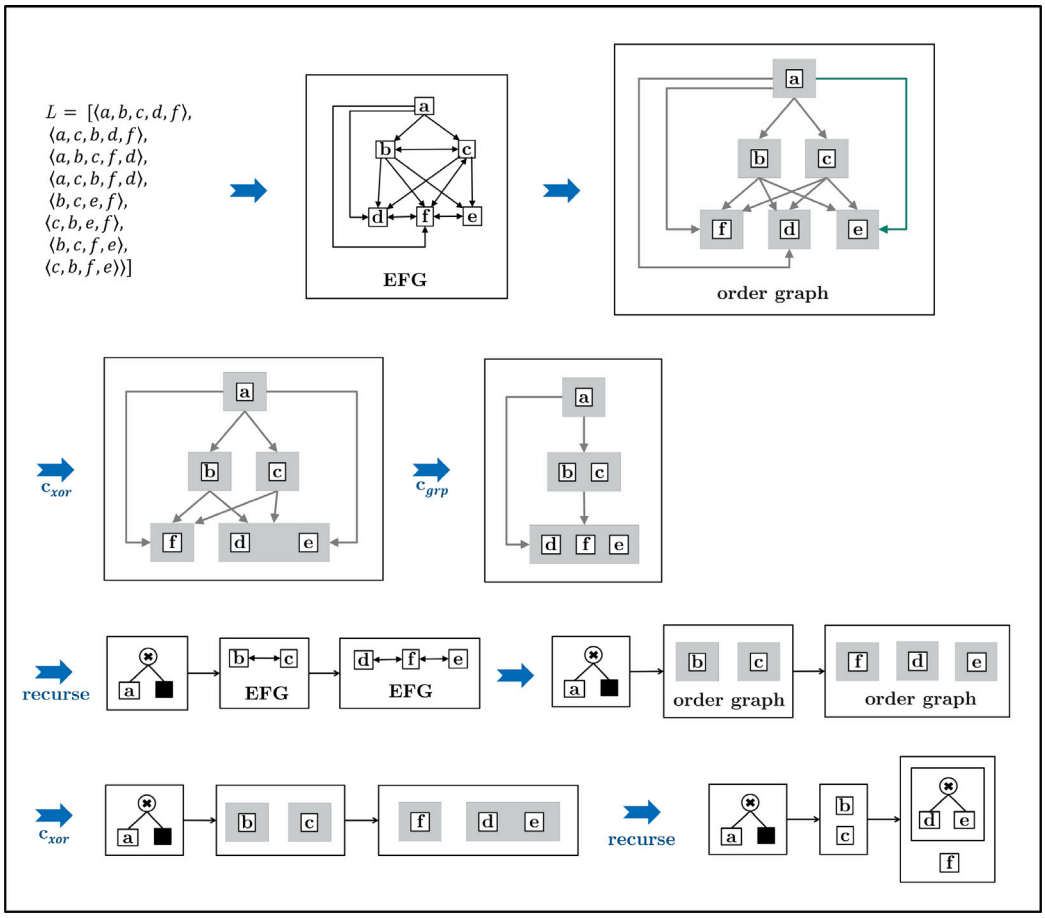


Fig. 10. Application of PM_{clst} on an example event log.

- **Trace Variant Frequency Filtering:** As illustrated in Fig. 11(b), this method applies filtering after the step of partial order cut detection and before invoking the fall-through function. It filters the most frequent trace variants using a dynamic weight factor for adjusting the frequency threshold.

The integration of frequency information into process discovery can be particularly useful in contexts where less frequent behavior is considered less impactful. However, we acknowledge that this approach may not be suitable in scenarios where rare behaviors are crucial to the process.

7.1. Incorporating eventually-follows frequencies in the generation of the order graph

The first filtering method we propose extends the dynamic clustering partial order cut detection step of PM_{clst}. As shown in Fig. 11(a), we adapt the order graph to incorporate the relative frequencies of eventually-follows connections instead of just capturing their existence. This extension allows us to evaluate and filter connections in the eventually-follows graph based on the observed frequencies, facilitating the identification of significant patterns that are more likely to be representative of the underlying process.

We introduce the concept of *eventually-follows frequency*. For an event log L and two activities a and b within L, the frequency score $\#_L(a, b)$ is defined as the number of traces in L where a is eventually followed by b, providing a quantifiable measure of the connection strength between two activities. We extend this notion to sets of activities. For two sets of activities A and B, the frequency score $\#_L(A, B)$ is defined as the number of traces in L where at least one activity from A is eventually followed by at least one activity from B.

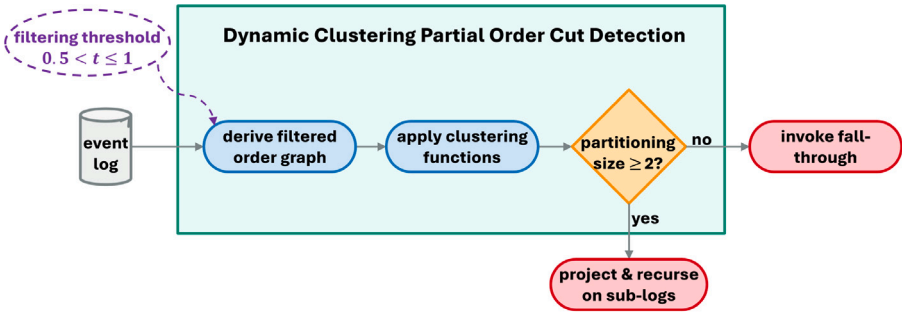
Definition 17 (*Eventually-Follows Frequency*)**.** Let $L \in \mathcal{M}(\Sigma^*)$ be an event log and $A, B \subseteq \Sigma_L$ be two subsets of activities. The eventually-follows frequency of (A, B) in L is defined as:

$$\#_L(A, B) = \sum_{\sigma \in L} \begin{cases} L(\sigma) & \text{if } \exists 1 \leq i < j \leq |\sigma| \ (\sigma(i) \in A \ \wedge \ \sigma(j) \in B), \\ 0 & \text{otherwise.} \end{cases}$$

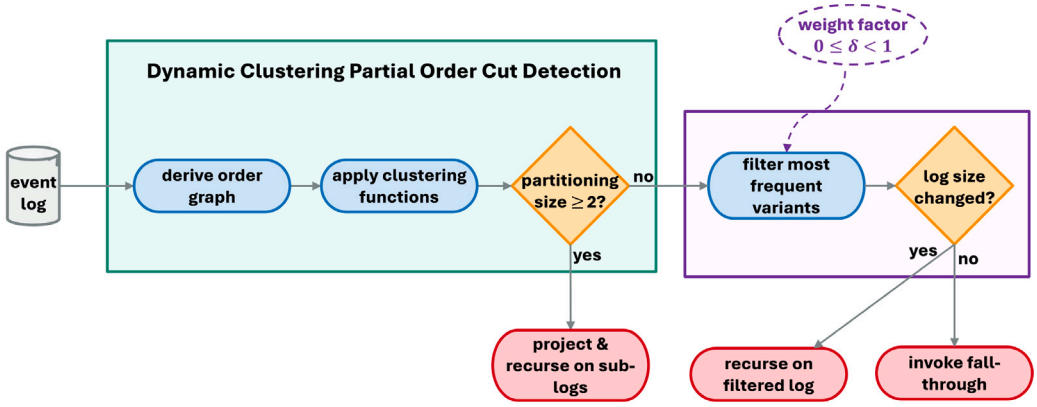
The order graph (Definition 13) is constructed based on the existence of the eventually-follows relationships between activities. If an activity from A_1 is eventually followed by an activity from A_2 in any trace within the event log, and there are no traces where activities from A_2 are followed by activities from A_1, then a directed edge from A_1 to A_2 is included in the order graph. This approach provides a binary view: a connection is either present or not, irrespective of how often it occurs within the event log.

We extend the order graph with a filtering threshold t that serves to quantify the significance of eventually-follows relationships. A higher value of t indicates a stricter requirement for the frequency of one subset of activities following another. This approach shares similarities with filtering methods used in other process mining techniques. For instance, the heuristics miner [22] generates an intermediate graph representing dependencies between activities, and it uses a threshold to filter these dependencies based on a frequency metric.

In the filtered order graph, we add a directed edge from A_1 to A_2 if the relative frequency of A_1 being eventually followed by A_2, compared to A_2 being eventually followed by A_1, meets or exceeds the given threshold t. The initial order graph is then extended with transitive relationships that are missing due to incompleteness in the data (i.e., between groups that are never eventually followed by each other). The threshold t has values higher than 0.5 and up to 1. Values lower than 0.5 are not supported because they indicate that A_2 is



(a) Filtering based on eventually-follows frequencies.



(b) Filtering based on trace variant frequencies.

Fig. 11. Incorporating frequency-based filtering in/after the partial order cut detection step of PM_{clst}.

more frequently followed by A_1 than A_1 is followed by A_2, thereby contradicting the intended direction of the edge from A_1 to A_2. Adding such edges conflicts with our criteria for establishing directed edges based on the more frequent flow patterns in the data.

Definition 18 (*Filtered Order Graph*)**.** Let $L \in \mathcal{M}(\Sigma^*)$ be an event log and $P \in Part(\Sigma_L)$. The order graph of P with respect to a filtering threshold $0.5 < t \leq 1$ is a binary relation $\twoheadrightarrow_P \subseteq P \times P$ uniquely defined as follows:

- $\prec = \{(A_1, A_2) \in (P \times P) \mid \#_L(A_1, A_2) + \#_L(A_2, A_1) > 0 \wedge \frac{\#_L(A_1,A_2)}{\#_L(A_1,A_2)+\#_L(A_2,A_1)} \geq t\}$.
- $\twoheadrightarrow_P = \prec \cup \{(A_1, A_2) \in \prec^+ \mid \#_L(A_1, A_2) + \#_L(A_2, A_1) = 0\}$.

When setting t to 1, an edge from A_1 to A_2 is established only if A_2 is never eventually followed by A_1 within the event log. This aligns with the original definition of the order graph, which accounts purely for the existence of eventually-follows relationships.

7.2. Dynamic trace variant frequency filtering

Real-life event logs often follow a Pareto distribution of traces [23], i.e., a large fraction of behavior is often covered by a small fraction of unique activity sequences (*trace variants*) that occur frequently. Therefore, we use a filtering mechanism that incorporates variant frequencies within the event log in the discovery of POWL models.

We extend PM_{clst} with a trace variant filtering step that is applied after failing to discover a partial order cut and before invoking the fall-through function. We filter the event log to keep the most frequent trace variants using a weight factor for dynamically adjusting the frequency threshold. If the filtering is successful (i.e., it results in a smaller log), we re-apply the algorithm on the filtered log; otherwise, the fall-through function is invoked. Note that the dynamic trace variant filtering method is more general than order graph filtering (Section 7.1). Although we implement the dynamic trace variant filtering method as an extension to PM_{clst}, it can be integrated into any variant of the inductive miner.

Our filtering method selects variants based on their relative frequency, ensuring that the most representative behaviors of the process are retained. We employ a dynamically adjusting frequency threshold, called the *decremental weight factor* $0 \leq \delta < 1$ [24]. Variants are sorted based on their frequencies, and we start by adding the most frequent variant. Then, the next variant is only added if its frequency is greater than the frequency of the previously added variant multiplied by the decremental weight factor. The introduction of the weight factor δ for dynamically adjusting the frequency threshold allows for a flexible and adaptive filtering process, accommodating different distributions and complexities within event logs. This design choice ensures avoiding frequency gaps between variants by comparing the frequency of each variant in question with the frequencies of the previously added ones.

Notation. Given an event log $L \in \mathcal{M}(\Sigma^*)$, a *variant* $\sigma \in L$ as a unique sequence of activities in L. The frequency $L(\sigma)$ of a variant σ is the count of its occurrences in L.

Definition 19 (*Dynamic Variant Filtering*)**.** Let $L \in \mathcal{M}(\Sigma^*)$ be an event log and $0 \leq \delta < 1$ be the decremental weight factor. Let $\langle v_1, \ldots, v_n \rangle$ denote the sorted sequence of variants in L by decreasing frequencies, i.e., $L = [{v_1}^{L(v_1)}, \ldots, {v_n}^{L(v_n)}]$ and $L(v_1) \geq \cdots \geq L(v_1)$. The log L is filtered with respect to the decremental weight factor δ into $L_\delta = [{v_i}^{L(v_i)} \mid i = 1 \vee (2 \leq i \leq n \wedge L(v_j) > \delta \times L(v_{j-1}) \text{ for all } 2 \leq j \leq i)]$.

Note that using a decremental weight factor of $\delta = 0$ disables the filtering by adding all trace variants to the filtered log.

Let us consider an example event log $L = [\langle a, b, c \rangle^3, \langle a, b, d \rangle^2, \langle a, c, d \rangle^1, \langle b, c, d \rangle^1]$. This log consists of four variants sorted by their frequencies. Let us apply dynamic variant filtering with a weight factor of $\delta = 0.5$. The first variant is always included in the filtered log, in our case $\langle a, b, c \rangle$ with a frequency of 3. The second variant $\langle a, b, d \rangle$ with

a frequency of 2 is also included in the filtered log because $2 > 0.5 \times 3$. Next, we examine $\langle a, c, d \rangle$ with a frequency of 1. Since $1 \not> 0.5 \times 2$, all remaining variants are filtered out. Therefore, applying dynamic variant filtering leads to $L_{0.5} = [\langle a, b, c \rangle^3, \langle a, b, d \rangle^2]$.

8. Visualization of POWL models

To provide alternative ways of visualizing POWL models, we propose several variations that cater to different user needs and contexts. We aim to retain the underlying structure of the POWL model while introducing visual elements that could make the model more intuitive and easier to analyze in certain use cases. It is important to note that these visualizations are intended to offer alternatives rather than definitive improvements. Different users may find different visualizations more or less effective, depending on their background and the specific requirements of their tasks. Future work could explore the empirical impact of these alternatives on user performance and satisfaction.

We have implemented three concepts that provide alternative approaches to the visualization of POWL models: reduction rules, frequency tags, and decision gates.

Reduction rules

Similar to process tree reduction rules [2], we define several rules to reduce the complexity of POWL models without changing their semantics. We identify and eliminate unnecessary nesting, flatten hierarchies, and streamline sequences. The result is a simplified model that retains the original behavior and logical structure. This visualization is particularly useful for users who prioritize a cleaner, more concise view of the process model, such as analysts working on large-scale models where minimizing visual clutter is essential. This approach aligns with the study in [25] showing that flattening unnecessary hierarchies in business process models can significantly improve their understandability.

We integrate standard process tree reduction rules [2] for the operators $\times$ and $\circlearrowleft$. For example, we use a simple loop $\circlearrowleft(\tau, \psi)$ to replace both structures $\times(\tau, \circlearrowleft(\tau, \psi))$ and $\times(\tau, \circlearrowleft(\psi, \tau))$. Additionally, we employ the two additional reduction rules illustrated in Fig. 12. The idea behind these reduction rules is to eliminate unnecessary nestings of partial orders as long as this reduction does not increase the number of arcs in the visualization. The following list details the two rules for eliminating unnecessary nestings of partial orders:

- The first rule targets scenarios where a partial order ρ_1 is a child of another partial order ρ_2 and is concurrent with the other children of ρ_2. In this case, the children of ρ_1 are relocated to become direct children of ρ_2. This adjustment reduces the depth of the process model, minimizing unnecessary hierarchical complexity while preserving the concurrency relationships inherent in the model.
- The second rule applies when a partial order ρ_1 is a child of another partial order ρ_2 and ρ_1 has exactly one unique start node and one unique end node. In this scenario, the children of ρ_1 are shifted to become direct children of ρ_2 and connected to the children of ρ_2, taking ρ_1's original position in the ordering. Having unique start and end nodes is critical because this ensures that eliminating the nesting does not increase the number of arcs in the visualization.

Frequency tags

The inductive miner tends to produce models with skippable activities and self-loops. A skippable activity is represented in a process tree or a POWL model as a choice $\times(a, \tau)$, while a self-loop is represented either as $\circlearrowleft(a, \tau)$ if it is not skippable or as $\circlearrowleft(\tau, a)$ if it is skippable.

In order to simplify representing such trivial structures, we incorporate *frequency tags* into the visualization of activities in POWL models. We use the skip forward symbol to indicate that an activity is skippable and the repeat symbol to indicate that an activity can be repeated in a self-loop. Combining both frequency tags indicates that the self-loop can be skipped. This visualization may be especially beneficial for users familiar with the concept of skippable activities and self-loops, such as experienced process miners.

The integration of frequency tags aims to streamline visual complexity by replacing frequent sub-models with intuitive symbols. This approach aligns with the study in [26] that shows that modularity through the introduction of sub-processes improves the understandability of complex process models. However, it is important to recognize that adding new notational elements, such as frequency tags, can also introduce additional cognitive load. To mitigate this, we have limited the use of frequency tags to just two symbols that capture frequent patterns.

Fig. 13 shows the visualization of the POWL model discovered by applying PM_{clst} on the Sepsis Cases event log [27] before and after applying the reduction rules and incorporating frequency tags. In this example, 10 out of the 16 activities are marked with frequency tags to replace choice and loop blocks. This example shows the large impact of frequency tags on reducing the complexity of models generated by the inductive miner.

Decision gates

Business Process Model and Notation (BPMN) is a standard for business process modeling widely utilized and understood in the field of process management. We define an alternative visualization that brings POWL closer to the BPMN standard in order to facilitate the adoption of POWL models in real-world scenarios and help bridge the gap between academic process modeling and industry standards. The new visualization aims to provide a more intuitive representation of POWL models, especially for stakeholders who are familiar with the BPMN standard.

This visualization aligns with cognitive theories on familiarity, which suggest that people tend to make hypotheses about new information based on the similarity to what is already stored in memory [28]. By employing elements similar to those used in BPMN, we aim to leverage users' familiarity with BPMN, reducing the cognitive load required to interpret POWL models.

In BPMN, choice and loop constructs are modeled using exclusive choice gates, which are represented using diamond-shaped icons with an "X" inside. Similarly, we define decision gateways to model choice and loop blocks in POWL models. A *split decision gate*, visualized as a green diamond with an "X" inside it, is a decision point that has one incoming arc and more than one outgoing arcs indicating that the process can take only one path out of many available paths (i.e., start of a choice block). A *join decision gate*, visualized as an orange diamond with an "X" inside it, is a decision point that has one outgoing arc and more than one incoming arcs indicating that several alternative paths converge into one (i.e., end of a choice block). Loops can also be represented by decision gates: a split decision gate indicating a choice after the do-part between terminating the loop or entering the redo-part; and a join decision gate before entering the do-part. In addition to the decision gates, we utilize specific symbols to mark the start of the process (play symbol) and the end of the process (stop symbol).

Fig. 14 shows the visualization of the POWL model from Fig. 13 after incorporating decision gates. Silent transitions are utilized differently in the two visualizations. On the one hand, some silent transitions are eliminated in Fig. 14 by directly connecting decision gates, leading to a cleaner representation of skippable parts. On the other hand, additional silent transitions were added to model the start and end of partially ordered submodels.

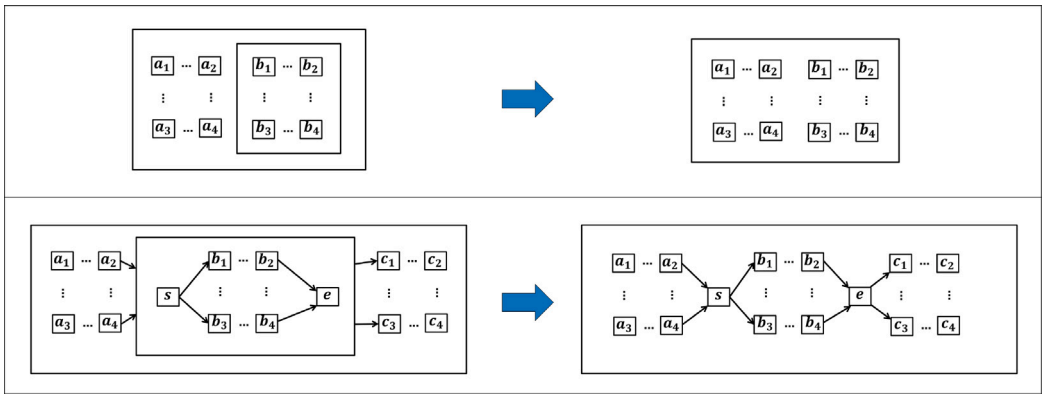
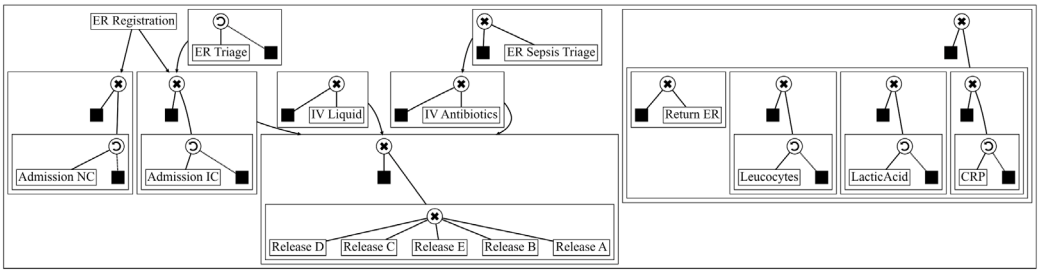
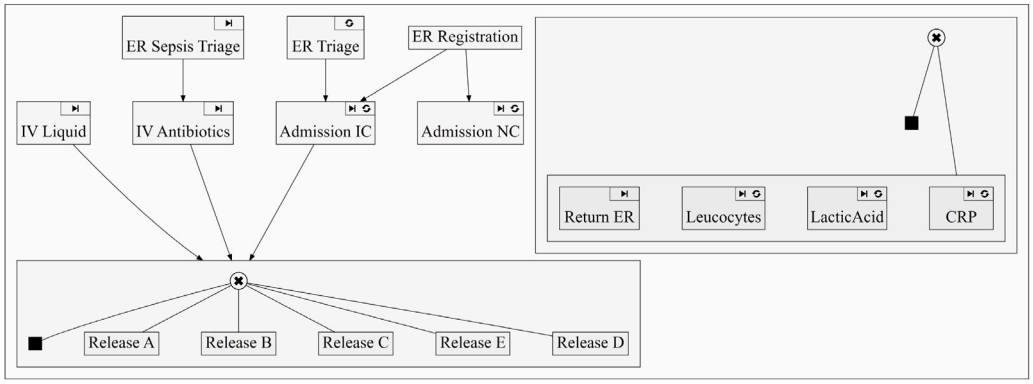


Fig. 12. Reduction rules to eliminate unnecessary nestings of partial orders.



(a) Old visualization before applying the reduction rules and incorporating frequency tags.



(b) New visualization after applying the reduction rules and incorporating frequency tags.

Fig. 13. Visualizations of a model discovered for the Sepsis Cases event log.

9. Evaluation

In this section, we evaluate the different approaches we presented in Section 6 for the discovery of POWL models. The approaches are evaluated both in terms of the time performance, as well as the quality of the discovered models. Moreover, we investigate the impact using different thresholds for the two filtering techniques suggested in Section 7.

9.1. Implementation

We implemented all approaches and methods presented in this paper as an extension to the PM4Py library [24]. The code is available under https://github.com/fit-humam-kourani/POWL_ISJ.

Listing 1: shows the code required to apply PM_{clst} on an input event log with an order graph filtering threshold of 0.8 (lines 8–10). Moreover, it shows how to visualize the generated model using the two versions of the visualization we proposed in Section 8: a version that includes the reduction rules and frequency tags (cf. Fig. 13(b)) and another version that additionally uses decision gates (cf. Fig. 14). Finally, the generated model is exported as a workflow net (lines 19–20) and as a BPMN model (lines 23–24).

9.2. Setup

In order to allow for a fair comparison between the generated models, we do not use evaluation metrics that are specific to either process trees or POWL models directly. Instead, we transform both the resulting process trees and POWL models into workflow nets, and we evaluate the quality of these workflow nets. This approach aligns with the fact that both process trees and POWL models can serve as intermediate representations in process discovery due to their soundness guarantee, while providing end users with final models in standard notations they are more familiar with.

We compute the three conformance-checking metrics implemented in PM4Py: fitness [29], precision [30], and simplicity [31]. Fitness quantifies the reproducibility of the behavior recorded in the event log. Precision reflects how well the model is restricted to the behavior recorded in the event log. In PM4Py, the simplicity metric assesses the

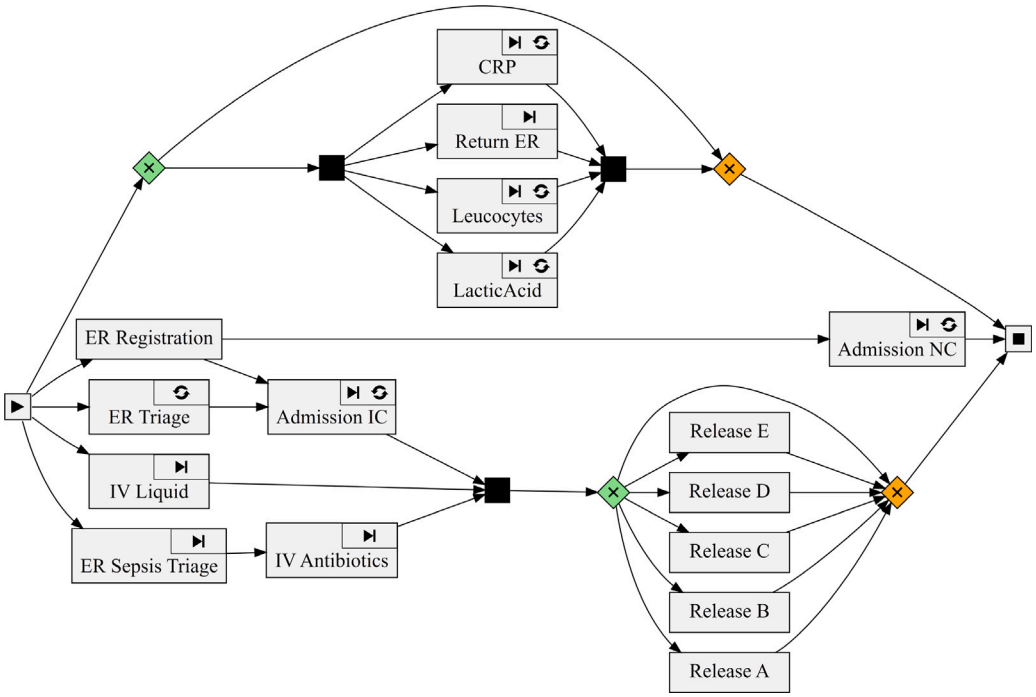


Fig. 14. Incorporating decision gates into the visualization of the POWL model from Fig. 13(b).

Listing 1: Python code for the discovery of a POWL model from an event log, visualizing it, and exporting it as Workflow net and BPMN model.

```
import pm4py
from pm4py.algo.discovery.powl.inductive.variants.powl_discovery_varaints import POWLDiscoveryVariant as
    Variants


log = pm4py.read_xes("PATH/TO/LOG")

# discovery
powl_model = pm4py.discover_powl(log,
                                 variant=Variants.DYNAMIC_CLUSTERING,
                                 order_graph_filtering_threshold=0.8)

# visualization with frequency tags
pm4py.view_powl(powl_model)

# visualization with frequency tags and decision gates
pm4py.view_powl_net(powl_model)

# export as Workflow net
petri_net, i_marking, f_marking = pm4py.convert_to_petri_net(powl_model)
pm4py.write_pnml(petri_net, i_marking, f_marking, "PATH/TO/NET")

# export as BPMN
bpmn_model = pm4py.convert_to_bpmn(powl_model)
pm4py.write_bpmn(bpmn_model, "PATH/TO/BPMN")
```

model's simplicity based on its average number of arcs per place or transition.

We use several real-life event logs for the evaluation: Sepsis Cases [27], Fine Management [32], Hospital Billing [33], BPI Challenge 2017 [34], BPI Challenge 2018 [35], BPI Challenge 2019 [36], and the five event logs of the BPI Challenge 2020 [37] (Request For Payment, Prepaid Travel Costs, Travel Permit Data, International Declarations, and Domestic Declarations).

We define the following research questions for our evaluation:

RQ1 How does the POWL inductive miner behave compared to other state-of-the-art discovery algorithms?

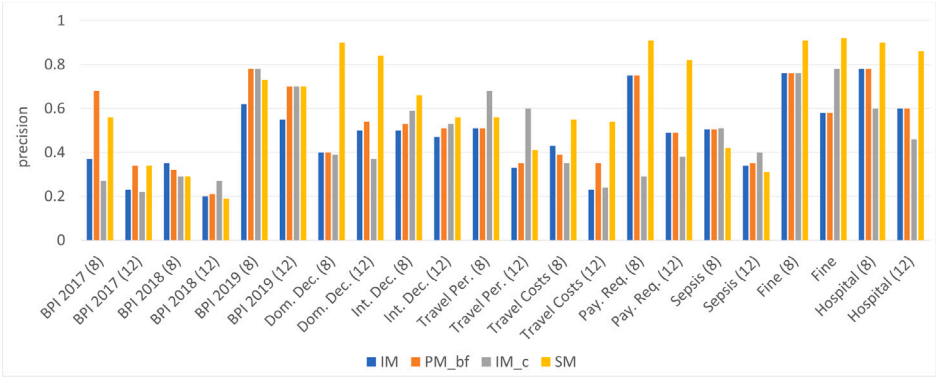
RQ2 Which approaches within the POWL inductive mining framework are scalable enough to deal with large event logs?

RQ3 What are the differences in the quality of the models discovered by the different POWL inductive mining approaches?

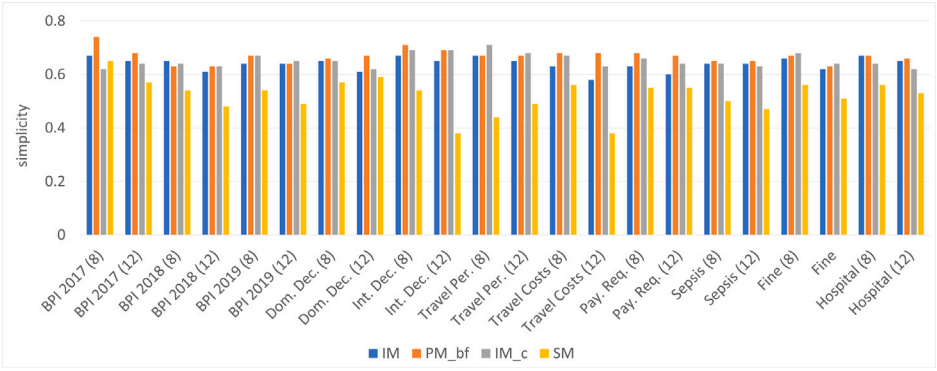
RQ4 What is the impact of the two filtering methods we proposed in Section 7 on the quality of the discovered models?

9.3. Comparison with State-Of-The-Art approaches

In this section, we investigate the research question RQ1 by comparing the brute-force POWL inductive miner (PM_{bf}) with state-of-the-art process discovery algorithms. We use the base inductive miner (IM) since the POWL inductive miner is an extension to IM, and we also consider an approach that adapts the inductive miner to handle incompleteness in the event log (IM_c) [2]. We additionally apply another



(a) Precision.



(b) Simplicity.

Fig. 15. Comparing the quality of the models discovered by the brute-force POWL inductive miner with the base inductive miner, the extension of the inductive miner that handles incompleteness, and the split miner. The x-axis is labeled with the names of event logs accompanied by the number of activities considered in each, e.g., "BPI 2017 (8)" refers to the projection of the BPI Challenge 2017 event log onto its eight most frequent activities.

state-of-the-art discovery approach: the Split Miner (SM) [38] with the filtering threshold set to 0 and default values for other parameters.

Since PM_{bf} uses a brute-force approach for generating candidate partial order over all possible partitions of activities, we preprocess all event logs in this section creating two variants of each log containing the most frequent 8 and 12 activities respectively. Note that the event logs were filtered based on the trace frequencies of the activities (i.e., the number of traces in which each activity occurs).

In Fig. 15, we report the precision and simplicity values for all discovered models. We omit fitness because both IM and PM_{bf} led to a fitness of 1.0 for all models due to the fitness guarantee they provide.

Overall, the three inductive mining approaches (IM, IM_c, and PM_{bf}) lead to models of greater simplicity compared to the split miner. The brute-force POWL inductive miner (PM_{bf}) yields the simplest models, averaging a simplicity score of 0.67, compared to an average simplicity of 0.52 for the split miner. It is important to note that these scores reflect only the simplicity of the models after converting them into WF-nets, not the simplicity of the different types of modeling languages used.

Precision levels show variability across different event logs. In some cases, the split miner outperforms the inductive miner in precision but leads to lower simplicity (e.g., Fine Management). In other cases, the inductive miner excels in both precision and simplicity (e.g., Sepsis Cases).

By comparing the inductive mining approaches, we observe that on average, PM_{bf} yields the highest precision. In general, PM_{bf} results in more precise models than IM. This precision improvement in certain instances is due to the handling of incompleteness in PM_{bf}. By employing the eventually-follows graph over the directly-follows graph to detect partial order cuts, the POWL inductive miner is able to handle incompleteness in certain cases. For example, for the BPI Challenge 2019 log with 12 activities, precision improves from 0.55 to 0.7. Nevertheless, the POWL model discovered by PM_{bf} for this log does not include structures beyond what a process tree can capture, essentially mirroring the process tree behavior discovered by IM_c.

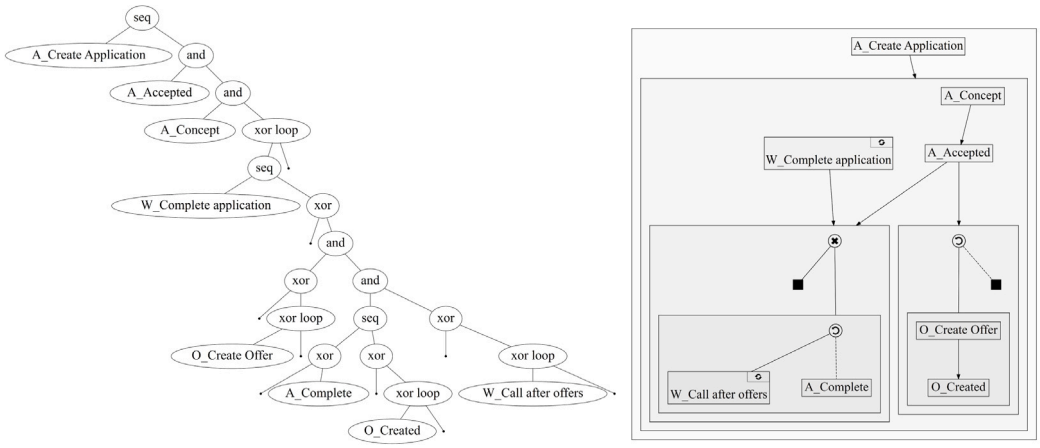
The models discovered by IM and PM_{bf} for the BPI Challenge 2017 log with 8 activities are shown in Fig. 16. This POWL model reaches a precision of 0.68, surpassing the 0.37 precision of the process tree. PM_{bf} is able to identify local activity dependencies that IM fails to discover. For example, PM_{bf} identifies a sequential relationship between "A_Concept" and "A_Accepted". IM overlooks this local sequential dependency because it does not correspond to a global sequence process tree cut that encompasses all activities. The POWL model contains a partial order over the activity sets {A_Concept}, {A_Accepted}, {O_Create Offer, O_Created}, {W_Complete application}, and {W_Call after offers, A_Complete}. Since process trees are not able to model this partial order, IM invokes the fall-through function instead, which returns a concurrency cut between {A_Accepted} and the rest of the activities.

While PM_{bf} generally results in more precise models than IM, there are a few exceptions. An example of such cases is BPI Challenge 2018 with 8 activities, where precision drops from 0.35 to 0.32. In this example, employing the fall-through function yields a more precise result than identifying a partial order. To continue the recursion, the fall-through function returns a concurrency cut between an activity appearing at most once in every trace and the remaining activities.

9.4. Scalability

In this section, we investigate the research question RQ2. We apply our three approaches for the discovery of POWL models on real-life event logs.

We filter the event logs to create three variants of each log: a projection on the most frequent 8 activities, a projection on the most



(a) Process tree (IM). (b) POWL model (PM_{bf}).

Fig. 16. Models discovered for the BPI Challenge 2017 event log with 8 activities.

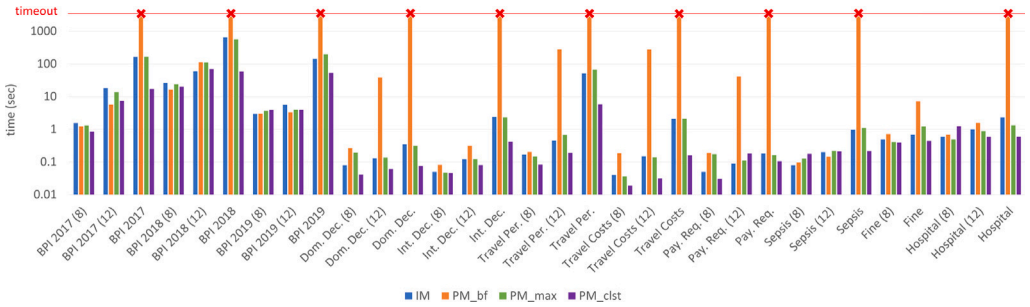


Fig. 17. Comparing the time performance of the three POWL inductive mining approaches with the base inductive miner.

frequent 12 activities, and the complete unfiltered log. We compute the time required to discover each model, and we use a timeout of one hour.

As expected, the results in Fig. 17 show that PM_{bf} is not able to scale well on large event logs. It results in timeouts for all unfiltered logs except the Fine Management log, which only covers eleven unique activities. By comparing PM_{bf} with IM, we observe a significant increase in time after increasing the number of activities. For example, after increasing the number of considered activities for the Travel Permit log from 8 to 12, the time increases from 0.17 to 0.45 for IM and from 0.2 s to 280.02 for PM_{bf}. This sharp increase in time when adding more activities clearly shows the inefficiency of the step of partial order cut detection using the brute-force approach.

The other two POWL inductive mining approaches (PM_{max} and PM_{clst}) are able to scale well on event logs with large numbers of activities. We observed no timeouts for these two approaches. The observed time values for PM_{max} are close to the time values required by IM, while PM_{clst} achieves better results for large event logs. For instance, PM_{clst} required 58 seconds to discover a model for the unfiltered BPI Challenge 2018, compared to 564 seconds for IM and 645 seconds for PM_{max}.

9.5. Qualitative evaluation

In this section, we compare the quality of the models discovered by the different POWL inductive mining approaches (RQ3).

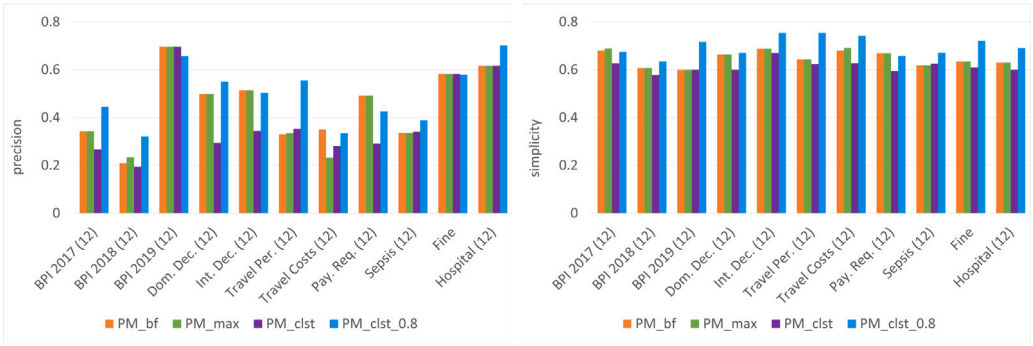
Since PM_{bf} is not scalable on event logs with a large number of activities, we consider in this section the models discovered for event logs projected on the most frequent 12 activities. For the dynamic clustering approach, we consider two settings: the base version with no filtering (PM_{clst}) and another variant using an order graph filtering threshold of 0.8 ($PM_{clst\_0.8}$).

The results of the qualitative evaluation are shown in Fig. 18. We omit fitness because $PM_{clst\_0.8}$ produced models of a trace fitness higher than 0.98, while the other three approaches ensure perfect fitness.

We observe that $PM_{clst\_0.8}$ achieves the highest simplicity score in most cases due to the applied filtering, while the other three approaches lead to models of similar levels of simplicity. However, we observe some exceptions with notable differences in simplicity between the first three approaches. For example, PM_{clst} discovers an invalid partial order for the Travel Costs log instead of invoking the fall-through function, and this increases the complexity of the model.

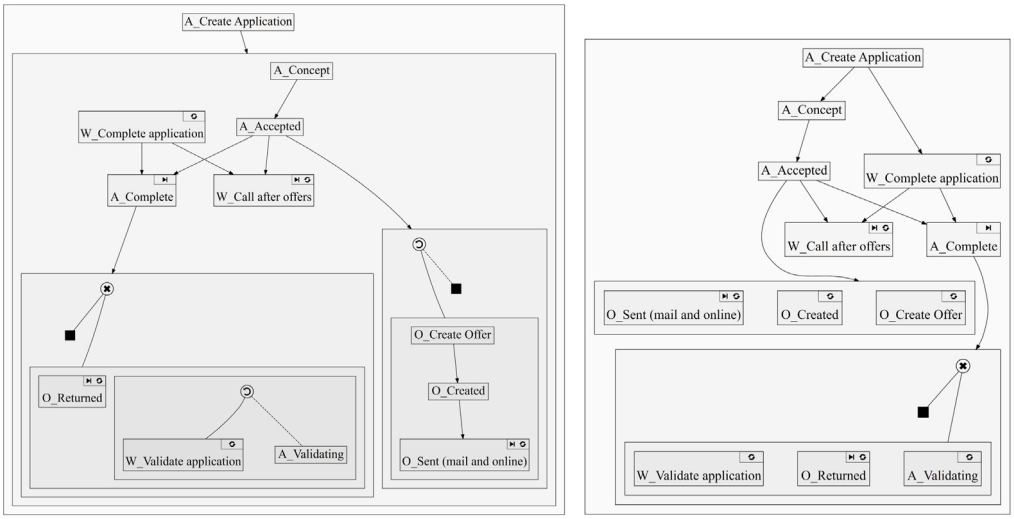
Precision varies among the different event logs. On the one hand, PM_{max} achieves higher precision than PM_{bf} for BPI Challenge 2018 due to grouping concurrent activities, enabling the identification of additional structures in following iterations. On the other hand, we observe that PM_{bf} performs better for the Travel Costs log because it discovers an incomplete partial order cut in the first iteration, while PM_{max} invokes the fall-through function instead.

In general, PM_{max} tends to produce more precise models than PM_{clst}, but applying the order graph filtering to PM_{clst} significantly increases the precision. For example, Fig. 19 shows the three models discovered for the BPI Challenge 2017 log by PM_{max}, PM_{clst}, and $PM_{clst\_0.8}$. In this example, the precision drops from 0.34 to 0.27 after enabling the detection of invalid partial orders. The fall-through function returns a loop cut over the activities “O_Create Offer”, “O_Created”, and “O_Send (mail and online)”, which is more precise than the empty partial order created over these activities by PM_{clst}. After enabling order graph filtering, the precision increases to 0.45 as the new model contains more order restrictions (e.g., {“O_Create Offer”, “O_Created”} $\prec$ {“O_Send (mail and online)”}.

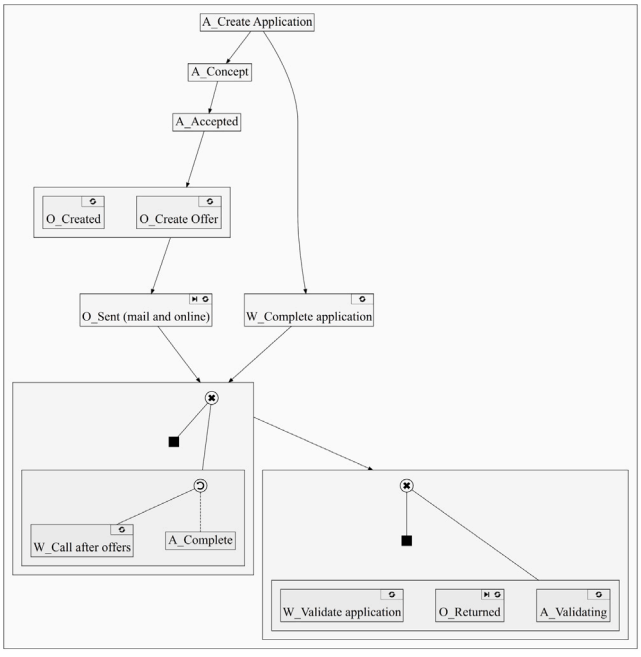


(a) Precision. (b) Simplicity.

Fig. 18. Comparing the quality of the models discovered by the different POWL inductive mining approaches.

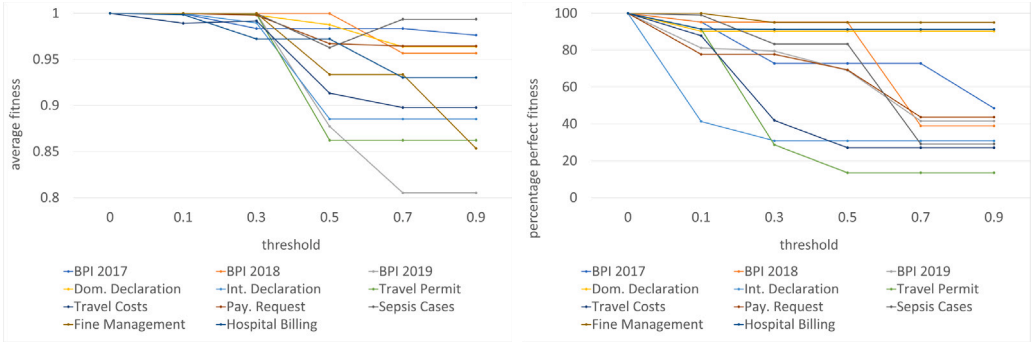


(a) PM_{max}. (b) PM_{clst} with the unfiltered order graph.

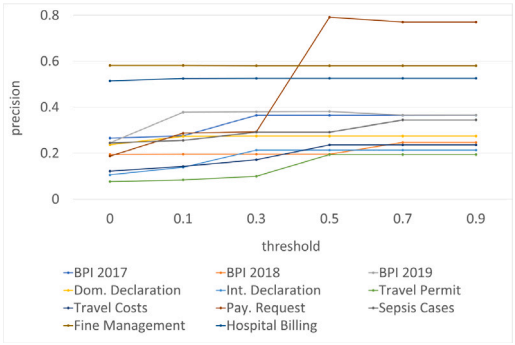


(c) PM_{clst} with a filtered order graph using a threshold of 0.8.

Fig. 19. Models discovered by the different POWL inductive mining approaches for the BPI Challenge 2017 log.



(a) Fitness. (b) Percentage of perfectly fitting traces.



(c) Precision.

Fig. 20. Evaluation results of the decremental weight factor δ.

9.6. Impact of filtering

In this section, we address the research question RQ4 by comparing the two filtering methods we proposed in Section 7. We use the unfiltered variants of all event logs except BPI Challenge 2017 and BPI Challenge 2018. For these two logs, we use the projected logs on the most frequent 12 activities because we were not able to get conformance checking results for the unfiltered ones. We create two variants of PM_{clst}: an extension with dynamic variant filtering using a weight factor $\delta \in \{0.0, 0.1, 0.3, 0.5, 0.7, 0.9\}$, and an extension with order graph filtering using a threshold $t \in \{0.6, 0.7, 0.8, 0.9, 1.0\}$.

For each case, we report the average trace fitness, the percentage of perfectly fitting traces (i.e., traces of fitness 1.0), and the precision score. We aim to identify the parameter values that lead to the best trade-off between fitness and precision.

The evaluation results of the decremental weight factor δ are shown in Fig. 20, and the evaluation results of order graph filtering threshold t are shown in Fig. 21. In general, increasing the value of δ and decreasing the value of t both lead to models of equal or higher precision and equal or lower fitness; i.e., the more behavior we filter out the more constraints we deduce from the data. This applies to most cases in our evaluation with only a few exceptions. One exception is the Payment Request log where increasing δ from 0.5 to 0.7 leads to a drop in precision from 0.79 to 0.77.

Although filtering violates the fitness guarantee and might significantly lower the percentage of perfectly fitting traces, the effect on the average fitness score is limited. However, increasing the value of δ does not necessarily imply improving the quality of the model. For example, changing the value of δ from 0.3 to 0.5 for the BPI Challenge 2019 event log has no notable impact on the precision, but it lowers the model's fitness from 0.99 to 0.88 and the percentage of perfectly fitting traces from 80 to 69. Similarly, decreasing the value of t from 0.8 to 0.7 has a limited impact on both the precision and average fitness scores, but it lowers the percentage of perfectly fitting traces from 60% to 41%.

The values of the filtering thresholds that lead to the best trade-off between fitness and precision vary among the different event logs. For most cases in our evaluation, we obtain optimal results using values between 0.1 and 0.3 for δ and between 0.8 and 0.7 for t.

While both filtering methods have a similar impact on precision, graph order filtering performs better in terms of fitness. Order graph filtering leads to models of a trace fitness higher than 0.97 in all cases, compared to a lower bound of 0.8 for the dynamic variant filtering method. The average percentage of perfectly fitting traces for the models discovered by applying order graph filtering is 82%, compared to 71% for dynamic variant filtering.

9.7. Evaluation summary

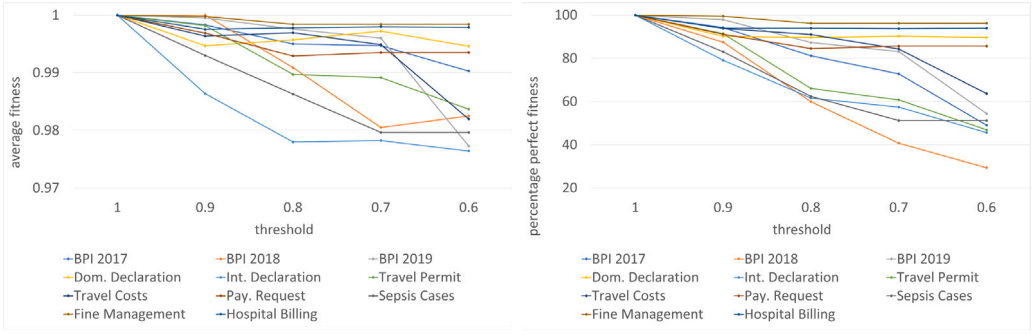
The evaluation shows that the POWL inductive miner is able to discover non-hierarchical structures that cannot be captured by process trees, thus improving the quality of the discovered models. While PM_{bf} faces challenges with larger logs, resulting in timeouts, the other approaches demonstrate better scalability.

By comparing the quality of the models discovered by different POWL inductive mining approaches, the results show that PM_{max} generally yields more precise models. However, the application of order graph filtering in PM_{clst} can significantly enhance precision, underscoring the impact of using different configurations on the quality of the discovered models.

Incorporating filtering in the discovery (both dynamic variant filtering and order graph filtering) often improves the precision of the model but worsens its fitness. The best trade-off is achieved with a careful selection of the filtering threshold. In general, both filtering methods have a similar impact on the precision of the generated models, but order graph filtering produces models of higher fitness.

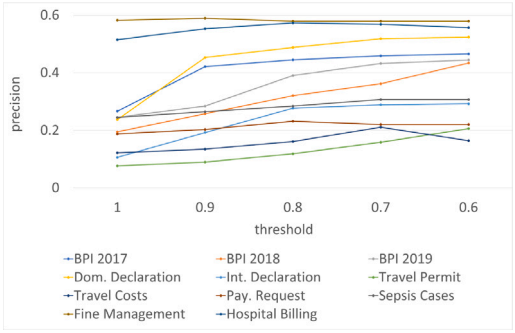
10. Conclusion

A POWL model is a partially ordered graph representation, extended with control-flow operators for modeling choice and loop structures.



(a) Fitness.

(b) Percentage of perfectly fitting traces.



(c) Precision.

Fig. 21. Evaluation results of the order graph filtering threshold ι.

POWL models can be converted into workflow nets, and we showed that the generated Workflow nets are guaranteed to be sound. We proposed several approaches for the discovery of POWL models from event data. The proposed approaches address the challenge of scalability on large data sets and the integration of frequency-based filtering. Evaluation results demonstrate that POWL models, with their combination of hierarchical and partial order elements, provide a comprehensive representation of process behaviors, outperforming traditional process trees in specific scenarios. We have also proposed multiple alternative visualizations of POWL models, offering options that may improve simplicity and interpretability in various use cases.

Future research directions include utilizing life-cycle information contained in event logs for the discovery of POWL models. This involves moving beyond the assumption of events as atomic units in event logs, and instead, considering both the start and end timestamps of each executed activity. Future research will also explore leveraging POWL in process modeling to generate insights derived from both unstructured textual data (e.g., as explored in [39]) and structured event logs. This combined approach promises to yield more complete and nuanced process models that capture both recorded process executions and domain knowledge. Additionally, we aim to develop conformance checking techniques for POWL models to directly assess their quality without transforming them into workflow nets. Lastly, the concept of combining various modeling notations can be applied to create further process modeling languages, opening up new possibilities in the field of process modeling.

CRediT authorship contribution statement

Humam Kourani: Writing – original draft. **Sebastiaan J. van Zelst:** Supervision. **Daniel Schuster:** Supervision. **Wil M.P. van der Aalst:** Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

References

[1] W.M.P. van der Aalst, The application of Petri nets to workflow management, J. Circuits Syst. Comput. 8 (1) (1998) 21–66, http://dx.doi.org/10.1142/S0218126698000043.

[2] S.J.J. Leemans, Robust Process Mining with Guarantees — Process Discovery, Conformance Checking and Enhancement, in: LNBIP, vol. 440, Springer, 2022, http://dx.doi.org/10.1007/978-3-030-96655-3.

[3] H. Kourani, S.J. van Zelst, POWL: Partially Ordered Workflow Language, in: C. Di Francescomarino, A. Burattin, C. Janiesch, S. Sadiq (Eds.), Business Process Management 2023, Proceedings, in: LNCS, vol. 14159, Springer, 2023, pp. 92–108, http://dx.doi.org/10.1007/978-3-031-41620-0_6.

[4] H. Kourani, D. Schuster, W.M.P. van der Aalst, Scalable discovery of partially ordered workflow models with formal guarantees, in: 5th International Conference on Process Mining, ICPM 2023, Rome, Italy, October 23–27, 2023, IEEE, 2023, pp. 89–96, http://dx.doi.org/10.1109/ICPM60904.2023.10271941.

[5] S.J.J. Leemans, S.J. van Zelst, X. Lu, Partial-order-based process mining: a survey and outlook, Knowl. Inf. Syst. 65 (1) (2023) 1–29, http://dx.doi.org/10.1007/S10115-022-01777-3.

[6] D. Schuster, F. Zerbato, S.J. van Zelst, W.M.P. van der Aalst, Defining and visualizing process execution variants from partially ordered event data, Inform. Sci. (ISSN: 0020-0255) 657 (2024) 119958, http://dx.doi.org/10.1016/j.ins.2023.119958.

[7] B.F. van Dongen, J. Desel, W.M.P. van der Aalst, Aggregating causal runs into workflow nets, Trans. Petri Nets Other Model. Concurr. 6 (2012) 334–363, http://dx.doi.org/10.1007/978-3-642-35179-2_14.

[8] R. Bergenthum, J. Desel, R. Lorenz, S. Mauser, Synthesis of Petri nets from infinite partial languages, in: J. Billington, Z. Duan, M. Koutny (Eds.), ACSD 2008, IEEE, 2008, pp. 170–179, http://dx.doi.org/10.1109/ACSD.2008.4574609.

[9] H. Mannila, H. Toivonen, A.I. Verkamo, Discovery of frequent episodes in event sequences, Data Min. Knowl. Discov. 1 (3) (1997) 259–289, http://dx.doi.org/10.1023/A:1009748302351.

[10] M. Leemans, W.M.P. van der Aalst, Discovery of frequent episodes in event logs, in: P. Ceravolo, B. Russo, R. Accorsi (Eds.), Data-Driven Process Discovery and Analysis — 4th International Symposium, SIMPDA 2014, Milan, Italy, November 19–21, 2014, Revised Selected Papers, in: Lecture Notes in Business Information Processing, vol. 237, Springer, 2014, pp. 1–31, http://dx.doi.org/10.1007/978-3-319-27243-6_1.

[11] M. Golani, S.S. Pinter, Generating a process model from a process audit log, in: W.M.P. van der Aalst, A.H.M. ter Hofstede, M. Weske (Eds.), Business Process Management, International Conference, BPM 2003, Eindhoven, the Netherlands, June 26–27, 2003, Proceedings, in: Lecture Notes in Computer Science, vol. 2678, Springer, 2003, pp. 136–151, http://dx.doi.org/10.1007/3-540-44895-0_10.

[12] M. Dumas, L. García-Bañuelos, Process mining reloaded: Event structures as a unified representation of process models and event logs, in: R.R. Devillers, A. Valmari (Eds.), Application and Theory of Petri Nets and Concurrency — 36th International Conference, PETRI NETS 2015, Brussels, Belgium, June 21–26, 2015, Proceedings, in: Lecture Notes in Computer Science, vol. 9115, Springer, 2015, pp. 33–48, http://dx.doi.org/10.1007/978-3-319-19488-2_2.

[13] M. Nielsen, G.D. Plotkin, G. Winskel, Petri nets, event structures and domains, Part I, Theoret. Comput. Sci. 13 (1981) 85–108, http://dx.doi.org/10.1016/0304-3975(81)90112-2.

[14] A. Mokhov, J. Carmona, Event log visualisation with conditional partial order graphs: from control flow to data, in: W.M.P. van der Aalst, R. Bergenthum, J. Carmona (Eds.), Proceedings of the International Workshop on Algorithms & Theories for the Analysis of Event Data, Satellite Event of Petri Nets 2015 and ACSD 2015, Brussels, Belgium, June 22–23, 2015, in: CEUR Workshop Proceedings, vol. 1371, CEUR-WS.org, 2015, pp. 16–30.

[15] A. Mokhov, A. Yakovlev, Conditional partial order graphs: Model, synthesis, and application, IEEE Trans. Comput. 59 (11) (2010) 1480–1493, http://dx.doi.org/10.1109/TC.2010.58.

[16] W.M.P. van der Aalst, R. De Masellis, C. Di Francescomarino, C. Ghidini, H. Kourani, Discovering hybrid process models with bounds on time and complexity: When to be formal and when not? Inf. Syst. 116 (2023) 102214, http://dx.doi.org/10.1016/j.is.2023.102214.

[17] T. Slaats, D.M.M. Schunselaar, F.M. Maggi, H.A. Reijers, 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, in: Lecture Notes in Computer Science, vol. 10033, 2016, pp. 531–551, http://dx.doi.org/10.1007/978-3-319-48472-3_32.

[18] C. Ouyang, E. Verbeek, W.M.P. van der Aalst, S. Breutel, M. Dumas, A.H.M. ter Hofstede, Formal semantics and analysis of control flow in WS-BPEL, Sci. Comput. Program. 67 (2–3) (2007) 162–198, http://dx.doi.org/10.1016/J.SCICO.2007.03.002.

[19] A. Augusto, R. Conforti, M. Dumas, M. La Rosa, F.M. Maggi, A. Marrella, M. Mecella, A. Soo, Automated discovery of process models from event logs: Review and benchmark, IEEE Trans. Knowl. Data Eng. 31 (4) (2019) 686–705, http://dx.doi.org/10.1109/TKDE.2018.2841877.

[20] B.F. van Dongen, A.K.A. de Medeiros, L. Wen, Process mining: Overview and outlook of Petri net discovery algorithms, Trans. Petri Nets Other Model. Concurr. 2 (2009) 225–242, http://dx.doi.org/10.1007/978-3-642-00899-3_13.

[21] W.M.P. van der Aalst, Workflow verification: Finding control-flow errors using Petri-net-based techniques, in: Business Process Management, Models, Techniques, and Empirical Studies, in: Lecture Notes in Computer Science, vol. 1806, Springer, 2000, pp. 161–183, http://dx.doi.org/10.1007/3-540-45594-9_11.

[22] A.J.M.M. Weijters, W.M.P. van der Aalst, Rediscovering workflow models from event-based data using little thumb, Integr. Comput. Aided Eng. 10 (2) (2003) 151–162.

[23] W.M.P. van der Aalst, On the Pareto Principle in Process Mining, Task Mining, and Robotic Process Automation, in: S. Hammoudi, C. Quix, J. Bernardino (Eds.), Proceedings of the 9th International Conference on Data Science, Technology and Applications, DATA 2020, Lieusaint, Paris, France, July 7–9, 2020, SciTePress, 2020, pp. 5–12.

[24] A. Berti, S. van Zelst, D. Schuster, PM4Py: A process mining library for Python, Software Impacts (ISSN: 2665-9638) 17 (2023) 100556, http://dx.doi.org/10.1016/j.simpa.2023.100556.

[25] O. Türetken, T. Rompen, I.T.P. Vanderfeesten, A. Dikici, J. van Moll, The effect of modularity representation and presentation medium on the understandability of business process models in BPMN, in: M. La Rosa, P. Loos, O. Pastor (Eds.), Business Process Management — 14th International Conference, BPM 2016, Rio de Janeiro, Brazil, September 18–22, 2016. Proceedings, in: Lecture Notes in Computer Science, vol. 9850, Springer, 2016, pp. 289–307, http://dx.doi.org/10.1007/978-3-319-45348-4_17.

[26] H.A. Reijers, J. Mendling, Modularity in process models: Review and effects, in: M. Dumas, M. Reichert, M. Shan (Eds.), Business Process Management, 6th International Conference, BPM 2008, Milan, Italy, September 2–4, 2008. Proceedings, in: Lecture Notes in Computer Science, vol. 5240, Springer, 2008, pp. 20–35, http://dx.doi.org/10.1007/978-3-540-85758-7_5.

[27] F. Mannhardt, Sepsis Cases — Event Log, 2016, http://dx.doi.org/10.4121/uuid:915d2bfb-7e84-49ad-a286-dc35f063a460.

[28] E.N. Kamas, L.M. Reder, The role of familiarity in cognitive processing., in: R.F. Lorch, E.J. O'Brien (Eds.), Sources of Coherence in Reading, Lawrence Erlbaum Associates, Inc, 1995, pp. 177–202.

[29] A. Berti, W.M.P. van der Aalst, Reviving token-based replay: Increasing speed while improving diagnostics, in: W.M.P. van der Aalst, R. Bergenthum, J. Carmona (Eds.), Proceedings of the International Workshop on Algorithms & Theories for the Analysis of Event Data, Satellite Event of Petri Nets 2019 and ACSD 2019, in: CEUR Workshop Proceedings, vol. 2371, CEUR-WS.org, 2019, pp. 87–103.

[30] J. Munoz-Gama, J. Carmona, A fresh look at precision in process conformance, in: R. Hull, J. Mendling, S. Tai (Eds.), BPM 2010. Proceedings, in: LNCS, vol. 6336, Springer, 2010, pp. 211–226, http://dx.doi.org/10.1007/978-3-642-15618-2_16.

[31] F.R. Blum, Metrics in process discovery, Tech. Rep. TR/DCC-2015-6, Computer Science Department, Universidad de Chile, Chile, 2015.

[32] M. de Leoni, F. Mannhardt, Road Traffic Fine Management Process, 2015, http://dx.doi.org/10.4121/uuid:270fd440-1057-4fb9-89a9-b699b47990f5.

[33] F. Mannhardt, Hospital Billing — Event Log, 2017, http://dx.doi.org/10.4121/uuid:76c46b83-c930-4798-a1c9-4be94dfeb741.

[34] B. van Dongen, BPI Challenge 2017, 2017, http://dx.doi.org/10.4121/uuid:5f3067df-f10b-45da-b98b-86ae4c7a310b.

[35] B. van Dongen, F. Borchert, BPI Challenge 2018, 2018, http://dx.doi.org/10.4121/uuid:3301445f-95e8-4ff0-98a4-901f1f204972.

[36] B. van Dongen, BPI Challenge 2019, 2019, http://dx.doi.org/10.4121/uuid:d06aff4b-79f0-45e6-8ec8-e19730c248f1.

[37] B. van Dongen, BPI Challenge 2020, 4TU.ResearchData, 2020, http://dx.doi.org/10.4121/uuid:52fb97d4-4588-43c9-9d04-3604d4613b51.

[38] A. Augusto, R. Conforti, M. Dumas, M. La Rosa, A. Polyvyanyy, Split miner: automated discovery of accurate and simple business process models from event logs, Knowl. Inf. Syst. 59 (2) (2019) 251–284, http://dx.doi.org/10.1007/S10115-018-1214-X.

[39] H. Kourani, A. Berti, D. Schuster, W.M.P. van der Aalst, Process modeling with large language models, in: H. van der Aa, D. Bork, R. Schmidt, A. Sturm (Eds.), Enterprise, Business-Process and Information Systems Modeling, Springer Nature Switzerland, Cham, 2024, pp. 229–244, http://dx.doi.org/10.1007/978-3-031-61007-3_18.