Unlocking Non-Block-Structured Decisions: Inductive Mining with Choice Graphs

Humam Kourani[1,2][0000−0003−2375−2152], Gyunam Park[1][0000−0001−9394−6513], and Wil M.P. van der Aalst[1,2][0000−0002−0955−6940]

[1] Fraunhofer Institute for Applied Information Technology FIT, Schloss Birlinghoven, 53757 Sankt Augustin, Germany
{humam.kourani,gyunam.park,wil.van.der.aalst}@fit.fraunhofer.de
[2] RWTH Aachen University, Ahornstraße 55, 52074 Aachen, Germany

Abstract. Process discovery aims to automatically derive process models from event logs, enabling organizations to analyze and improve their operational processes. Inductive mining algorithms, while prioritizing soundness and efficiency through hierarchical modeling languages, often impose a strict block-structured representation. This limits their ability to accurately capture the complexities of real-world processes. While recent advancements like the Partially Ordered Workflow Language (POWL) have addressed the block-structure limitation for concurrency, a significant gap remains in effectively modeling non-block-structured decision points. In this paper, we bridge this gap by proposing an extension of POWL to handle non-block-structured decisions through the introduction of choice graphs. Choice graphs offer a structured yet flexible approach to model complex decision logic within the hierarchical framework of POWL. We present an inductive mining discovery algorithm that uses our extension and preserves the quality guarantees of the inductive mining framework. Our experimental evaluation demonstrates that the discovered models, enriched with choice graphs, more precisely represent the complex decision-making behavior found in real-world processes, without compromising the high scalability inherent in inductive mining techniques.

Keywords: process discovery, process modeling, inductive miner

1 Introduction

Process discovery is a key branch of process mining that aims to automatically construct process models from event data, typically recorded in event logs. By uncovering the "as-is" process, organizations can gain insights into how their processes actually operate, identify bottlenecks, deviations from prescribed procedures, and opportunities for optimization.

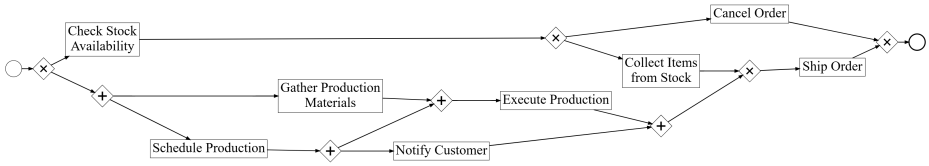
A wide variety of process discovery approaches exist, each with its own strengths and weaknesses, particularly concerning scalability, model quality, and the ability to represent complex process behavior. The Inductive Miner [18] stands out as a prominent process discovery algorithm. A key characteristic of

the Inductive Miner is its use of hierarchical process modeling languages for intermediate representation. This hierarchical nature offers several advantages. Firstly, the discovered models are *sound* by construction, meaning that they are free from common workflow anomalies like deadlocks. Secondly, the hierarchical nature enables a recursive approach to discovery, ensuring efficient processing of large event logs. Furthermore, the generated hierarchical process models can be easily transformed into standard notations, such as BPMN [21] and Petri nets [23], facilitating their use in a variety of analysis and implementation contexts.

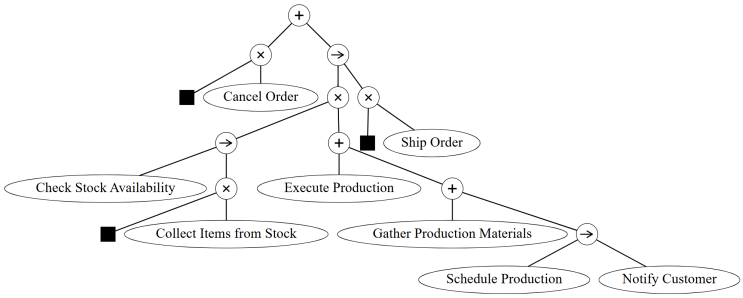
The original Inductive Miner uses *process trees* for intermediate representation. A process tree corresponds to a mathematical tree where leaves represent activities and internal node are control-flow operators that model the relationships between children. The base variant of the Inductive Miner supports four control-flow operators for modeling common process constructs: *sequence*, *exclusive choice*, *concurrency*, and *loop*. While process trees inherently guarantee soundness, their strict block-structured nature limits their expressiveness. The Inductive Miner only discovers well-nested blocks, meaning that every branching split (XOR/AND) has a corresponding matching join in the model.

Let's consider an example of a retailer's order fulfillment process. This retailer handles two types of orders: those for in-stock items that the retailer sells but does not produce itself, and customized orders that require production. The ground truth process model for this process is shown in the BPMN notation in Figure 1a. After receiving an order, the process starts with a decision point to determine the order type. The production sub-process follows a non-block-structured path involving gathering production materials, scheduling production, notifying the customer about the expected delivery date, and executing production. In this sub-process, notifying the customer is independent of gathering the production materials and executing the production, but it happens after scheduling the production so the delivery date can be estimated. Executing the production happens after both gathering the production materials and scheduling the production. This type of non-block-structured concurrency cannot be modeled in a process tree. For example, the tree shown in Figure 1b is the result of applying the base Inductive Miner to an event log that simulates this process. This process tree overgeneralizes the process, allowing production to be executed before scheduling it or collecting the required materials.

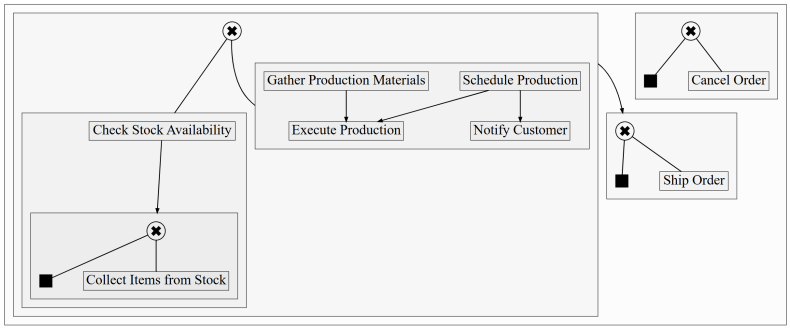
The *Partially Ordered Workflow Language* (POWL) [16] extends the capabilities of process trees. POWL relaxes the block-structure requirement for concurrency by using partial orders instead of the concurrency control-flow operator. A partial order specifies a set of children (i.e., sub-models) that must all be executed, but only imposes a partial ordering constraint on their execution, allowing for flexible interleavings. The Inductive Miner was extended to leverage POWL, allowing for the discovery of process models with non-block-structured concurrency components. For example, Figure 1c shows a POWL model discovered using the POWL Inductive Miner from [15] for our running example. Unlike the process tree, the discovered POWL model can precisely describe the dependencies between the four activities involved in the production sub-process. These



(a) Ground truth BPMN model.



(b) Process tree discovered with the base Inductive Miner [18].

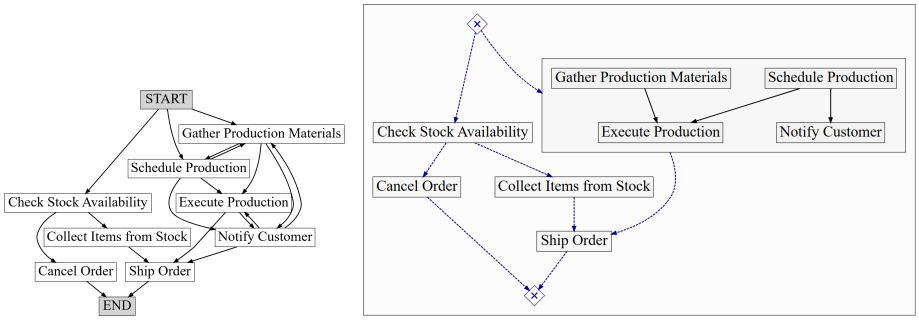


(c) POWL model discovered with the POWL Inductive Miner [15].

Fig. 1: Process models for a retailer's order fulfillment process.

dependencies are correctly described using a compact partial order with three ordering edges: Gather Production Materials → Execute Production, Schedule Production → Execute Production, Schedule Production → Notify Customer.

While POWL eliminates the block-structure limitation for concurrency, this limitation remains for decision points (represented by the XOR operator in both process trees and standard POWL models). In our running example in Figure 1a, there is an initial choice between following the in-stock orders path or the production path. Then, within the in-stock orders path, there is another decision: either cancel the order if the item is out of stock, or join the other path leading to the shipping step. Both the base Inductive Miner and the POWL Inductive Miner fail to model this behavior precisely due to their lack of support for non-block-structured decision points. For instance, the discovered process tree (cf.



(a) Directly-follows graph. (b) POWL 2.0 model.

Fig. 2: Process models for a retailer's order fulfillment process.

Figure 1b) and POWL model (cf. Figure 1c) incorrectly allow the order to be canceled and shipped simultaneously.

Directly-Follows Graphs (DFGs) provide an alternative perspective on process behavior. A DFG is a simple graph where nodes represent activities, and edges represent direct succession relationships. DFGs are computationally efficient to construct from event logs, and they excel at naturally representing complex, non-block-structured branching decisions. For example, the DFG shown in Figure 2a accurately captures the non-block-structured decisions in our process. However, the presence of concurrency in the production sub-process introduces cycles into the DFG, which makes it imprecise. For instance, the DFG allows for skipping the activity "Execute Production" or repeating the sequence ("Gather Production Materials" → "Notify Customer") multiple times.

Our example highlights the strengths of DFGs in modeling decision paths and their tendency to overgeneralize behavior when cycles are present. In a DFG, cycles arise from concurrency or repeated behaviors. POWL provides dedicated, semantically precise operators for these structures (partial orders and loops). This motivates the integration of a graph-based representation similar to DFGs for handling choices within the hierarchical structure of POWL, leveraging the strengths of both approaches while mitigating their individual limitations. Building upon this idea, this paper introduces POWL 2.0, an extension of POWL that enhances its capabilities to model non-block-structured decisions. Specifically, POWL 2.0 replaces the XOR operator with a more expressive construct: the *choice graph*. Inspired by DFGs, choice graphs provide a structured way to represent complex, non-block-structured choices within POWL's hierarchical framework while preserving desirable properties such as soundness.

Figure 2b illustrates a POWL 2.0 model discovered for our running example. The edges of choice graphs are visualized using blue dashed arcs to distinguish them from the solid black arcs in partial orders. In this process model, a choice graph captures the initial choice between the in-stock orders path and the production path, as well as the subsequent choice within the in-stock orders

path—either to cancel the order or to join the other path and proceed to the shipment step. The production path is modeled using a partial order, correctly representing the dependencies between its activities. This example demonstrates POWL 2.0's ability to accurately capture intricate choice behavior that process trees and standard POWL models struggle with, while retaining precise concurrency handling through partial orders.

The remainder of this paper is structured as follows. Section 2 discusses related work, and Section 3 introduces necessary preliminaries. In Section 4, we formally define POWL 2.0 and discuss its quality guarantees. In Section 5, we present a discovery algorithm that extends the Inductive Miner to handle choice graphs, and we prove that this approach preserves the fitness guarantee of the Inductive Miner. Section 6 presents an experimental evaluation of the proposed approach. Section 7 concludes the paper and discusses future work.

2 Related Work

A wide range of process discovery techniques has been developed. For a comprehensive overview, we refer to [4] and [11]. Approaches such as the Inductive Miner [18] and the Evolutionary Tree Miner [9] generate block-structured process models, striking a balance between model quality and simplicity. In [16], POWL models were introduced, enabling the discovery of non-block-structured concurrency with the Inductive Miner. Further adaptations of the Inductive Miner for discovering POWL models were proposed in [15] and [17].

Several approaches discover process models directly in more flexible notations, such as Petri nets and BPMN. For instance, the Split Miner [5] constructs a BPMN model by selectively adding split and join gateways based on log dependencies and frequencies. While the Split Miner can capture complex, non-block-structured dependencies, it does not guarantee soundness. Region-based mining techniques (e.g., [7] and [10]) generate Petri nets without enforcing block-structure constraints. However, these methods are sensitive to noise and can produce overly complex models or suffer from state-space explosion when applied to large event logs, limiting their scalability.

Approaches for directly translating DFGs into Petri nets have been explored (e.g., [19]). In [12] and [6], methods are proposed for discovering prime event structures that integrate conflict relations into partially ordered graphs. These conflict relations offer a flexible way to represent non-block-structured decisions, improving the expressiveness of the resulting models.

The idea of combining different modeling methods to leverage their respective strengths has been explored in various contexts. In [20], a discovery approach is proposed that combines multiple tree-based techniques. Furthermore, Petri nets are extended with additional causal edges in [1], while the approach in [22] combines imperative process models with declarative constraints.

3 Preliminaries

This section presents fundamental notations that are used throughout the paper.

3.1 Basic Notation

A sequence over a set X is an ordered list of elements from X, expressed as $\sigma = \langle \sigma(1), \ldots, \sigma(n) \rangle$. The length of σ is denoted by $|\sigma| = n$, and the set of all sequences over a set X is denoted by X^*. Given two sequences σ_1 and σ_2, their concatenation is written as $\sigma_1 \cdot \sigma_2$, for instance, $\langle x_1 \rangle \cdot \langle x_2, x_1 \rangle = \langle x_1, x_2, x_1 \rangle$. For two sets of sequences L_1 and L_2, we define their concatenation as $L_1 \cdot L_2 = \{\sigma_1 \cdot \sigma_2 \mid \sigma_1 \in L_1,\ \sigma_2 \in L_2\}$. The projection of a sequence σ onto a set Y is denoted by $\sigma{\restriction}_Y$, for example, $\langle x_1, x_2, x_1 \rangle{\restriction}_{\{x_1, x_3\}} = \langle x_1, x_1 \rangle$.

A *multi-set* extends the concept of a set by keeping track of the multiplicities of its elements. A multi-set over a set X is represented as $M = [{x_1}^{c_1}, \ldots, {x_n}^{c_n}]$, where $x_1, \ldots, x_n \in X$ are the elements of M (denoted as $x_i \in M$ for $1 \leq i \leq n$), and $M(x_i) = c_i \geq 1$ specifies the multiplicity of x_i for each $1 \leq i \leq n$. We use $\mathcal{B}(X)$ to denote the set of all multi-sets over a set X.

For a set X, a *partition* of X of size $n \geq 1$ is a set of subsets $P = \{X_1, \ldots, X_n\}$ such that $X = X_1 \cup \cdots \cup X_n$, $X_i \neq \emptyset$, and $X_i \cap X_j = \emptyset$ for all $1 \leq i < j \leq n$. For any $x \in X$, we use P_x to denote the subset in the partition (also called a *part*) that contains x, i.e., $P_x \in \{X_1, \ldots, X_n\}$ and $x \in P_x$. For example, consider the partition $P = \{\{a, b\}, \{c\}\}$ of the set $\{a, b, c\}$. Then, we have $P_a = P_b = \{a, b\}$ and $P_c = \{c\}$.

Let $\prec \subseteq X \times X$ be a binary relation over a set X. We write $x_1 \prec x_2$ to denote $(x_1, x_2) \in \prec$ and $x_1 \not\prec x_2$ to indicate $(x_1, x_2) \notin \prec$. We define the *transitive closure* of $\prec$ as $\prec^+ = \{(x, y) \mid \exists_{x_1, \ldots, x_n \in X}\ x = x_1\ \wedge\ y = x_n\ \wedge \forall_{1 \leq i < n} x_i \prec x_{i+1}\}$.

A *strict partial order* (or simply *partial order*) over a set X is a binary relation that satisfies *irreflexivity* ($x \not\prec x$ for all $x \in X$) and *transitivity* ($x_1 \prec x_2 \wedge x_2 \prec x_3 \Rightarrow x_1 \prec x_3$). These properties together imply *asymmetry* ($x_1 \prec x_2 \Rightarrow x_2 \not\prec x_1$). For $n \geq 2$, $\mathcal{O}^n$ denotes the set of all partial orders over $\{1, \ldots, n\}$. Given a set $X = \{x_1, \ldots, x_n\}$ with $n \geq 2$ and a partial order $\prec \in \mathcal{O}^n$, we use $\prec(x_1, \ldots, x_n)$ to denote the partial order $\prec'$ over X defined as follows: $i \prec j \Leftrightarrow x_i \prec' x_j$ for all i, $j \in \{1, \ldots, n\}$.

Let $\sigma_1, \ldots, \sigma_n \in X^*$ be sequences over a set X with $n \geq 2$, and let $\prec \in \mathcal{O}^n$. The *order-preserving shuffle operator* $\sqcup\!\sqcup_{\prec}$ produces the set of sequences obtained by interleaving $\sigma_1, \ldots, \sigma_n$ while maintaining both the partial order $\prec$ among the sequences and the inherent sequential order within each sequence. For example, consider the sequences $\sigma_1 = \langle a, b \rangle$, $\sigma_2 = \langle c \rangle$, $\sigma_3 = \langle d, e \rangle$, and the partial order $\prec = \{(1, 2), (1, 3)\} \in \mathcal{O}^3$. Then, the set of valid interleavings is $\sqcup\!\sqcup_{\prec}(\sigma_1, \sigma_2, \sigma_3) = \{\langle a, b, c, d, e \rangle, \langle a, b, d, c, e \rangle, \langle a, b, d, e, c \rangle\}$.

3.2 Event Log

We use Σ to denote the set of all activities. Additionally, we introduce $\tau \notin \Sigma$ to denote the *silent activity*, which is often used to model choices between executing or skipping a path in a process model.

An *event log* $L \in \mathcal{B}(\Sigma^*)$ is a multi-set of sequences of activities. A *trace* $\sigma \in L$ represents the execution of a single process instance as a sequence of activities. Given an event log L, we define the following notations:

- $\Sigma_L = \{a \in \sigma \mid \sigma \in L\}$ is the set of activities appearing 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.
- $\mapsto \subseteq \Sigma_L \times \Sigma_L$ is the *Directly-Follows Graph (DFG) relation*, defined as follows:

$$a \mapsto b \quad \text{iff} \quad \exists_{\sigma \in L, 1 \leq i < |\sigma|}\ \sigma(i) = a \ \wedge\ \sigma(i+1) = b.$$

For example, consider the event log $L_1 = [\langle a,b,c\rangle^3, \langle a,b,d\rangle^2]$, which consists of five traces. The corresponding sets are $\Sigma_{L_1} = \{a,b,c,d\}$, $L_{1\triangleright} = \{a\}$, and $L_{1\square} = \{c,d\}$. The directly-follows graph of L_1 is $\mapsto = \{(a,b),(b,c),(b,d)\}$.

4 Extending POWL with Choice Graphs

In this section, we introduce a new class of models that extends POWL, enabling a more expressive representation of branching behavior. Furthermore, we discuss the guarantees and limitations of this new class.

A POWL model [16] is constructed recursively from a set of activities, combined using partial orders and the control flow operators $\times$ and $\circlearrowleft$. The operator $\times$ represents an exclusive choice between submodels, whereas $\circlearrowleft$ captures cyclic behavior between two submodels: the *do-part* is executed first, and each execution of the *redo-part* is followed by another execution of the do-part. In a partial order, all submodels are executed while preserving the specified execution order.

To enable non-block-structured decisions in POWL, we introduce *choice graphs*, a structured approach to modeling exclusive choice paths in processes. A choice graph consists of a set of nodes connected by directed edges such that each node lies on a path from a designated *start node* to a designated *end node*.

Definition 1 (Choice Graph). *A* choice graph *over a set of nodes X is a tuple $G = (N, E)$ where:*

- *$N = X \cup \{\triangleright, \square\}$ with two artificial start and end nodes $\triangleright, \square \notin X$.*
- *$E \subseteq N \times N$ is binary relation over N.*
- *$\triangleright$ is the unique start node, i.e., $\{\triangleright\} = \{x \in N \mid (N \times \{x\}) \cap E = \emptyset\}$.*
- *$\square$ is the unique end node, i.e., $\{\square\} = \{x \in N \mid (\{x\} \times N) \cap E = \emptyset\}$.*
- *Every node is on a connected path from $\triangleright$ to $\square$.*

We use $\mathcal{DG}(X)$ to denote the set of all choice graphs over a set X. A choice graph represents a collection of possible execution paths, each beginning at the start node and ending at the end node. Formally, for a choice graph $G = (N, E) \in \mathcal{DG}(X)$ over a set X, we define the set of all paths in G as follows:

$$\overrightarrow{G} = \{\langle x_1, ..., x_n\rangle \in X^* \mid (\triangleright, x_1), (x_1, x_2), ..., (x_{n-1}, x_n), (x_n, \square) \in E\}.$$

Note that Definition 1 provides a general structure for choice graphs that could, in principle, include cycles. However, in our discovery approach, we restrict ourselves to acyclic choice graphs and model explicit loops using the dedicated operator $\circlearrowleft$. The general definition nonetheless leaves the door open for future extensions to support the discovery of more complex cyclic decision structures.

With choice graphs established, we now define *POWL 2.0*, an extended version of POWL that supports more complex decision-making structures while preserving its fundamental principles. The key difference from standard POWL is that exclusive choices between submodels are now represented using choice graphs instead of the $\times$-operator.

Definition 2 (POWL 2.0). *A POWL model is recursively defined as follows:*

- *An activity $a \in \Sigma \cup \{\tau\}$ is a POWL model.*
- *Let $P = \{\psi_1, ..., \psi_n\}$ be a set of $n \geq 2$ POWL models.*
  - *$\circlearrowleft(\psi_1, \psi_2)$ is a POWL model.*
  - *A choice graph $G \in \mathcal{DG}(P)$ is a POWL model.*
  - *For a partial order $\prec \in \mathcal{O}^n$, $\prec(\psi_1, ..., \psi_n)$ is a POWL model.*

Similar to the original POWL framework [16], we define the semantics of a POWL 2.0 model recursively based on the semantics of its operators. The introduction of choice graphs alters how choices are represented. The language of a choice graph is defined as the set of sequences obtained by concatenating sequences from the languages of the submodels along a valid path in the choice graph.

Definition 3 (POWL 2.0 Semantics). *The language of a POWL model is recursively defined as follows:*

- *$\mathcal{L}(a) = \{\langle a \rangle\}$ for $a \in \Sigma$.*
- *$\mathcal{L}(\tau) = \{\langle\rangle\}$.*
- *$\mathcal{L}(\circlearrowleft(\psi_1, \psi_2)) = \mathcal{L}(\psi_1) \cdot (\mathcal{L}(\psi_2) \cdot \mathcal{L}(\psi_1))^*$.*
- *Let $P = \{\psi_1, ..., \psi_n\}$ be a set of $n \geq 2$ POWL models.*
  - *For $\prec \in \mathcal{O}^n$, $\mathcal{L}(\prec(\psi_1, ..., \psi_n)) = \{\sigma \in \sqcup\!\sqcup_{\prec}(\sigma_1, ..., \sigma_n) \mid \forall_{1 \leq i \leq n}\ \sigma_i \in \mathcal{L}(\psi_i)\}$.*
  - *For $G \in \mathcal{DG}(P)$, $\mathcal{L}(G) = \bigcup_{\Pi \in \overrightarrow{G}} \mathcal{L}(\Pi_1) \cdot ... \cdot \mathcal{L}(\Pi_{|\Pi|})$.*

Note that POWL models can be defined over *transitions* to allow for duplicating activities (i.e., each transition is considered as an instance of an activity). However, we assume a one-to-one mapping between transitions and activities in this paper and we define POWL models over activities for simplicity.

Discussion: Conversion to WF-Nets, Guarantees, Limitations

POWL 2.0 models can be systematically converted into equivalent Workflow Nets (WF-nets), which, in turn, can be translated into BPMN models. A WF-net [23] is a Petri net with a unique source place and a unique sink place, where

every other place and transition lies on a path from the source to the sink. The conversion algorithm, extending the one for standard POWL from [17], recursively translates each POWL 2.0 construct into a WF-net fragment. Translating a choice graph into a WF-net involves: (i) introducing a unique source and sink place, (ii) recursively converting each sub-model (i.e., node in the choice graph) into its corresponding WF-net, and (iii) adding silent transitions to connect the sub-nets according to the edges of the choice graph. Adapting the conversion algorithm to handle choice graphs preserves its key guarantees: language equivalence and soundness. Formal inductive proofs extending the proofs from [17] can be easily constructed.

A *workflow graph* is a directed graph whose vertices are either activities or BPMN-style split/join gateways. It can be trivially shown that every POWL 2.0 construct can be interpreted as a single-entry-single-exit (SESE) fragment of such a graph. In other words, any POWL 2.0 model expands to a workflow graph that is hierarchically composed of SESE fragments containing only XOR or only AND gateways. Such workflow graphs can translated into equivalent free-choice workflow nets [13].

POWL 2.0's expressiveness places it within the hierarchy of sound WF-net subclasses. A *marked graph* is a Petri net where each place has at most one incoming arc and at most one outgoing arc. A *state machine* is a Petri net where each transition has at most one incoming arc and at most one outgoing arc. Any sound marked-graph WF-net can be represented in POWL 2.0 using a partial order, while any sound state-machine WF-net can be represented using a choice graph. More broadly, POWL can effectively represent *semi-block-structured* WF-nets [14], and the introduction of choice graphs further enhances the expressiveness of POWL to cover a broader class of free-choice WF-nets.

Although POWL 2.0 removes the block-structure requirements for both concurrency and decision components, allowing more complex dependencies within each of those structures, it still enforces a block-structure between different types of operators. In other words, POWL 2.0 does not support structures where decision and concurrency split/join points interleave arbitrarily.

5 Discovery of POWL 2.0 Models

In this section, we propose an approach for the discovery of POWL 2.0 model from event logs.

5.1 Inductive Miner

The Inductive Miner [18] is a process discovery algorithm that constructs process trees from event logs in a recursive manner. This approach was extended to discover POWL models by adding support for partial orders. Various variants of the POWL Inductive Miner were proposed in [16], [15], and [17], differing primarily in how partial order cuts are identified while maintaining the same overall methodology. Our new methodology can be integrated with any of these

variants as the proposed changes do not affect the partial order cut detection step. In the remainder of the paper, we use the term POWL Inductive Miner (PM) to refer to the variant defined in [15].

The Inductive Miner follows a recursive, top-down approach. It attempts to identify a *cut*, which involves detecting a behavioral pattern in the event log and partitioning the activities accordingly. The algorithm recognizes different types of cuts that correspond to one of the process tree or POWL operators ($\times$, $\circlearrowleft$, $\rightarrow$, $+$, and partial orders). Once a cut is identified, the event log is projected onto the identified parts (i.e., activity subsets), generating smaller sub-logs. The same procedure is then recursively applied to each sub-log until a *base case* is reached—either an empty event log or an event log containing a single activity. In such cases, the base case is trivially converted into a process tree or a POWL model.

If neither a base case nor a cut can be found, the Inductive Miner applies a *fall-through* function, which ensures that the recursion continues. This function may, for example, place an activity that appears in every trace into concurrency with the remaining activities. For a comprehensive overview of the base Inductive Miner and its adaptation to POWL, we refer to [18] and [17], respectively.

5.2 Mining for Choice Graphs

In this section, we extend PM to discover POWL 2.0 models. We achieve this extension by replacing the $\times$-cut detection step with a new procedure for detecting choice graphs. This allows for a more expressive representation of choices in process models while preserving the general structure of the inductive mining approach.

To incorporate choice graphs into the discovery process, we introduce the notion of a *choice graph cut*, which involves partitioning the activities and assigning a choice graph structure over the partition to model branching behavior.

Definition 4 (Choice Graph Cut). *Let L be an event log. A choice graph cut over L is a tuple (A, G) where $A = \{A_1, ..., A_n\}$ is a partition of Σ_L into $n \geq 2$ parts and $G \in \mathcal{DG}(A)$ is a choice graph over A.*

Not every choice graph cut provides a valid representation of an event log. To ensure correctness, we impose structural constraints that align the choice graph with the event log's DFG. Additionally, we add an acyclicity requirement. This restriction aims at enabling a more precise representation for repetition and concurrency behaviors using POWL's dedicated constructs—partial orders for concurrency and loop operators for cyclic behavior.

Notation. For two subsets of activities $A, B \subseteq \Sigma_L$, we extend the notation for the DFG relation as follows:

$$A \mapsto B \quad \text{iff} \quad \exists_{a \in A, b \in B}\ a \mapsto b.$$

Input: An event log L.
Output: A partition A of the activities of L.
Function `MineDG`(L)**:**
  $A \leftarrow \{\{a\} \mid a \in \Sigma_L\}$
  for $(a_1, a_2) \in \Sigma_L \times \Sigma_L$ **do**
    if $(a_1 \mapsto^+ a_2 \ \wedge \ a_2 \mapsto^+ a_1)$ **then**
      $A \leftarrow A \setminus \{A_{a_1}, A_{a_2}\} \cup \{A_{a_1} \cup A_{a_2}\}$
  return A

Algorithm 1: Generating a candidate partition for a choice graph cut.

Definition 5 (Valid Choice Graph Cut). *Let $C = (A = \{A_1, ..., A_n\}, G = (N, E))$ be a choice graph cut over an event log L. C is valid if and only if for all $A_i, A_j \in \{A_1, ..., A_n\}$:*

1. $(A_i \mapsto A_j \ \wedge \ A_i \neq A_j) \Leftrightarrow (A_i, A_j) \in E$.
2. $A_i \cap L_{\triangleright} \neq \emptyset \Leftrightarrow (\triangleright, A_i) \in E$.
3. $A_i \cap L_{\square} \neq \emptyset \Leftrightarrow (A_i, \square) \in E$.
4. $\langle\rangle \in L \Leftrightarrow (\triangleright, \square) \in E$.
5. $(A_i \mapsto^+ A_j \ \wedge \ A_j \mapsto^+ A_i) \Rightarrow A_i = A_j$.

The first condition connects DFG edges to the choice graph's edges. The second and third conditions ensure that subsets containing start/end activities connect to the artificial start/end nodes. The forth condition links empty traces in the event log to a direct edge from start to end in the choice graph. The last condition enforces that if two parts are mutually reachable in the DFG, they must be the same part, introducing an acyclicity requirement.

Note that the requirements (1-4) of Definition 5 uniquely define a binary relation for any given partitioning of activities. In other words, in order to detect a choice graph cut, we need to generate a partition of activities that conforms with the acyclicity requirement and contains at least two parts.

Algorithm 1 generates a candidate partition of activities for a choice graph cut that ensures acyclicity. It achieves this by (i) initially putting every activity in a single part and (2) only merging two parts if they contain activities that are mutually reachable in the DFG. If the returned partition A consists of a single part, then no valid choice graph cut exists. Otherwise, the requirements of Definition 5 define a unique choice graph over A.

After identifying a valid choice graph cut, the event log L is projected on the different parts, creating sub-logs.

Definition 6 (Choice Graph Projection). *Let L event log L and $A \subseteq \Sigma_L$ be a subset of its activities. The choice graph projection of L onto A is defined as $proj(L, A) = \{\sigma\restriction_A \mid \sigma \in L \wedge \sigma\restriction_A \neq \langle\rangle\}$.*

We use $PM_\times$ to denote our extended variant of PM that mines for valid choice graph cuts instead of $\times$-cuts. If a valid choice graph cut $(A = \{A_1, ..., A_n\}, G = (N, E))$ over an event log L is identified, $PM_\times$ proceeds as follows:

1. The event log L is projected onto each part A_i, creating a sub-log $L_i = proj(L, A_i)$.
2. The algorithm is recursively applied on each sub-log L_i, creating a POWL model ψ_i.
3. The algorithm maps each part A_i into its corresponding POWL model ψ_i in G, creating a new choice graph G'.
4. The algorithm returns G'.

Several reduction rules can be applied to a POWL 2.0 model generated by $PM_\times$. For example, pure sequential behavior can be discovered using both partial order cuts and choice graph cuts. To ensure consistency, all such sequences are converted into partial orders. Additionally, nodes in a choice graph that share the same preceding and succeeding nodes are grouped together to enhance understandability. These reduction rules do not alter the language of the generated POWL 2.0 model and are therefore considered optional.

To ensure that the discovered POWL 2.0 models adequately represent the given event log, we establish a fitness guarantee for $PM_\times$. Specifically, we prove that every trace in the event log is included in the language of the discovered model.

Lemma 1 (Fitness Preservation for Valid Choice Graph Cuts). *Let L be an event log and let $(A = (A_1, \ldots, A_n), G = (N, E))$ be a valid choice graph cut over L. Let G' be the choice graph obtained by replacing each A_i in G with the POWL model $\psi_i = PM_\times(proj(L, A_i))$. Assume that $proj(L, A_i) \subseteq \mathcal{L}(\psi_i)$ for all $1 \leq i \leq n$. Then for all $\sigma \in L$, we have $\sigma \in \mathcal{L}(G')$.*

Proof. Let $\sigma \in L$. We need to show that $\sigma \in \mathcal{L}(G')$. Recall that the language of the choice graph G' is defined as:

$$\mathcal{L}(G') = \bigcup_{\Pi' = \langle \psi_1, \ldots, \psi_k \rangle \in \overrightarrow{G'}} \mathcal{L}(\psi_1) \cdot \ldots \cdot \mathcal{L}(\psi_k).$$

This can be rewritten as:

$$\mathcal{L}(G') = \bigcup_{\Pi = \langle A_1, \ldots, A_k \rangle \in \overrightarrow{G}} \mathcal{L}(PM_\times(proj(L, A_1))) \cdot \ldots \cdot \mathcal{L}(PM_\times(proj(L, A_k))).$$

We construct a suitable path $\Pi = \langle A_1, \ldots, A_k \rangle \in \overrightarrow{G}$ such that σ is the concatenation of traces from the languages of the sub-models along that path. We have two cases:

- **Case $\sigma = \langle\rangle$:** By Definition 5, if the empty trace is in L, then $(\triangleright, \square) \in E$. This gives us a direct path representing $\langle\rangle$. Consequently, $\langle\rangle \in \mathcal{L}(G')$.
- **Case $\sigma = \langle a_1, \ldots, a_m \rangle$ with $m > 0$:** We build the corresponding path $\Pi = \langle A_1, \ldots, A_k \rangle \in \overrightarrow{G}$ $(k \leq m)$ as follows:
  1. **Start node:** Since $a_1 \in L_\triangleright$, by Definition 5, it follows that $(\triangleright, A_{a_1}) \in E$. Thus, we start the path with A_{a_1}, i.e., $A_1 = A_{a_1}$.

2. **Intermediate nodes:** For subsequent activities a_j in σ ($2 \leq j \leq m$):
   - **Case** $A_{a_j} = A_{a_{j-1}}$**:** We remain within the same part and do not extend the path.
   - **Case** $A_{a_j} \neq A_{a_{j-1}}$**:**
     a_{j-1} is directly followed by a_j in σ, implying $A_{a_{j-1}} \mapsto A_{a_j}$. Therefore, we can conclude by the first condition of Definition 5 that $(A_{a_{j-1}}, A_{a_j}) \in E$ holds, and hence we extend our path with A_{a_j}.
3. **End node:** Since $a_m \in L_\square$, Definition 5 implies $(A_{a_m}, \square) \in E$. This shows that Π is a valid path from $\triangleright$ to $\square$ in G.

Thus, for every $\sigma \in L$, we have constructed a corresponding path $\Pi = \langle A_1, \ldots, A_k \rangle \in \overrightarrow{G}$. Now, decompose σ into sub-traces $\sigma_1, \ldots, \sigma_k$ by projecting it on the different path steps $A_1, \ldots, A_k$, i.e., $\sigma_i = \sigma\restriction_{A_i} \in \mathit{proj}(L, A_i)$. By the assumption of the lemma, each $\sigma_i \in \mathcal{L}(\psi_i)$. Since a valid choice graph is cycle-free, no part can be visited more than once within a single path. Therefore, concatenating the sub-traces in accordance with the path Π yields σ. Thus:

$$\sigma = \sigma_1 \cdot \ldots \cdot \sigma_k \in \mathcal{L}(\psi_1) \cdot \ldots \cdot \mathcal{L}(\psi_k) \subseteq \mathcal{L}(G').$$

Theorem 1 (Fitness Guarantee). *Let L be an event log and $\psi = PM_\times(L)$. Every trace $\sigma \in L$ is included in the model's language, i.e., $\sigma \in \mathcal{L}(PM_\times(L))$.*

Proof. We prove this theorem by induction on the recursive application of the algorithm as it constructs the POWL 2.0 model.
Base Case: If the algorithm detects a base case, then the theorem trivially holds [17].
Inductive Hypothesis: Assume the theorem holds for smaller sub-logs. This assumption means that any sub-model generated by the recursive application of the algorithm includes all traces of the corresponding sub-log in its language.
Inductive Step: We distinguish two cases:

1. **If a standard POWL cut is detected or the fall-through function is invoked:** The theorem holds since these cuts and the fall-through function are fitness-preserving [17].
2. **If a valid choice graph cut** $(A = (A_1, \ldots, A_n), G)$ **is detected:** The algorithm maps each part A_i into its corresponding POWL 2.0 model $\psi_i = PM_\times(\mathit{proj}(L, A_i))$ in G, returning another choice graph G'. By the inductive hypothesis, each sub-trace $\sigma_i \in \mathit{proj}(L, A_i)$ is in the language of the recursively discovered sub-model, i.e., $\sigma_i \in \mathcal{L}(PM_\times(\mathit{proj}(L, A_i)))$. Therefore, we can apply Lemma 1, which directly gives us that for all $\sigma \in L$, $\sigma \in \mathcal{L}(G')$.

Thus, by induction, any trace $\sigma \in L$ is included in $\mathcal{L}(PM_\times(L))$.

Filtering Infrequent Behavior. Achieving a perfect fitness is not always desirable in real-life settings. The inductive mining framework often incorporates noise filtering mechanisms to improve the quality of discovered models. A common approach, and one utilized in our experimental evaluation (see Section 6), is

DFG-based noise filtering [18]. This filtering is applied at each recursive step: before attempting to find any cut, the DFG derived from the current (sub-)log is pruned by removing edges (directly-follows relations) that fall below a certain frequency threshold, relative to other edges involving the same source activity. This is an established filtering step within the inductive mining framework, which can be directly inherited by our approach, rather than a modification to the core cut-finding logic itself.

6 Evaluation

We implemented $PM_\times$ as an extension to the standard POWL Inductive Miner (PM). Both approaches are available in the Python library 'powl', which can be installed via 'pip install powl' (`https://doi.org/10.5281/zenodo.15655784`). An example script for using the library is available at `https://github.com/humam-kourani/POWL/blob/main/examples/powl_discovery.py`. Additionally, a web application demonstrating the approach is accessible online at `https://powl-miner.streamlit.app/`.

We compare the two discovery approaches, evaluating two variants of each: one without any noise filtering and another variant applying the DFG-based noise filtering approach proposed in [18]. We use a noise filtering threshold of 0.2. The experiments are conducted using 17 real-life event logs from `https://data.4tu.nl/`. For event logs that include life-cycle information with completion events (i.e., events with the state 'complete'), we only consider these completion events.

We assess both runtime performance and the quality of the generated process models. Model quality is evaluated by converting the discovered models into WF-nets and then applying two standard conformance checking metrics: fitness and precision. Fitness measures how well the model reproduces the behavior recorded in the event log [2], while precision assesses how strictly the model conforms to the observed behavior [3]. We use the fitness and precision implementations available in PM4Py [8] and set a timeout of six hours for each conformance checking computation. In addition to fitness and precision, we report the f-score, which is their harmonic mean.

All discovered process models are available at `https://doi.org/10.5281/zenodo.15050973`. The evaluation results are summarized in Table 1. For the case with no filtering, we only report the f-scores as quality indicators since all discovered models achieved perfect fitness, which is guaranteed by both PM and $PM_\times$.

Overall, we observe that the introduction of choice graphs has no negative impact on runtime performance. In cases without noise filtering, there is no significant difference in execution time between PM and $PM_\times$. However, with the noise filtering threshold of 0.2, $PM_\times$ exhibited improved runtime performance for some event logs. For instance, for the BPIC2019 event log, the execution time decreased from 287 seconds with PM to 20 seconds with $PM_\times$. This demonstrates that mining for choice graphs does not compromise the inherent scalability of

Table 1: Evaluation results. Missing values indicate timeouts. Higher quality scores for $PM_\times$ compared to PM are highlighted in green, while lower scores are highlighted in red. Significant time improvements are also highlighted in green.

Event Log	**No Filtering**				**Noise Threshold t=0.2**							
	Time (sec)		F-Score		Time (sec)		Fitness		Precision		F-Score	
	PM	$PM_\times$	PM	$PM_\times$	PM	$PM_\times$	PM	$PM_\times$	PM	$PM_\times$	PM	$PM_\times$
BPIC2012	19.3	18.6	0.27	0.28	11.0	1.9	0.98	0.89	0.26	0.47	0.41	0.62
BPIC2013 - Cl. Problems	0.0	0.1	0.88	0.88	0.1	0.1	0.99	0.99	0.95	0.95	0.97	0.97
BPIC2013 - Incidents	0.4	0.6	0.77	0.77	0.5	0.5	0.93	0.91	0.72	0.87	0.81	0.89
BPIC2013 - Open Problems	0.0	0.0	0.95	0.95	0.0	0.0	0.83	0.83	0.91	0.91	0.87	0.87
BPIC2017	15.2	14.8	0.45	0.47	15.2	2.7	0.99	0.98	0.32	0.55	0.48	0.70
BPIC2017 - Offer Log	0.4	0.3	0.91	1.00	0.5	0.5	1.00	1.00	0.90	1.00	0.94	1.00
BPIC2018	606.2	601.0	–	–	192.5	189.8	–	–	–	–	–	–
BPIC2019	200.3	198.5	–	–	287.3	20.2	–	–	–	–	–	–
BPIC2020 - Dom. Decl.	0.3	0.3	0.56	0.63	0.1	0.1	0.93	0.90	0.38	0.62	0.54	0.74
BPIC2020 - Int. Decl.	2.3	2.3	0.24	0.31	0.7	0.4	0.88	0.84	0.34	0.58	0.50	0.68
BPIC2020 - Permit Log	22.0	21.9	0.16	0.17	3.7	0.9	0.79	0.81	0.19	0.63	0.31	0.71
BPIC2020 - Prepaid Travel	1.4	1.4	0.22	0.22	0.3	0.1	0.87	0.88	0.42	0.74	0.56	0.80
BPIC2020 - Req. Payment	0.2	0.2	0.45	0.51	0.1	0.1	0.91	0.89	0.39	0.47	0.54	0.61
Hospital Billing	2.1	2.0	0.67	0.67	2.0	1.2	0.96	0.97	0.56	0.93	0.71	0.95
Receipt Phase WABO	0.5	0.5	0.28	0.30	0.1	0.0	0.81	0.88	0.25	0.55	0.38	0.67
Road Traffic Fine	1.0	0.9	0.74	0.74	0.9	0.9	0.87	0.98	0.78	1.00	0.82	0.99
Sepsis Cases	1.2	1.2	0.44	0.44	0.5	0.2	0.91	0.91	0.40	0.54	0.56	0.68

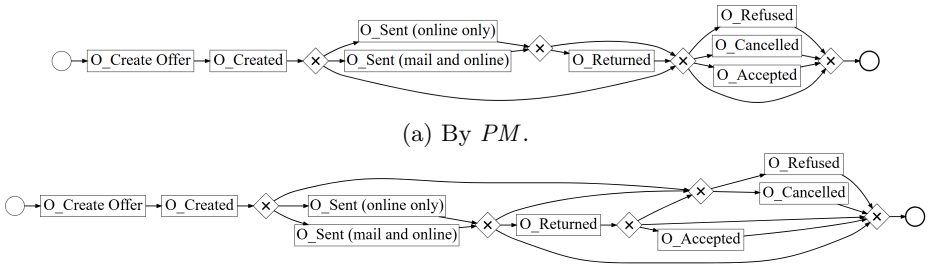
the Inductive Miner; in fact, it can improve runtime performance when valid choice graphs are detected.

Regarding quality, we observe that $PM_\times$ achieved equal or higher precision and f-score compared to PM in all cases. In 21 out of the 30 cases with no timeouts, $PM_\times$ led to more precise models. In some instances, fitness dropped slightly, but these decreases were compensated by notable increases in precision, leading to consistently higher f-scores. In other cases, such as the Receipt Phase WABO log, mining for decision graphs with noise filtering led to improvements in both fitness (from 0.81 to 0.88) and precision (from 0.25 to 0.55).

Figure 3 presents the process models discovered for the BPIC2017 - Offer Log, converted into BPMN. In this case, mining for choice graphs increased the f-score from 0.91 to 1.0. One notable improvement for this event log is that the model discovered by PM allows an offer to be accepted immediately after creation, while the more precise model generated by $PM_\times$ ensures that offers can only be accepted after they have been sent and subsequently returned. This example highlights the advantage of incorporating choice graphs in process discovery, leading to improved precision without compromising scalability.

7 Conclusion

This paper introduced POWL 2.0, an extension of the Partially Ordered Workflow Language (POWL) designed to enable the modeling of non-block-structured decisions. By replacing the traditional exclusive choice operator with choice graphs, POWL 2.0 provides a more expressive and flexible way to represent complex branching logic within the hierarchical framework of POWL. We presented a discovery algorithm that extends the Inductive Miner to mine for acyclic



(a) By PM.

(b) By $PM_{\times}$.

Fig. 3: Process models discovered for the BPIC2017 - Offer Log.

choice graphs, and we formally proved that this algorithm preserves the fitness guarantee of the Inductive Miner. Our experimental evaluation demonstrated the ability of choice graphs to capture intricate decision-making patterns in real-world processes.

One direction for future work is to explore further extensions to POWL to capture more complex process structures, while carefully considering the trade-offs between expressiveness, scalability, and the preservation of formal guarantees. While the current discovery approach mines only for acyclic choice graphs, POWL 2.0 itself allows for more general decision structures. Future research will explore the controlled integration of cyclic choice graphs in process discovery. Furthermore, we plan to investigate the integration of POWL with other process mining and business process management techniques beyond process discovery.

References

1. van der Aalst, W.M.P., De Masellis, R., Di Francescomarino, C., Ghidini, C., Kourani, H.: Discovering hybrid process models with bounds on time and complexity: When to be formal and when not? Information Systems **116**, 102214 (2023)
2. Adriansyah, A., van Dongen, B.F., van der Aalst, W.M.: Conformance checking using cost-based fitness analysis. In: 2011 IEEE 15th international enterprise distributed object computing conference. pp. 55–64. IEEE (2011)
3. Adriansyah, A., Munoz-Gama, J., Carmona, J., van Dongen, B.F., van der Aalst, W.M.P.: Measuring precision of modeled behavior. Inf. Syst. E Bus. Manag. **13**(1), 37–67 (2015)
4. Augusto, A., Conforti, R., Dumas, M., Rosa, M.L., Maggi, F.M., Marrella, A., Mecella, M., Soo, A.: Automated discovery of process models from event logs: Review and benchmark. IEEE Trans. Knowl. Data Eng. **31**(4), 686–705 (2019)
5. Augusto, A., Conforti, R., Dumas, M., Rosa, M.L., Polyvyanyy, A.: Split miner: automated discovery of accurate and simple business process models from event logs. Knowl. Inf. Syst. **59**(2), 251–284 (2019)
6. van Beest, N.R.T.P., Dumas, M., García-Bañuelos, L., Rosa, M.L.: Log delta analysis: Interpretable differencing of business process event logs. In: Business Process Management - 13th International Conference, Proceedings. LNCS, vol. 9253, pp. 386–405. Springer (2015)

7. Bergenthum, R., Desel, J., Lorenz, R., Mauser, S.: Process mining based on regions of languages. In: International Conference on Business Process Management. pp. 375–383. Springer (2007)
8. Berti, A., van Zelst, S., Schuster, D.: PM4Py: A process mining library for python. Software Impacts **17**, 100556 (2023)
9. Buijs, J.C., van Dongen, B.F., van der Aalst, W.M.P.: A genetic algorithm for discovering process trees. In: 2012 IEEE Congress on Evolutionary Computation. pp. 1–8. IEEE (2012)
10. Carmona, J., Cortadella, J., Kishinevsky, M.: A region-based algorithm for discovering petri nets from event logs. In: Business Process Management: 6th International Conference, BPM 2008, Milan, Italy, September 2-4, 2008. Proceedings 6. pp. 358–373. Springer (2008)
11. van Dongen, B.F., de Medeiros, A.K.A., Wen, L.: Process mining: Overview and outlook of Petri net discovery algorithms. Trans. Petri Nets Other Model. Concurr. **2**, 225–242 (2009)
12. Dumas, M., García-Bañuelos, L.: Process mining reloaded: Event structures as a unified representation of process models and event logs. In: Application and Theory of Petri Nets and Concurrency - 36th International Conference, Proceedings. LNCS, vol. 9115, pp. 33–48. Springer (2015)
13. Favre, C., Fahland, D., Völzer, H.: The relationship between workflow graphs and free-choice workflow nets. Inf. Syst. **47**, 197–219 (2015)
14. Kourani, H., Park, G., van der Aalst, W.M.P.: Translating workflow nets into the partially ordered workflow language. In: Amparore, E., Mikulski, Ł. (eds.) Application and Theory of Petri Nets and Concurrency. pp. 242–264. Springer Nature Switzerland, Cham (2025)
15. Kourani, H., Schuster, D., van der Aalst, W.M.P.: 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. pp. 89–96. IEEE (2023)
16. Kourani, H., van Zelst, S.J.: POWL: partially ordered workflow language. In: Business Process Management - 21st International Conference, BPM 2023, Utrecht, The Netherlands, September 11-15, 2023, Proceedings. LNCS, vol. 14159, pp. 92–108. Springer (2023)
17. Kourani, H., van Zelst, S.J., Schuster, D., van der Aalst, W.M.P.: Discovering partially ordered workflow models. Inf. Syst. **128**, 102493 (2025)
18. Leemans, S.J.J.: Robust Process Mining with Guarantees - Process Discovery, Conformance Checking and Enhancement, LNBIP, vol. 440. Springer (2022)
19. Leemans, S.J.J., Poppe, E., Wynn, M.T.: Directly follows-based process mining: Exploration & a case study. In: ICPM. pp. 25–32. IEEE (2019)
20. Leemans, S.J.J., Tax, N., ter Hofstede, A.H.M.: Indulpet miner: Combining discovery algorithms. In: OTM Conferences (1). Lecture Notes in Computer Science, vol. 11229, pp. 97–115. Springer (2018)
21. von Rosing, M., White, S., Cummins, F., de Man, H.: Business process model and notation - BPMN. In: The Complete Business Process Handbook: Body of Knowledge from Process Modeling to BPM, Volume I, pp. 429–453. Morgan Kaufmann/Elsevier, Massachusetts, USA (2015)
22. Slaats, T., Schunselaar, D.M.M., Maggi, F.M., Reijers, H.A.: The semantics of hybrid process models. In: OTM Conferences. Lecture Notes in Computer Science, vol. 10033, pp. 531–551 (2016)
23. van Hee, K.M., Sidorova, N., van der Werf, J.M.E.M.: Business process modeling using Petri nets. Trans. Petri Nets Other Model. Concurr. **7**, 116–161 (2013)