Kourani *et al. Process Science* (2026) 3:14
https://doi.org/10.1007/s44311-026-00046-8

Process Science

RESEARCH Open Access

A discovery technique for expressive yet sound process models

Humam Kourani[1,2*], Gyunam Park[1] and Wil M. P. van der Aalst[1,2]

*Correspondence:
Humam Kourani
humam.kourani@fit.fraunhofer.de
[1]Fraunhofer Institute for Applied Information Technology FIT, Schloss Birlinghoven, 53757 Sankt Augustin, Germany
[2]RWTH Aachen University, Ahornstraße 55, 52074 Aachen, Germany

Abstract
Process discovery enables organizations to analyze and improve their operations by automatically deriving process models from event logs. While the Inductive Mining framework is widely adopted for ensuring model soundness, its strict block-structured nature often fails to capture the true complexity of real-world control flows. Recent advancements, such as the Partially Ordered Workflow Language (POWL), have relaxed these constraints for concurrency, yet a significant gap remains in effectively modeling complex decision logic and unstructured loops. We bridge this gap by introducing *POWL 2.0*, an extended modeling language that integrates *choice graphs* to represent non-block-structured decisions and cyclic flows within a hierarchical framework. In this paper, we present a robust inductive discovery algorithm that leverages POWL 2.0, and we explore several mining strategies for choice graphs. These strategies range in strictness to resolve the ambiguity between genuine loops and hidden concurrency. Our experimental evaluation demonstrates that the proposed approach captures complex behaviors without compromising the scalability and quality guarantees of the Inductive Mining framework.

Keywords Process discovery, Process modeling, Inductive Miner, Unstructured loops

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 (Leemans 2022) 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, it enables a recursive approach to discovery, ensuring efficient

Springer

processing of large event logs. Furthermore, the generated hierarchical process models can be easily transformed into standard notations, such as BPMN (von Rosing et al. 2015) and Petri nets (Desel et al., 2004), facilitating their use in a variety of analysis and implementation contexts.

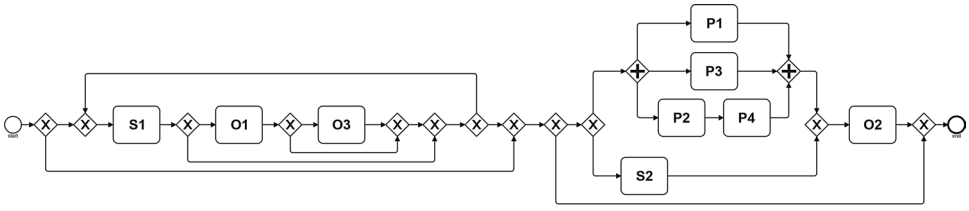
The original Inductive Miner uses *process trees* for intermediate representation. A process tree corresponds to a mathematical tree where leaves represent activities and internal nodes are control-flow operators that model the relationships between children. The base variant of the Inductive Miner supports four control-flow operators for modeling common process constructs: *sequence*, *exclusive choice (XOR)*, *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 no splits and joins can overlap or interleave in a non-hierarchical manner.

In (Kourani and van Zelst 2023), the Inductive Miner was extended by introducing the *Partially Ordered Workflow Language* (POWL) as a more expressive intermediate process representation than process trees. By using partial orders instead of the process tree concurrency operator, this extension enabled the discovery algorithm to more accurately capture non-block-structured concurrent behavior. Our previous contribution (Kourani et al. 2025a) further extended the discovery capabilities toward unstructured decision logic. This was achieved by introducing *choice graphs* in POWL 2.0, which replace the XOR operator with a more expressive construct. Choice graphs allow the discovery algorithm to represent complex, non-block-structured choices within the hierarchical discovery framework, while preserving desirable properties such as soundness.

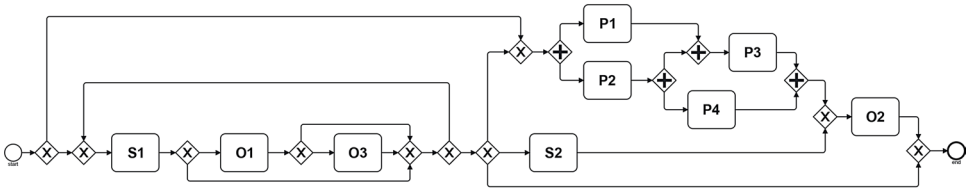
The introduction of choice graphs in (Kourani et al. 2025a) represented a significant step forward, allowing for non-block-structured decision flows to be modeled accurately within a sound structure. However, the discovery approach presented in that work was restricted to detecting *acyclic* choice graphs. Choice graphs are mined based on the *Directly-Follows Graph (DFG)*, which is a graphical representation of direct succession dependencies present in the event log. The restriction of acyclicity was a deliberate, conservative design choice intended as a proof of concept. The underlying rationale was to avoid the common pitfall of DFG-based mining: misinterpreting concurrency as cycles. By enforcing strict acyclicity within the choice graph, the algorithm delegates all cyclic behavior to the dedicated, block-structured process tree loop operator. While this strategy ensures that concurrency is rarely mistaken for a loop, it artificially limits the expressiveness of choice graphs. The standard loop operator is designed for binary "do-redo" structures (execute a, then optionally do b and repeat a). It cannot naturally capture complex, unstructured cycles (such as "state-machine" loops that jump back to arbitrary earlier points in the process), which choice graphs are theoretically capable of representing.

In this paper, we bridge this gap by extending the Inductive Mining framework to discover cyclic choice graphs. To illustrate the evolution of modeling capabilities with choice graphs, consider the comparison in Fig. 1. The first model (Fig. 1a), generated by the base Inductive Miner, adheres to a strict block structure for both exclusive choices and loops, requiring every XOR-split to be paired with a matching join to form well-nested blocks. The second model (Fig. 1b), derived from our previous acyclic approach (Kourani et al. 2025a), relaxes this constraint to allow flexible forward jumps within a decision flow; however, it still restricts cycles to rigid, block-structured loops containing

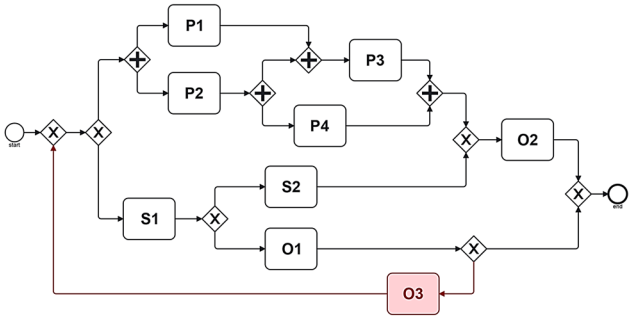
distinct “do” and “redo” parts. Our proposed approach overcomes this limitation to produce the third model (Fig. 1c): it supports fully unstructured decision logic where the control flow can loop back to arbitrary earlier stages.



(a) Base Inductive Miner: Constrained to block-structured splits and joins. This model overgeneralizes the observed behavior, as it fails to precisely capture unstructured concurrency, choices, and cycles (precision score: 0.46). For example, it allows production to be executed before gathering the required materials.



(b) POWL Miner with Acyclic Choice Graphs (previous work): Supports unstructured concurrency and unstructured “forward” choices but enforces block-structured loops. This model remains imprecise because cyclic decision behavior involving canceling orders and placing new ones cannot be represented (precision score: 0.58). For example, the model allows production and shipment to occur after an order is canceled, without placing a new order.



(c) POWL Miner with Cyclic Choice Graphs (proposed): Supports fully unstructured decision logic, including complex cyclic behavior. This model accurately captures the intended process behavior (precision score: 1.00).

Abb.	Original Activity Label
S1	Check Stock Availability
S2	Collect Items from Stock
P1	Gather Production Materials
P2	Schedule Production
P3	Execute Production
P4	Notify Customer
O1	Cancel Order
O2	Ship Order
O3	Place New Order

(d) Mapping of abbreviations used in the figures to the original activity labels.

Fig. 1 BPMN models discovered for an example order fulfillment process

Enabling cyclic discovery introduces a significant algorithmic challenge: the ambiguity between genuine decision loops and hidden concurrency. In the DFG derived from an event log, a cycle between two activities a and b can have two very different interpretations. It could represent a short decision loop (execute a, then b, then a again), or it could represent concurrency (in some traces a follows b, in others b follows a). A naive approach that interprets every cycle in the DFG as a decision loop within a choice graph would destroy the ability to discover concurrency, effectively linearizing parallel behavior. Conversely, the previous approach avoided this by strictly forbidding cycles in choice graphs, thereby delegating the resolution to recursive steps, but at the cost of failing to capture unstructured loops. The core challenge, therefore, lies in finding a strategy that allows for the discovery of legitimate, unstructured decision cycles while effectively filtering out the “false” cycles that arise from concurrency.

To effectively address the algorithmic ambiguity between legitimate decision loops and cycles induced by hidden concurrency, we introduce a hierarchy of mining strategies ranging from permissive to strict. Depending on the chosen level of strictness, cyclic behavior in the DFG is handled differently:

- **Level 0 (Highly Permissive):** The algorithm accepts all cycles in the DFG (e.g., $a \leftrightarrow b$), separating all activities into distinct nodes of a choice graph. This interprets the behavior entirely as a decision loop, which can act as a generic fallback but risks linearizing actual concurrency.
- **Level 1:** The algorithm explicitly targets direct cycles in the DFG (i.e., $a \leftrightarrow b$). To avoid masking concurrency, it merges a and b into the same block, intentionally delegating their structural resolution (concurrency vs. strict binary loop) to the next recursive step. However, it still allows longer, indirect cycles (e.g., $a \rightarrow c \rightarrow d \rightarrow a$) to form unstructured loops directly within the choice graph.
- **Level 2:** The algorithm broadens the restriction to handle noisy or incomplete event logs. It merges blocks of activities if there is any bidirectional flow between them, even if the flow involves different activities within those blocks. For example, consider a DFG that contains the connections $a \leftrightarrow b$, $b \rightarrow c$, and $c \rightarrow a$. First, a and b are merged due to the direct cycle; then, because b is linked to c and c is linked back to a, c is also merged into the same block. The structural resolution for the entire set $\{a, b, c\}$ is then delegated to the next recursive step.
- **Level 3 (Acyclic):** The algorithm forbids all cycles within the choice graph. Any activities involved in a cycle, even if only indirectly via a path like $a \rightarrow b \rightarrow c \rightarrow a$, are merged into a single block, forcing all cyclic behavior to be handled by the traditional, block-structured loop operator in later recursive steps.

We rigorously evaluate these strategies to investigate which approach yields the optimal results in terms of model precision and fitness. Furthermore, we integrate a new frequency-based filtering mechanism into our discovery approach, and we leverage choice graphs to introduce a new *DFG-based fall-through mechanism*.

This paper is an extended version of the work from (Kourani et al. 2025a). While the preliminary work introduced the foundational concept of choice graphs and a discovery algorithm limited to acyclic structures, this extension significantly broadens the scope to handle cyclic behavior. The specific contributions of this paper, extending the work in (Kourani et al. 2025a), are:

(1) **Generalized Validity Definition & Algorithm:** A revised formalism for valid choice graph cuts that eliminates the acyclicity constraint, accompanied by a refined discovery algorithm and a formal proof of fitness preservation.
(2) **Strategies for Cyclic Choice Graph Mining:** A definition of four cut detection strategies that differ in strictness, allowing us to investigate the optimal balance between capturing unstructured loops and avoiding the misinterpretation of hidden concurrency.
(3) **DFG-based Fall-Through:** The introduction of a novel fall-through mechanism based on the most general choice graph strategy (Level 0), which replaces the traditional *flower model* fall-through used in previous approaches. The aim is to preserve direct-succession constraints even when no other structure is found.

(4) **Connectivity-Preserving Filtering:** A noise filtering mechanism that simplifies the DFG based on edge frequencies while employing a repair strategy to ensure all activities remain reachable on a path between start and end nodes, preventing the premature loss of activities due to the filtering.
(5) **Extended Evaluation:** An experimental comparison of choice graph mining strategies on 17 real-life event logs, identifying the approach that yields the optimal trade-off between expressiveness and precision.

The remainder of this paper is structured as follows. We first discuss related work in Sect. "Related Work". Then, we motivate our contribution through an example in Sect. "Motivating Example". We introduce necessary preliminaries in Sect. "Preliminaries", and we formally define POWL 2.0 and discuss its quality guarantees in Sect. "POWL 2.0: Extending POWL with Choice Graphs". In Sect. "Discovery of POWL Models with Cyclic Choice Graphs", we present the extended discovery algorithm, the hierarchy of mining strategies, and the proposed filtering mechanism. Section "Evaluation" presents the experimental evaluation, and Sect. "Conclusion" concludes the paper.

Related work

A wide range of process discovery techniques has been developed. For a comprehensive overview, we refer to (Augusto et al. 2019a) and (van Dongen Bf et al. 2009). Approaches such as the Inductive Miner (Leemans 2022) and the Evolutionary Tree Miner (Buijs et al. 2012) generate block-structured process models, striking a balance between model quality and simplicity. In (Kourani and van Zelst 2023), 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 (Kourani et al. 2023) and (Kourani et al. 2025b).

Several approaches discover process models directly in more flexible notations, such as Petri nets and BPMN. For instance, the Split Miner (Augusto et al. 2019a) 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., (Bergenthum et al. 2007) and (Carmona et al. 2008) 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.

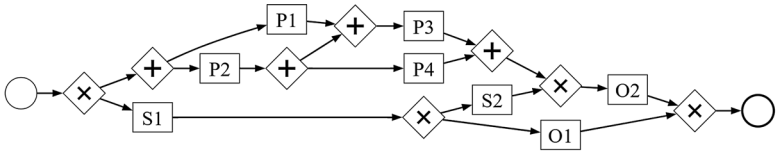
Approaches for directly translating DFGs into Petri nets have been explored, e.g., (Leemans et al. 2019). In (Dumas and García-Bañuelos 2015) and (van Beest et al. 2015), methods are proposed for discovering prime event structures that integrate conflict relations into partially ordered graphs. These 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. For instance, the Indulpet Miner (Leemans et al. 2018) integrates multiple distinct discovery algorithms within the Inductive Miner framework, delegating the discovery of sub-processes to external techniques when standard tree cuts fail. Similarly, Fusion-Based Process Discovery (Dahari et al. 2018) combines the outputs of several different discovery algorithms to synthesize

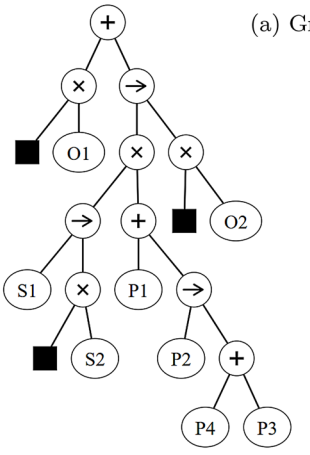
a unified, higher-quality model. Furthermore, Petri nets are extended with additional causal edges in (van der Aalst et al., 2023), while the approach in (Slaats et al. 2016) combines imperative process models with declarative constraints.

Motivating example

Let us 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 Fig. 2a.



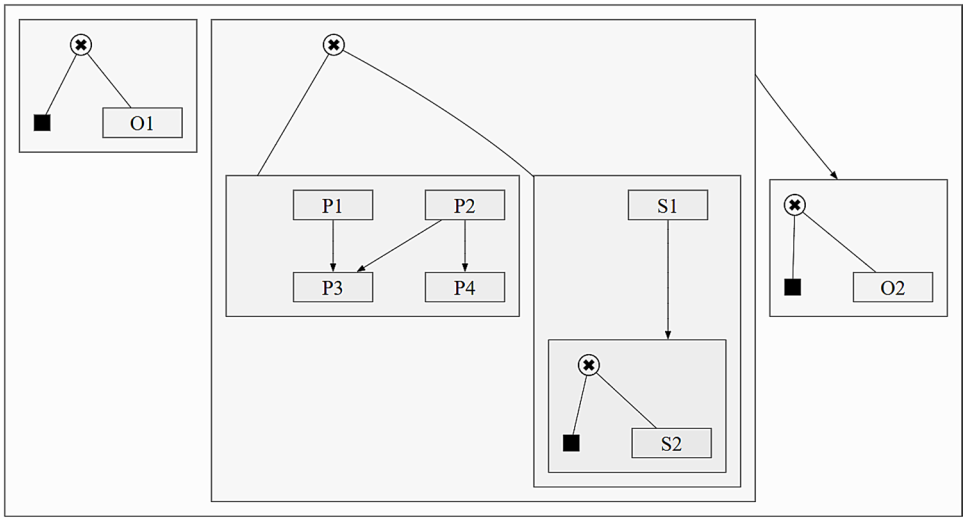
(a) Ground truth BPMN model.



(b) Process tree discovered with the base Inductive Miner (Leemans 2022).

Abb.	Original Activity Label
S1	Check Stock Availability
S2	Collect Items from Stock
P1	Gather Production Materials
P2	Schedule Production
P3	Execute Production
P4	Notify Customer
O1	Cancel Order
O2	Ship Order

(c) Mapping of abbreviations used in the figures to the original activity labels.



(d) POWL model discovered with the POWL Inductive Miner (Kourani et al. 2023).

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

Unstructured concurrency

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 Fig. 2b 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.

POWL uses partial orders for modeling non-block-structured concurrency. A partial order specifies a set of children (i.e., sub-processes) that must all be executed, but only imposes a partial ordering constraint on their execution, allowing for flexible interleavings.

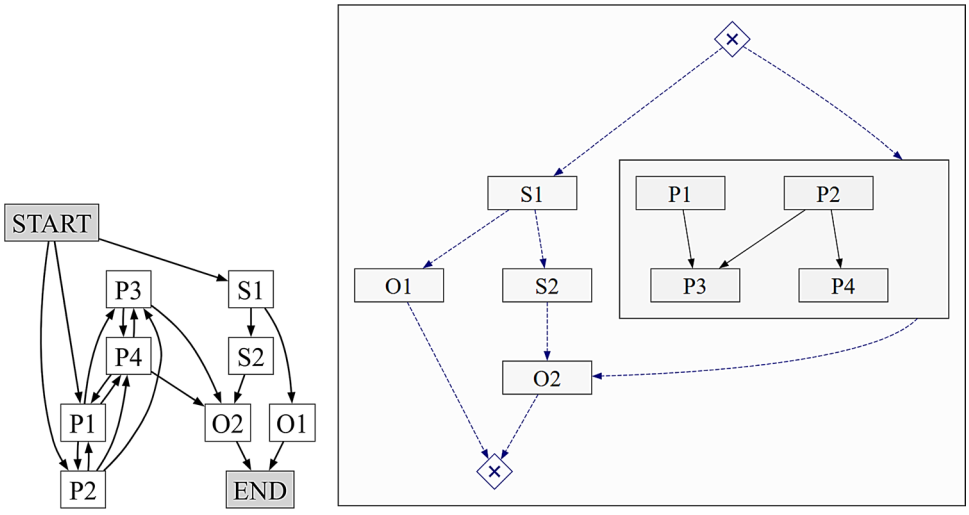
Figure 2d shows the POWL model discovered using the POWL Inductive Miner for our running example. Unlike the process tree, the discovered POWL model can precisely describe the dependencies between the four activities involved in the production sub-process. These dependencies are correctly described using a compact partial order with three ordering edges: Gather Production Materials → Execute Production, Schedule Production → Execute Production, and Schedule Production → Notify Customer.

Unstructured choice

While POWL eliminates the block-structure requirement 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 Fig. 2a, 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. Fig. 2b) and POWL model (cf. Fig. 2d) incorrectly allow the order to be canceled and shipped simultaneously.

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 Fig. 3a 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



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

Fig. 3 Process models for a retailer's order fulfillment process

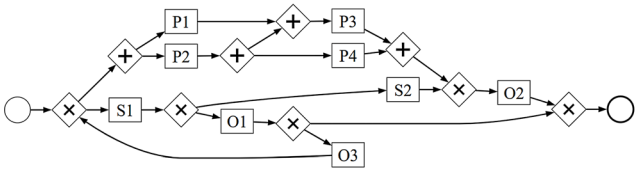
from concurrency or repeated behaviors. POWL provides dedicated, semantically precise operators for these structures (partial orders and loops). Recognizing the complementary nature of these two approaches provides the central motivation for POWL 2.0. We integrate a graph-based representation similar to DFGs (choice graphs) for handling choices directly into the hierarchical structure of POWL. This integration allows for leveraging the strengths of both modeling paradigms while mitigating their individual limitations, ultimately leading to more expressive yet sound process models.

Figure 3b 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.

Unstructured cycles

Let us extend our running example to include cyclic behavior. Suppose we refine the behavior of the "In-Stock" path: after the "Cancel Order" activity, the customer is presented with a choice. They can either end the process or use the store credit to "Place New Order". Crucially, choosing to place a new order loops the process back to the initial decision point, allowing the customer to again choose between the "In-Stock" and "Production" paths for the new order. The cyclic POWL 2.0 model shown in Fig. 4d accurately captures the intended process behavior with the described unstructured cycle; its equivalent BPMN representation is shown in Fig. 4a.

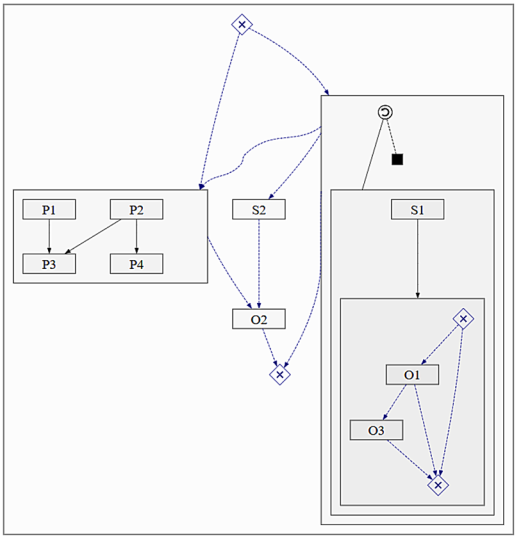
When the acyclic discovery approach from (Kourani et al. 2025a) encounters this behavior, it detects a cycle in the DFG involving the activities "Cancel Order", "Place New



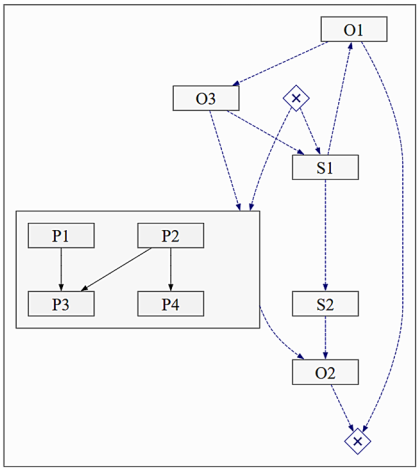
(a) Ground truth BPMN model.

Abb.	Original Activity Label
S1	Check Stock Availability
S2	Collect Items from Stock
P1	Gather Production Materials
P2	Schedule Production
P3	Execute Production
P4	Notify Customer
O1	Cancel Order
O2	Ship Order
O3	Place New Order

(b) Mapping of abbreviations used in the figures to the original activity labels.



(c) Imprecise model discovered by the acyclic approach (Kourani et al. 2025a). The BPMN model in Figure 1b models the same behavior.



(d) Capturing the intended behavior using a cyclic choice graph. The BPMN model in Figure 4a models the same behavior.

Fig. 4 POWL 2.0 models for the refined cyclic order fulfillment process

Order", and "Check Stock Availability". Because it enforces strict acyclicity within choice graphs, it merges all nodes involved in this cycle into a single strongly connected component. The algorithm then attempts to handle this merged block recursively. Lacking a clear structure, the miner resorts to fall-through mechanisms, which results in a highly imprecise self-loop. The resulting model, visualized in Fig. 4c, is semantically flawed: for instance, it allows "Check Stock Availability" to be directly followed by the production subprocess, and it permits the sequence ("Check Stock Availability" $\rightarrow$ "Cancel Order") to be repeated multiple times without executing "Place New Order".

This example highlights the need for a more powerful discovery technique that is not bound by the strict acyclicity requirement. To capture the intended cyclic behavior and discover the model shown in Fig. 4d instead of the imprecise one in Fig. 4c, the mining algorithm must be capable of identifying and constructing cyclic choice graphs directly.

Preliminaries

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

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. Formally, a multi-set over a set X is a function $M : X \to \{0, 1, 2, \dots\}$, where $M(x)$ denotes the multiplicity (number of occurrences) of x in M. We define the membership operator such that $x \in M$ if and only if $x \in X \ \wedge \ M(x) > 0$. A multi-set is represented as $M = [{x_1}^{c_1}, \dots, {x_n}^{c_n}]$, where $x_1, \dots, x_n \in M$ are the elements of M and $c_i = M(x_i)$ 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 $P = \{X_1, \dots, X_n\}$ such that $X = X_1 \cup \dots \cup X_n$, $X_i \neq \emptyset$ for all $1 \leq i \leq n$, 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., $P_x \in \{X_1, \dots, X_n\}$ 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 $R \subseteq X \times X$ be a binary relation over a set X. The *transitive closure* of R, denoted R^+, is the smallest binary relation over X that contains R ($R \subseteq R^+ \subseteq X \times X$) and is transitive (if $(x_1, x_2) \in R^+$ and $(x_2, x_3) \in R^+$, then $(x_1, x_3) \in R^+$).

A *strict partial order* (or simply *partial order*) over a set X is a binary relation $\prec \subseteq X \times X$ that satisfies transitivity and irreflexivity ($(x, x) \notin \prec$ for all $x \in X$). These two properties together imply *asymmetry* (if $(x_1, x_2) \in \prec$, then $(x_2, x_1) \notin \prec$). For a partial order $\prec \subseteq X \times X$, we write $x_1 \prec x_2$ to denote $(x_1, x_2) \in \prec$ and $x_1 \not\prec x_2$ to indicate $(x_1, x_2) \notin \prec$.

Event log

We define Σ as the universe of activity labels to establish the set of observable tasks that can occur in a process. 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.

Let $A \subseteq \Sigma$ be a finite set of activity labels. An *event log* over A is defined as a multi-set of sequences of activities, i.e., $L \in \mathcal{B}(A^*)$. 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 multi-set of *start activities* in L.
- $L_{\square} = [\sigma(|\ \sigma\ |) \mid \sigma \in L]$ is the multi-set of *end activities* in L.
- $L_{\mapsto} = [(\sigma(i), \sigma(i+1)) \mid \sigma \in L \ \wedge \ 1 \leq i < |\ \sigma\ |]$ is the multi-set of *direct succession pairs* in L.
- $\mapsto \subseteq \Sigma_L \times \Sigma_L$ is the *Directly-Follows Graph (DFG) relation*, defined as follows:

$$a \mapsto b \quad \text{iff} \quad (a, b) \in L_{\mapsto}.$$

- $\rightsquigarrow \subseteq \Sigma_L \times \Sigma_L$ is the *Eventually-Follows Graph (EFG) relation*, defined as follows:

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

For example, consider the event log $L_1 = [\langle a, b \rangle^3, \langle b, c, d \rangle^2]$, which consists of five traces. The corresponding sets are $\Sigma_{L_1} = \{a, b, c, d\}$, $L_1 \triangleright = \{a, b\}$, and $L_1 \square = \{b, d\}$. The DFG of L_1 is $\mapsto = \{(a, b), (b, c), (c, d)\}$, and the EFG is $\rightsquigarrow = \{(a, b), (b, c), (b, d), (c, d)\}$. Note that while (a, c) is in the transitive closure of the DFG (since $a \mapsto b$ and $b \mapsto c$), it is not in the EFG because a is never eventually followed by c within the same trace.

POWL 2.0: extending POWL with choice graphs

A POWL model (Kourani and van Zelst 2023) 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 sub-processes, whereas $\circlearrowleft$ captures cyclic behavior between two sub-processes: 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 sub-processes 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 a 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 $\mathcal{G}(X)$ 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 $G = (N, E) \in \mathcal{G}(X)$ over a set X, we define the set of all paths in G as follows:

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

With choice graphs established, we define *POWL 2.0* (Kourani et al. 2025a), 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 and cycles between sub-models are now represented using choice graphs instead of the process tree operators $\times$ and $\circlearrowleft$.

Notation

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 $i, j \in \{1, \ldots, n\}$.

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 $\psi_1, \ldots, \psi_n$ be $n \geq 2$ POWL models.
  - A choice graph $G \in \mathcal{G}(\{\psi_1, \ldots, \psi_n\})$ is a POWL model.
  - For a partial order $\prec \in \mathcal{O}^n$, $\prec(\psi_1, \ldots, \psi_n)$ is a POWL model.

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.

Similar to the original POWL framework (Kourani and van Zelst 2023), 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 sub-models along a valid path in the choice graph.

Notation

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 $\sqcup\!\sqcup_{\prec}(\sigma_1, \sigma_2, \sigma_3) = \{\langle a,b,c,d,e \rangle, \langle a,b,d,c,e \rangle, \langle a,b,d,e,c \rangle\}$.

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

- $\mathcal{L}(a) = \{\langle a \rangle\}$ for $a \in \Sigma$.
- $\mathcal{L}(\tau) = \{\langle\rangle\}$.
- Let $\psi_1, \ldots, \psi_n$ be $n \geq 2$ POWL models.
  - For $\prec \in \mathcal{O}^n$, $\mathcal{L}(\prec(\psi_1, \ldots, \psi_n)) = \{\sigma \in \sqcup\!\sqcup_{\prec}(\sigma_1, \ldots, \sigma_n) \mid \forall_{1 \leq i \leq n}\ \sigma_i \in \mathcal{L}(\psi_i)\}$.
  - For $G \in \mathcal{G}(\{\psi_1, \ldots, \psi_n\})$, $\mathcal{L}(G) = \bigcup_{\langle \psi'_1, \ldots, \psi'_k \rangle \in \vec{G}} \mathcal{L}(\psi'_1) \cdot \ldots \cdot \mathcal{L}(\psi'_k)$.

Note that POWL 2.0 also supports the standard process tree operators. These operators are omitted in Definition 2 for compactness, since they can all be translated into semantically equivalent POWL 2.0 constructs. A sequence or concurrency operator corresponds to a partial order, while an XOR or loop operator corresponds to a choice graph.

Definition 4 [Mapping Process Trees to POWL 2.0]. Let $\psi_1, \ldots, \psi_n$ be $n \geq 2$ POWL models. The behavior of standard process tree operators $(\rightarrow, \times, +, \circlearrowleft)$ can be modeled using POWL 2.0 constructs as follows:

- **Concurrency:** $+(\psi_1, \ldots, \psi_n)$ maps to the empty partial order $\prec(\psi_1, \ldots, \psi_n)$ where $\prec = \emptyset$.
- **Sequence:** $\rightarrow(\psi_1, \ldots, \psi_n)$ maps to the totally ordered partial order $\prec(\psi_1, \ldots, \psi_n)$ where $\prec = \{(i,j) \mid 1 \leq i < j \leq n\}$.[1]

[1] Alternatively, a sequence can be represented as a linear choice graph $G \in \mathcal{G}(\{\psi_1, \ldots, \psi_n\})$ where edges form a path $\triangleright \rightarrow \psi_1 \rightarrow \cdots \rightarrow \psi_n \rightarrow \square$.

- **Exclusive Choice:** $\times(\psi_1, \ldots, \psi_n)$ maps to a choice graph $G = (N, E) \in \mathcal{G}(\{\psi_1, \ldots, \psi_n\})$ where:

$$E = \{(\triangleright, \psi_i) \mid 1 \leq i \leq n\} \cup \{(\psi_i, \square) \mid 1 \leq i \leq n\}$$

- **Binary Loop:** $\circlearrowleft(\psi_1, \psi_2)$ maps to a choice graph $G = (N, E) \in \mathcal{G}(\{\psi_1, \psi_2\})$ where:

$$E = \{(\triangleright, \psi_1), (\psi_1, \square), (\psi_1, \psi_2), (\psi_2, \psi_1)\}$$

While POWL 2.0 provides significant flexibility, the combination of choice graphs and partial orders means the two constructs are not strictly orthogonal. As highlighted in Definition 4, pure sequential behavior can be represented either as a totally ordered partial order or as a linear choice graph. In traditional modeling language design, such overlap is typically avoided to prevent ambiguity. However, POWL 2.0 is deliberately designed as an *intermediate representation* to support the recursive divide-and-conquer strategy of the Inductive Miner. This structural flexibility allows the discovery algorithm to capture behavioral dependencies using whichever construct best fits the local log characteristics at a given recursive step. Most importantly, this non-orthogonality does not impact the final end-user model. When a POWL 2.0 model is translated into standard target notations, such as Petri nets or BPMN, these internal representational deviations disappear; a sequence discovered as a partial order and a sequence discovered within a choice graph will map to the same standard sequential structures in the target language. Furthermore, internal structural reduction rules can be applied as a post-processing step to simplify and canonicalize the POWL 2.0 model itself post to discovery.

Discussion: conversion to WF-Nets, guarantees, limitations

POWL 2.0 models can be systematically converted into equivalent Workflow Nets (WF-nets). A WF-net 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 (Kourani et al. 2025b), 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 (Kourani et al. 2025b) can be easily constructed.

A *workflow graph* is a directed graph whose vertices are either activities or BPMN-style split/join gateways. As illustrated in Fig. 5, every POWL 2.0 construct can be interpreted as a single-entry-single-exit (SESE) fragment of such a graph: partial orders are mapped to concurrent regions using parallel gateways, while choice graphs are mapped to decision paths connected by exclusive gateways. 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 be translated into equivalent *free-choice* workflow nets (Favre et al. 2015). A Petri net is free-choice if for every two transitions that share an input place, that place is their unique input place (Desel and Esparza 1995). This structural property ensures that choices and

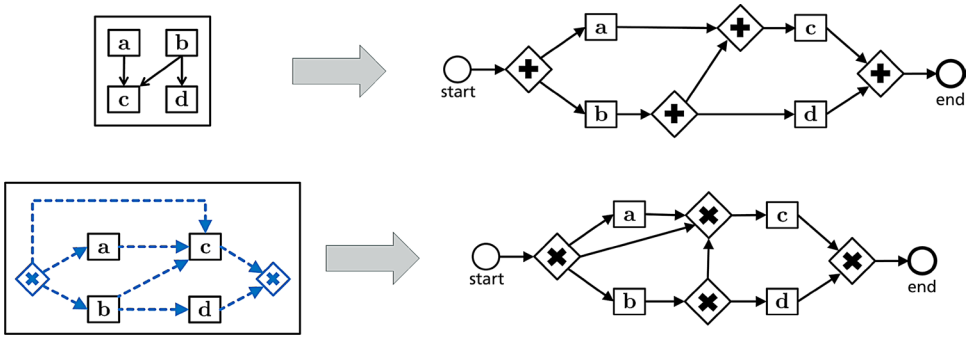


Fig. 5 Translation of POWL 2.0 constructs into BPMN

synchronizations do not intertwine, which aligns perfectly with POWL 2.0's strict separation of choice graphs and partial orders.

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 generally, the expressiveness of POWL 2.0 covers a broader subclass of free-choice WF-nets, which is termed *separable WF-nets* (Kourani et al. 2026).

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.

Discovery of POWL models with cyclic choice graphs

This section presents an Inductive Mining algorithm for discovering POWL 2.0 models. We begin by providing an overview of the recursive framework in Sect. "Algorithm Overview". In Sect. "Choice Graph Cut", we formalize the concept of choice graph cuts. Sect. "Strategies for Mining for Choice Graphs" details different choice graph mining strategies ranging from permissive to strict, and Sect. "Connectivity-Preserving Noise Filtering" introduces a connectivity-preserving filtering mechanism to handle noise.

Algorithm overview

The *Inductive Mining* framework (Leemans 2022) discovers process models through a recursive "divide-and-conquer" strategy. The core mechanism involves identifying a *cut*, which involves detecting a behavioral pattern in the event log and partitioning the activities accordingly. Once a cut is identified, the log is projected onto the resulting partitions, and the algorithm recurses on the sub-logs. If no structural cut is found, the algorithm resorts to *fall-through* strategies, which are special cuts designed to decompose the log when clear structure is absent. If the algorithm also fails to find a fall-through cut, it returns a *flower model* (allowing any behavior), which preserves fitness but sacrifices precision.

The standard Inductive Miner discovers *Process Trees*, limiting cuts to block-structured operators ($\times$, $\rightarrow$, $+$, and $\circlearrowleft$). Our extended approach, visualized in Fig. 6, adapts this framework to discover POWL 2.0 models. It aims to detect advanced POWL 2.0 cuts while retaining specific standard cuts to handle special cases. The procedure follows these steps:

Step 1: Base cases

Before attempting to find a cut, the algorithm starts by handling the following two base cases:

- **Empty Traces:** The algorithm checks if the current log L contains empty traces (i.e., $\langle\rangle \in L$). If present, this indicates that the process can be skipped. To model this, the algorithm effectively splits the process into a skip structure: $\times(\tau, \psi)$[2], where ψ is the result of the recursive application of the algorithm on $L' = L \setminus \{\langle\rangle\}$.
- **Single Activity:** If the log consists of a single activity a, the recursion terminates, and the algorithm returns the leaf node a.

Step 2: Mining for structural cuts

The algorithm attempts to detect strong structural patterns in the log. This phase combines the advanced expressiveness of POWL 2.0 with the robustness of standard process tree operators.

- **POWL 2.0 Cuts:** We first attempt to identify non-block-structured patterns using POWL 2.0 operators:
  - *Choice Graph Cut:* This new cut (c.f. Sect. "Choice Graph Cut") partitions the log based on the DFG structure to capture complex decision logic. In Sect. "Strategies for Mining for Choice Graphs", we will present different strategies that can be used to mine for choice graph cuts.
  - *Partial Order Cut:* This cut, adapted from (Kourani et al. 2023), aims at constructing a partial order $\prec$ where activity sets are ordered if sequential dependencies between them exist.

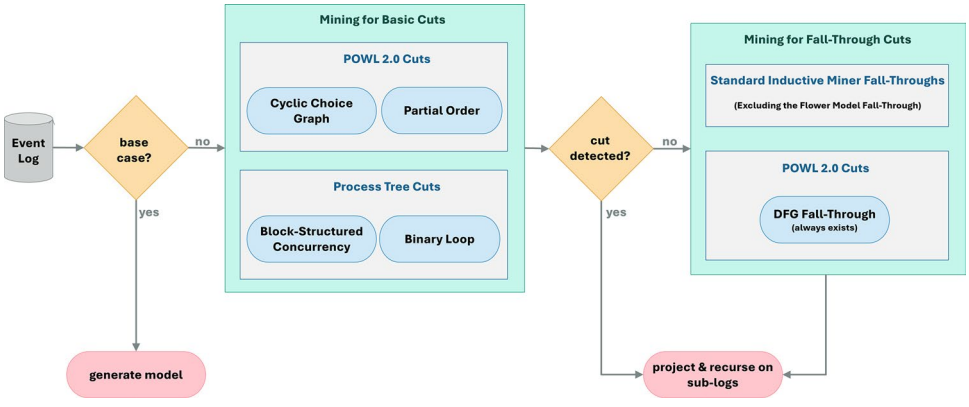


Fig. 6 Overview of the POWL 2.0 Inductive Miner

[2] The skip structure can also be represented in choice graph by adding a direct edge from the start node to the end node.

- **Process Tree Cuts:** We retain standard block-structured cuts to handle specific scenarios where generalized POWL cuts might be overly conservative. In trivial cases, such as full concurrency or simple binary loops ($a \rightarrow b \rightarrow a$), the structural signals in the DFG or Eventually-Follows Graph (EFG) may appear similar. To prevent misinterpreting concurrency as loops (or vice versa), our POWL cut detection mechanisms are designed to be cautious: they group such tightly coupled activities into the same part rather than forcing a potentially incorrect split. Consequently, we rely on the standard process tree concurrency (+) and loop ($\circlearrowleft$) cuts to explicitly identify these simple structures in the next iterations of the recursion, ensuring they are not overlooked or misinterpreted.

Step 3: Fall-through cuts

If no structural cut is found, the algorithm applies a series of standard fall-through strategies (Leemans 2022). These special cuts attempt to identify less obvious partitions, enabling the recursion to proceed. For example, one of the fall-throughs places an activity that appears in every trace in concurrency to the remaining activities.

In the original Inductive Miner, if all fall-through attempts fail, the algorithm returns a flower model ($\Sigma_L{}^*$ behavior). This guarantees termination but results in a complete loss of precision. In our approach, we replace the flower model fall-through with the *DFG fall-through*, which will be detailed in Sect. "Level 0: The Atomic Cut (DFG Fall-Through)". This cut is the most general type of choice graph cuts. It interprets the DFG of the current log as a choice graph, where every activity is a node. Since a DFG can always be constructed for a non-empty log, this cut *always exists*, guaranteeing the algorithm's termination. Unlike the flower model, the DFG fall-through preserves all direct-succession constraints observed in the log, serving as a more precise "last resort" compared to the flower model fall-through.

Choice graph cut

This section formalizes the discovery of choice graphs within the Inductive Mining framework. We first define the notion of a choice graph cut together with its validity conditions (Sect. "Valid Choice Graph Cut"). We then describe how event logs are projected onto the parts of a cut in the presence of cyclic behavior (Sect. "Projection of Choice Graph Cuts") and show that valid choice graph cuts preserve perfect fitness (Sect. "Perfect Fitness Guarantee").

Valid choice graph cut

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

Not every choice graph cut provides a valid representation of an event log. Intuitively, for a fixed partition of activities, the event log already dictates which inter-part dependencies must be present: if the log contains a direct succession from some activity in A_i

to some activity in A_j, then the choice graph must contain the edge $A_i \to A_j$. Similarly, parts containing start (resp. end) activities must be connected to the artificial start (resp. end) node. This motivates defining the choice graph *induced* by a partition.

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

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

Induced choice graph Fix an event log L and a partition $A = \{A_1, \ldots, A_n\}$ of Σ_L with $n \geq 2$. We define the *choice graph constructor* (induced by L) as a function

$$\diamondsuit_L(A) \ = \ (N, E),$$

where $N = A \cup \{\triangleright, \square\}$ and the edge set E is constructed as follows:

$$\begin{aligned} E \ = \ & \{(A_i, A_j) \mid A_i, A_j \in A,\ A_i \neq A_j,\ A_i \mapsto A_j\} \\ & \cup\ \{(\triangleright, A_i) \mid A_i \in A,\ A_i \cap L_\triangleright \neq \emptyset\} \\ & \cup\ \{(A_i, \square) \mid A_i \in A,\ A_i \cap L_\square \neq \emptyset\} \\ & \cup\ \{(\triangleright, \square) \mid \langle\rangle \in L\}. \end{aligned}$$

That is, $\diamondsuit_L(A)$ contains exactly the inter-part edges implied by the DFG, connects all start/end parts to the artificial start/end nodes, and adds a direct start-to-end edge if empty traces occur in the log.

Definition 6 [Valid Choice Graph Cut] Let $C = (A, G)$ be a choice graph cut over an event log L. We call C *valid* iff $G = \diamondsuit_L(A)$.

Definition 6 uniquely defines a choice graph over any given partition ($G = \diamondsuit_L(A)$). In other words, in order to detect a choice graph cut, we need to generate a partition of activities that contains at least two parts. Our strategies for the detection of choice graph cuts add requirements on the partitioning of the activities, mainly enforcing certain activities to be placed within the same part. The aim is to enable the discovery of more precise constructs for these grouped activities in the next iterations instead of returning loose choice graphs that overgeneralize the behavior of the event log.

Projection of choice graph cuts

A crucial aspect of the Inductive Mining framework is the projection of the event log onto the identified parts to create sub-logs. In the context of acyclic choice graphs (Kourani et al. 2025a), a trivial projection strategy was sufficient: for a partition part A_i , the sub-log was obtained by simply filtering each trace to keep only the activities in A_i. However, with the introduction of cyclic choice graphs, this trivial projection is no longer valid. In a cyclic graph, a single trace may enter and leave a specific part A_i multiple times. A simple projection would concatenate these distinct executions into a single sequence, potentially creating false direct-succession relationships. To handle this, we adopt a projection strategy that isolates consecutive executions, similar to the logic used for the loop operator in the base Inductive Miner (Leemans 2022).

Definition 7 [Choice Graph Projection] Let L be an event log and $A \subseteq \Sigma_L$ be a subset of its activities. The choice graph projection of L onto A, denoted by $proj(L, A)$, is a multi-set of sequences defined as follows:

$$\begin{aligned} proj(L, A) = [\sigma \in A^* \mid\ & \sigma_{pre} \cdot \sigma \cdot \sigma_{post} \in L \\ & \wedge\ \sigma \neq \langle\rangle \\ & \wedge\ \big(\sigma_{pre} = \langle\rangle\ \vee\ \sigma_{pre}(|\ \sigma_{pre}\ |) \notin A\big) \\ & \wedge\ \big(\sigma_{post} = \langle\rangle\ \vee\ \sigma_{post}(1) \notin A\big)]. \end{aligned}$$

Intuitively, for every trace in the log, we extract all maximal consecutive sub-sequences that consist exclusively of activities from A. This ensures that every time the control flow enters a choice graph node, the internal behavior is recorded as a separate instance in the sub-log. For example, consider an event log $L = [\langle a, b, c, b, d\rangle, \langle a, b\rangle]$ and a subset of activities $A = \{a, b\}$. Using the choice graph projection from Definition 7, we obtain $proj(L, A) = [\langle a, b\rangle^2,\ \langle b\rangle]$.

Perfect fitness guarantee

Let PM denote our POWL 2.0 process discovery framework (Fig. 6). To ensure that the discovered POWL 2.0 models adequately represent the given event log, we establish a fitness guarantee for *PM*. 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 $\psi_i = \text{PM}(proj(L, A_i))$. 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:

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

This can be rewritten as:

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

We construct a suitable path $\Pi = \langle A_1, \ldots, A_k\rangle \in \vec{G}$ 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 6, 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 $\Pi = \langle A_1, \ldots, A_k\rangle \in \vec{G}$ $(k \leq m)$ as follows:

1. **Start node:** Since $a_1 \in L_\triangleright$, by Definition 6, 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 Definition 6 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 6 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 $\Pi = \langle A_1, \ldots, A_k \rangle \in \vec{G}$. Now, decompose σ into non-empty sub-traces $\sigma_1, \ldots, \sigma_k$ such that each σ_i constitutes a maximal consecutive segment of activities from A_i. This decomposition $\sigma = \sigma_1 \cdot \ldots \cdot \sigma_k$ exists since we constructed the path by extending it only when the part changes. Consequently, by Definition 7, $\sigma_i \in proj(L, A_i)$ for all $1 \leq i \leq k$. By the assumption of the lemma, each $proj(L, A_i) \subseteq \mathcal{L}(\psi_i)$. Thus:

$$\sigma = \sigma_1 \cdot \ldots \cdot \sigma_k \in proj(L, A_1) \cdot \ldots \cdot proj(L, A_k) \subseteq \mathcal{L}(\psi_1) \cdot \ldots \cdot \mathcal{L}(\psi_k) \subseteq \mathcal{L}(G').$$

□

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

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 (Kourani et al. 2025b).
- **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 partial order cut or a standard process tree cut is detected (either in the main cut detection phase or as a fall-through):** The theorem holds since these cuts are fitness-preserving (Kourani et al. 2025b).
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 $\psi_i = PM(proj(L, A_i))$ in G, returning another choice graph G'. By the inductive hypothesis, each sub-trace $\sigma_i \in proj(L, A_i)$ is in the language of the recursively discovered sub-model, i.e., $\sigma_i \in \mathcal{L}(PM(proj(L, A_i)))$. 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 $\mathcal{L}(\text{PM}(L))$. □

Strategies for mining for choice graphs

An initial approach for mining for choice graph cuts was proposed in (Kourani et al. 2025a). This approach relies on identifying acyclic choice graphs. The acyclicity assumption forces any cyclic behavior to be captured either by the process tree binary loop

operator ($\circlearrowleft$) or, failing that, by the fall-through mechanism. While this ensures sound, structured models, it limits the expressiveness of the discovered models regarding unstructured cycles. To address this, we extend our discovery framework to discover *cyclic choice graphs*.

Conceptually, the DFG of an event log can be interpreted as a candidate cyclic choice graph. However, utilizing the raw DFG structure is risky because it overgeneralizes concurrency. In a DFG, concurrency between two activities a and b manifests as a cycle ($a \mapsto b$ and $b \mapsto a$). If we blindly interpret this as a loop, we mask the concurrency, preventing the Inductive Miner from detecting it in subsequent recursive steps. Therefore, the challenge is to identify a partition of activities A such that the cycles appearing between the parts in the induced graph $\diamondsuit_L(A)$ represent genuine decision points rather than concurrency.

To balance the trade-off between capturing flexible decision logic and preventing concurrency from being misidentified as loops, we propose a hierarchy of four cut strategies. This hierarchy ranges from the most permissive approach (Level 0), which accepts all cycles, to intermediate strategies (Levels 1 and 2) that selectively merge components connected by short loops (bidirectional edges). Finally, it includes the strictest strategy (Level 3), which enforces total acyclicity, aligning with the approach from (Kourani et al. 2025a).

Level 0: The Atomic cut (DFG fall-through)

The most fundamental form of a cyclic choice graph cut is one where every activity remains independent (i.e., forms its own singleton in the partition), allowing for any cycle present in the DFG. We refer to this as the *Atomic Cut*.

Definition 8 [Atomic Choice Graph Cut]. Let L be an event log. The atomic choice graph cut is defined as $C = (A, G)$ where $A = \{\{a\} \mid a \in \Sigma_L\}$ and $G = \diamondsuit_L(A)$.

Since a DFG can always be constructed for an event log, this cut is unique and *always exists*. This cut is the least precise choice graph cut as it interprets concurrency as cycles and, therefore, it should not be used as a primary strategy for choice graph cut detection. In the context of our discovery framework (cf. Fig. 6), this cut serves as the ultimate fall-through mechanism. Unlike the standard "flower model" fall-through, which loses all ordering information, the Atomic Choice Graph Cut preserves the direct-succession constraints observed in the log.

Level 1: Short-loop-free cut

To improve precision, we must distinguish between genuine decision loops and loops arising from concurrency. The strongest indicator of concurrency in a DFG is an activity-level short loop. If two activities a and b directly follow each other ($a \mapsto b$ and $b \mapsto a$), they are likely either concurrent or form a strict binary loop best handled by the process tree operator $\circlearrowleft$. Therefore, *Level 1* imposes a restriction: no two distinct parts of the partition may be connected by a bidirectional edge pair between the same activities. This forces such activities to be merged into the same part, delegating the resolution of their relationship (concurrency or a binary loop) to the recursive step.

Definition 9 [Short-Loop-Free Cut] A choice graph cut $(A = \{A_1, \ldots, A_n\}, G = (N, E))$ is *short-loop-free* if and only if $G = \Diamond_L(A)$ and for all $A_i, A_j \in A; a \in A_i, b \in A_j$:

$$a \mapsto b \wedge b \mapsto a \Rightarrow A_i = A_j.$$

Crucially, this cut prevents the most obvious form of concurrency from being masked as a choice loop, while still allowing for cycles between parts if the entry and exit activities are distinct (e.g., entering A_j via a but returning to A_i via b, where $a \neq b$).

Algorithm 1 outlines the procedure for identifying a *maximal* short-loop-free cut. It initializes each activity as a separate group and iteratively merges any pair of parts containing activities connected by bidirectional edge pairs. By performing the minimum number of merges required to satisfy the condition, the algorithm guarantees finding the partition of maximal size that is free of short loops.

Input: An event log L.
Output: A choice graph cut $(A, \Diamond_L(A))$ or $\bot$.
Function `MineCG_1(`L`)`**:**
  $A \leftarrow \{\{a\} \mid a \in \Sigma_L\}$
  for $(a, b) \in \Sigma_L \times \Sigma_L$ **do**
    if $(a \mapsto b \ \wedge \ b \mapsto a)$ **then**
      $A \leftarrow A \setminus \{A_a, A_b\} \cup \{A_a \cup A_b\}$
  if $|A| \geq 2$ **then**
    return $(A, \Diamond_L(A))$
  return $\bot$

Level 2: Strict short-loop-free cut

While Level 1 handles explicit concurrency, it relies on the DFG containing all necessary edges (i.e., $a \leftrightarrow b$ between all activity pairs $a \in A_i$ and $b \in A_j$). However, real-world data is often incomplete, and some concurrency edges may be missing. If Level 1 is used, these loose cycles would be interpreted as decision loops, preventing the discovery of the underlying "incomplete" concurrency.

To address this, *Level 2* employs a stricter strategy: it forbids *any* bidirectional flow between two parts, even if the edges involve different activities. If two parts A_i and A_j are mutually reachable ($A_i \mapsto A_j$ and $A_j \mapsto A_i$), they are merged into the same part. The motivation for this strictness is to leverage the Inductive Miner's robust *fall-through mechanisms*. The Inductive Miner includes fall-throughs specifically designed to detect "less obvious" concurrency (e.g., an activity that occurs in parallel with the rest) even when DFG edges are missing. By forcing a merge in ambiguous cyclic cases, Level 2 delegates the structural decision to these fall-throughs, which may provide a more precise representation than a loose choice graph cycle.

Definition 10 [Strict Short-Loop-Free Cut] A choice graph cut $(A = \{A_1, \ldots, A_n\}, G = (N, E))$ is *strict short-loop-free* if and only if $G = \Diamond_L(A)$ and for all $A_i, A_j \in A$:

$$(A_i \mapsto A_j \ \wedge \ A_j \mapsto A_i) \Rightarrow A_i = A_j.$$

It is important to note that this cut does not result in an acyclic graph. It only eliminates loops of length 2 between parts, relying on the binary loop cut ($\circlearrowleft$) to detect such loops

instead. It still supports longer decision cycles (e.g., $A_i \to A_j \to A_k \to A_i$), allowing for the discovery of unstructured cycles while preventing ambiguous bidirectional dependencies from masking concurrency.

Algorithm 2 outlines the procedure for identifying a maximal strict short-loop-free cut. Similar to the previous level, this algorithm starts with the atomic partition and performs merges only when the strict validity condition is violated (i.e., when two parts are mutually reachable). This process finds the maximal partition where no two parts are connected by a loop of length 2.

Input: An event log L.
Output: A choice graph cut $(A, \Diamond_L(A))$ or $\bot$.
Function `MineCG_2(`L`)`**:**
  $A \leftarrow \{\{a\} \mid a \in \Sigma_L\}$
  for $a, a', b, b' \in \Sigma_L$ **do**
    if $a \mapsto b \ \wedge \ b' \mapsto a' \ \wedge \ A_a = A_{a'} \ \wedge \ A_b = A_{b'}$ **then**
      $A \leftarrow A \setminus \{A_a, A_b\} \cup \{A_a \cup A_b\}$
  if $|A| \geq 2$ **then**
    return $(A, \Diamond_L(A))$
  return $\bot$

Level 3: Acyclic cut

Finally, we identify the strictest strategy: the *Acyclic Cut*. This strategy corresponds to the approach we proposed in (Kourani et al. 2025a).

While Levels 1 and 2 allow for unstructured cycles to exist within the choice graph, Level 3 forbids all cycles. By merging all activities involved in a cycle into a single part, this strategy limits the discovery algorithm to only capture cyclic behavior that can be represented by the dedicated block-structured loop operator ($\circlearrowleft$) in the recursive steps.

Definition 11 [Acyclic Choice Graph Cut] A choice graph cut $(A = \{A_1, \ldots, A_n\}, G = (N, E))$ is *acyclic* if and only if $G = \Diamond_L(A)$ and for all $A_i, A_j \in A$:

$$(A_i \mapsto^+ A_j \ \wedge \ A_j \mapsto^+ A_i) \Rightarrow A_i = A_j.$$

Algorithmically, finding the maximal acyclic cut corresponds to finding the *Strongly Connected Components (SCCs)* of the DFG. As shown in Algorithm 3, each SCC becomes a part in the partition.

Input: An event log L.
Output: A choice graph cut $(A, \Diamond_L(A))$ or $\bot$.
Function `MineCG_3(`L`)`**:**
  $A \leftarrow \{\{a\} \mid a \in \Sigma_L\}$
  for $(a, b) \in \Sigma_L \times \Sigma_L$ **do**
    if $(a \mapsto^+ b \ \wedge \ b \mapsto^+ a)$ **then**
      $A \leftarrow A \setminus \{A_a, A_b\} \cup \{A_a \cup A_b\}$
  if $|A| \geq 2$ **then**
    return $(A, \Diamond_L(A))$
  return $\bot$

Summary and hypothesis

We have introduced a hierarchy of strategies for mining choice graphs, each tailored to a different interpretation of cyclic behavior in the DFG. Level 0 is distinct from the others; it is not intended for primary cut detection but serves as a generic fall-through mechanism to ensure termination when no structural pattern is found.

Among the primary strategies (Levels 1–3), the choice of strictness directly dictates the theoretical consequences for the resulting model's expressiveness and precision:

- **Level 1 (Short-Loop-Free):** This is the most permissive strategy. While theoretically optimal for highly complete logs where all concurrent behavior is recorded as explicit length-2 loops, we hypothesize it is too permissive for real-world scenarios. It risks interpreting broken concurrency (due to missing data) as complex, imprecise decision loops.
- **Level 3 (Acyclic):** This is the strictest strategy. It aligns with previous approaches by forcing all cyclic behavior into well-nested, block-structured loops. We hypothesize this is overly restrictive, effectively negating the primary advantage of choice graphs to model unstructured "state-machine" cycles.
- **Level 2 (Strict Short-Loop-Free):** This strategy offers an intermediate approach. By filtering out all ambiguous bidirectional connections, whether tight or loose, it prevents the masking of hidden concurrency. It relies on the robust fall-through mechanisms of the Inductive Miner to resolve these ambiguous cases, while still allowing for clear, unstructured decision cycles.

Consequently, we hypothesize that Level 2 offers the most pragmatic trade-off for real-world event logs. We rigorously investigate this hypothesis and the practical consequences of these strategies in our experimental evaluation (Sect. "Evaluation").

Connectivity-preserving noise filtering

To effectively apply and evaluate our discovery algorithms on real-world event data, an adequate noise-handling mechanism is essential. In this section, we first illustrate the limitations of standard frequency-based filtering techniques in Sect. "Motivation: The Destructive Nature of Filtering on Graph Connectivity" and subsequently detail our proposed repair methodology in Sect. "Methodology: Global Structural Repair Phase".

Motivation: the destructive nature of filtering on graph connectivity

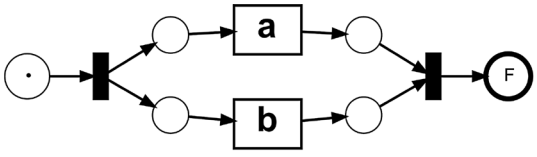
Real-world event logs frequently contain noise, outliers, and infrequent behavior that obscure the primary control flow. The Inductive Miner framework addresses this through a multi-stage cut detection strategy. At each recursive step, if no valid cut is found on the original DFG, the algorithm filters the DFG by removing edges that fall below a certain frequency threshold and retries.

While this strategy is effective at reducing complexity, the filtering step itself is often *destructive* regarding connectivity. Standard filtering approaches simply discard infrequent edges, which can isolate activities, rendering them unreachable from the process start or unable to reach the process end. Depending on the concrete implementation of each cut, such disconnected activities may be permanently discarded during the recursion steps, leading to a major loss of information where activities disappear entirely from the discovered model.

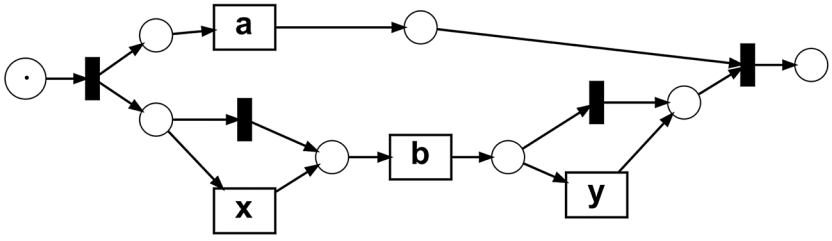
To demonstrate the issue with the standard filtering approach of the Inductive Miner, consider the following event log, which contains a primary concurrent-like behavior between activities a and b, alongside rare deviations involving activities x and y:

$$L_1 = [\langle a, b\rangle^5, \langle b, a\rangle^5, \langle a, x, b\rangle^1, \langle b, y, a\rangle^1].$$

The Inductive Miner filters the DFG by removing edges whose relative frequency falls below a user-defined threshold. Specifically, an edge is discarded if its occurrence count is less than the threshold proportion of the maximum frequency of any edge connected to the respective nodes. When applying the Inductive Miner using its default settings in ProM[3] (i.e., a filtering threshold of 0.2), the algorithm discards the edges (a, x), (x, b), (b, y), and (y, a) because their relative frequencies fall well below the 0.2 threshold compared to the dominant (a, b) and (b, a) relations. As depicted in Fig. 7a, the Petri net discovered by the Inductive Miner only contains activities a and b, completely removing x and y from the final discovered model.



(a) WF-net discovered by the Inductive Miner in ProM (default settings, threshold 0.2). Activities x and y are completely discarded due to the applied filtering.



(b) WF-net discovered using the Inductive Miner after integrating our connectivity-preserving repair step. Activities x and y are successfully retained and integrated into the global control flow.

Fig. 7 Comparison of discovery approaches on the event log $L_1 = [\langle a, b\rangle^5, \langle b, a\rangle^5, \langle a, x, b\rangle^1, \langle b, y, a\rangle^1]$

Applying repair mechanisms to enforce connectivity after graph filtering is a proven concept in process discovery. Legacy approaches, most notably the Heuristics Miner, attempt to overcome this limitation by introducing local connectivity guarantees. The Heuristics Miner includes an "All Tasks Connected" parameter, which is enabled by default. If the global filtering threshold isolates a node, the algorithm locally overrides the threshold to rescue the strongest original incoming and outgoing dependencies for that specific node. While this heuristic is effective at preventing isolated nodes, it fundamentally fails to guarantee global graph connectivity and reachability. To illustrate this critical limitation, consider event log L_2, which features a dominant process path and a rare deviation containing an internal loop:

$$L_2 = [\langle a, b\rangle^5, \langle a, x, y, z, x, y, z, x, y, z, b\rangle^1].$$

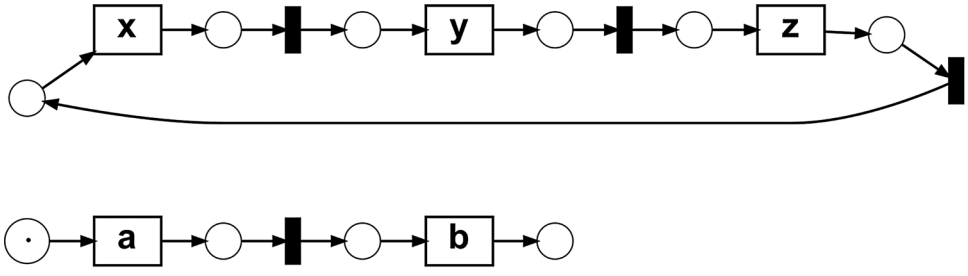
[3] https://promtools.org/.

Figure 8a illustrates the Petri net discovered by the Heuristics Miner on L_2 using default settings in ProM (with "All Tasks Connected" enabled). Because the entry and exit edges of the deviation ((a, x) and (z, b)) occur only once, they are filtered out by the global threshold. To satisfy the local connectivity rule, the Heuristics Miner rescues the internal loop edges because they are the strongest local relations for x, y, and z. The result is a structurally disconnected Petri net: it generates a main process line ($a \to b$) and a completely disconnected cycle ($x \to y \to z \to x$). Although local connectivity constraints are satisfied (no node has an in/out degree of 0), global reachability is broken. There is no valid path from the initial marking to activities x, y, and z. This example highlights why a robust filtering approach must move beyond local edge rescue and actively ensure global connectivity.

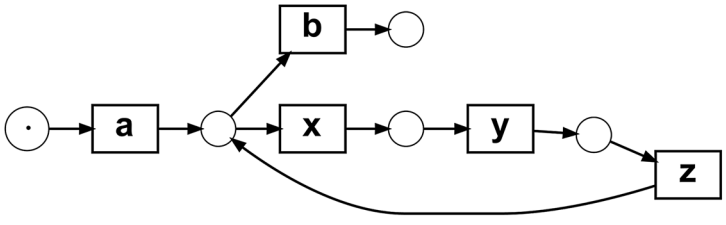
Methodology: global structural repair phase

Rather than introducing a completely new filtering technique, we reuse the standard DFG-based local filtering mechanism from (Leemans 2022) and extend it with a formal *structural repair* phase. This phase follows a similar logic to the repair mechanisms used in the Split Miner (Augusto et al. 2019) (although the Split Miner uses a different intermediate graph, not directly the DFG). This approach allows for aggressive filtering of infrequent behavior while strictly enforcing that every activity present in the original log remains part of the process backbone (i.e., on a valid path between the start and end nodes). Figures 7b and 8b demonstrate the sound WF-nets that can be discovered by extending the Inductive Miner's filtering logic with our proposed repair step.

Step 1: Augmented graph representation Let L be an event log over activities Σ_L. To formalize connectivity, we construct an *Augmented DFG*, denoted as $G_{aug} = (V_{aug}, w)$:



(a) Unsound Petri net discovered by the Heuristics Miner in ProM (default settings, "All Tasks Connected" enabled). The model contains a disconnected cyclic subgraph, breaking global reachability.



(b) Sound WF-net discovered using the inductive miner.

Fig. 8 Comparison of discovery approaches on event log $L_2 = [\langle a, b\rangle^5, \langle a, x, y, z, x, y, z, x, y, z, b\rangle^1]$

- $V_{aug} = \Sigma_L \cup \{\triangleright, \square\}$, where $\triangleright$ is an artificial unique source and $\square$ is an artificial unique sink.
- $w : V_{aug} \times V_{aug} \to \{0, 1, 2, \ldots\}$ is an edge weight function defined as follows:

$$w(u,v) = \begin{cases} L_{\triangleright}(v) & \text{if } u = \triangleright \ \wedge \ v \in L_{\triangleright}, \\ L_{\square}(u) & \text{if } v = \square \ \wedge \ u \in L_{\square}, \\ L_{\mapsto}(u,v) & \text{if } (u,v) \in L_{\mapsto}, \\ 0 & \text{otherwise.} \end{cases}$$

Step 2: Initial threshold filtering Given a user-defined noise threshold $0 < \tau \leq 1$, we first apply the local filtering criterion from (Leemans 2022). For every node $u \in V_{aug} \setminus \{\square\}$, an outgoing edge (u, v) is retained only if its weight is significant relative to the maximum outgoing frequency, i.e., the set of retained edges is:

$$E_{init} = \{(u,v) \in V_{aug} \times V_{aug} \mid w(u,v) \geq \tau \cdot \max_{v \in V_{aug}} w(u,v)\}.$$

Step 3: Connectivity repair mechanism The graph $G_\tau = (V_{aug}, E_{init})$ resulting from threshold filtering may be disconnected; we consider an activity *broken* if it is not reachable from the source or cannot reach the sink. To preserve the completeness of the activity set, we iteratively restore connectivity for these nodes using edges from the original augmented graph G_{aug}.

We formulate this repair strategy as an instance of the *Maximum Capacity Path problem* (Pollack 1960), which seeks to identify paths where the minimum edge weight is maximized. For every broken activity, we identify and add the optimal paths connecting the source to the activity (forward repair) and/or the activity back to the sink (backward repair), thereby ensuring the reintroduction of the most significant available behavior. The Maximum Capacity Path problem is a well-studied problem in graph theory. It can be solved for directed graphs using a modified version of Dijkstra's algorithm (Pollack 1960). We employ this standard algorithmic approach to repair the augmented DFG but omit detailed implementation specifics for brevity. We use E_{final} to denote the final repaired augmented edge set.

Step 4: Reconstruction of filtered artifacts The output of the filtering process is a refined set of artifacts that replaces the raw DFG and start/end information used for cut detection logic. The POWL discovery algorithm utilizes both the DFG and the EFG for cut detection. Specifically, the EFG is used for mining partial order cuts. By definition, a valid eventually-follows relation $a \rightsquigarrow b$ implies the existence of a path from a to b in the DFG as well. While this naturally holds for the original log ($\rightsquigarrow_L \subseteq \mapsto_{L^+}$), the filtering of the DFG may introduce contradictions between the EFG and the DFG. Therefore, we must restrict the EFG to only those dependencies that remain supported by valid paths in the filtered DFG.

Given the repaired augmented edge set E_{final}, we construct the final filtered process mining artifacts as follows:

- **Filtered Start Activities:** $L'_{\triangleright} = \{a \in \Sigma_L \mid (\triangleright, a) \in E_{final}\}$.
- **Filtered End Activities:** $L'_{\square} = \{a \in \Sigma_L \mid (a, \square) \in E_{final}\}$.
- **Filtered DFG:** $\mapsto'_L = E_{final} \cap (\Sigma_L \times \Sigma_L)$.

- **Filtered EFG:** $\rightsquigarrow'_L = \rightsquigarrow_L \cap \mapsto'^{+}_L$.

These filtered artifacts can serve as the input for the cut detection strategies within our discovery framework.

Adaptive choice graph projection under filtering Noise filtering simplifies the DFG by removing low-frequency edges, but this also weakens the evidence for control-flow boundaries. In particular, when projecting traces onto a choice graph part A, filtering may obscure whether two occurrences of activities from A within the same trace correspond to two distinct visits or to a single local execution separated by noise. We therefore adapt the projection for cyclic choice graph cuts to only separate segments when there is sufficient evidence that the trace genuinely leaves and later re-enters A.

Let $\sigma \in L$ be a trace that can be decomposed as $\sigma = \sigma_{\text{pre}} \cdot \sigma_1 \cdot \sigma_2 \cdot \sigma_3 \cdot \sigma_{\text{post}}$, where $\sigma_1, \sigma_3 \in A^*$ are non-empty segments over A and $\sigma_2 \in (\Sigma_L \setminus A)^*$ is non-empty. Let $a = \sigma_1(|\sigma_1|)$, $b = \sigma_2(|\sigma_2|)$, and $a' = \sigma_3(1)$. We keep σ_1 and σ_3 as separate projected segments only if the filtered DFG provides stronger evidence for re-entry from outside than for local continuation, i.e.,

$$(b, a') \in E_{final} \ \wedge \ w(b, a') > (a, a').$$

Otherwise, the two segments σ_1 and σ_3 are merged. This design favors combining behavior when the evidence for cyclic re-entry is weak.

Evaluation

We evaluate the proposed POWL 2.0 discovery framework by comparing the standard POWL Inductive Miner (PM) from (Kourani et al. 2023c) against three variants of our new approach. These variants correspond to the hierarchy of choice graph mining strategies defined in Sect. "Strategies for Mining for Choice Graphs":

- PM_{AC}: The **Acyclic** approach (Level 3), which enforces block-structured loops by merging all strongly connected components in the DFG.
- PM_{C1}: The **Short-Loop-Free** approach (Level 1), which permits cycles between parts provided they are not connected by direct bidirectional edges between the same activities.
- PM_{C2}: The **Strict Short-Loop-Free** approach (Level 2), which forbids any bidirectional direct reachability between parts. This is the strategy hypothesized in Sect. "Strategies for Mining for Choice Graphs" to offer the best trade-off between expressiveness and robustness.

All approaches are available in the Python library 'powl', which can be installed via 'pip install powl'. An example script for using the library is available at https://github.com/humam-kourani/POWL/blob/main/examples/powl_discovery.py.

We compare the discovery approaches, evaluating two variants of each: one without any noise filtering and another variant applying our noise filtering mechanism proposed in Sect. "Connectivity-Preserving Noise Filtering". 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.

All discovered process models are available at https://doi.org/10.5281/zenodo.18009133. 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 (Adriansyah et al. 2011), while precision assesses how strictly the model conforms to the observed behavior (Adriansyah et al. 2015). We use the fitness and precision implementations available in PM4Py (Berti et al. 2023) 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.

Runtime performance

Table 1 details the execution time for all approaches. A key observation is that the introduction of choice graph mining does not negatively impact scalability. On the contrary, significant performance gains are observed for a few cases. For instance, for the BPIC2018 log with noise filtering, the execution time dropped from 546.5 seconds with the baseline PM to 79.9 seconds for PM_{C1}. This demonstrates that mining for choice graphs does not compromise the inherent scalability of the Inductive Miner; in fact, it can improve runtime performance when valid choice graphs are detected in early iterations.

Quality of discovered models

We split the analysis of the qualitative results into two parts: first, comparing the baseline POWL Miner (PM) against our new POWL 2.0 miners in general, and second, validating our hypothesis regarding the optimal strategy for mining cyclic choice graphs. Table 2 summarizes the qualitative results.

Table 1 Time performance results in seconds

Event Log	No Filtering				Noise Threshold 0.2			
	PM	PM_{AC}	PM_{C1}	PM_{C2}	PM	PM_{AC}	PM_{C1}	PM_{C2}
BPIC2012	21.8	21.0	**5.3**	**5.5**	2.7	2.1	1.3	1.4
BPIC2013 - Closed	0.1	0.1	0.1	0.1	0.05	0.04	0.05	0.04
BPIC2013 - Incidents	0.7	0.7	0.5	0.7	0.5	1.0	0.5	0.7
BPIC2013 - Open	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.02
BPIC2017	18.1	17.7	**3.7**	15.3	3.7	3.3	1.9	3.1
BPIC2017 - Offer Log	0.7	0.6	0.6	0.6	0.7	0.6	0.6	0.6
BPIC2018	687.2	684.1	685.9	690.3	546.5	551.2	**79.9**	551.4
BPIC2019	270.9	268.5	**75.0**	272.1	56.0	34.1	35.8	37.2
BPIC2020 - Dom. Dec.	0.3	0.3	0.2	0.2	0.3	0.2	0.2	0.1
BPIC2020 - Int. Dec.	2.4	2.3	1.5	2.1	0.5	0.4	0.7	0.9
BPIC2020 - Permit	21.8	21.8	**6.5**	39.1	2.2	2.1	3.6	5.9
BPIC2020 - Prepaid	1.3	1.4	0.3	2.6	0.4	0.4	0.2	0.6
BPIC2020 - Request	0.2	0.2	0.1	0.1	0.2	0.1	0.1	0.1
Hospital Billing	2.9	2.7	2.6	3.0	2.5	2.0	1.8	1.9
Receipt Phase WABO	0.5	0.5	0.3	0.6	0.3	0.3	0.3	0.2
Road Traffic Fine	1.3	1.2	1.1	1.1	1.2	1.1	1.1	1.2
Sepsis Cases	1.5	1.6	0.7	1.8	0.3	0.3	0.3	0.3

Significant time improvements are highlighted in bold

Table 2 Qualitative evaluation results. For the cases with no filtering, we only report f-scores since all discovered models achieved perfect fitness. The highest f-score values per log are highlighted in bold. Missing values are due to timeouts

Event Log	**No Filtering**				**Noise Threshold 0.2**											
	f-score				**Fitness**				**Precision**				**f-score**			
	PM	PM_{AC}	PM_{C1}	PM_{C2}	PM	PM_{AC}	PM_{C1}	PM_{C2}	PM	PM_{AC}	PM_{C1}	PM_{C2}	PM	PM_{AC}	PM_{C1}	PM_{C2}
BPIC2012	0.27	0.28	**0.33**	**0.33**	0.96	0.95	0.90	0.90	0.32	0.43	0.31	0.31	0.48	**0.59**	0.46	0.46
BPIC2013 - Closed	0.88	0.88	0.88	0.88	0.88	0.88	0.88	0.88	0.80	0.80	0.80	0.80	0.84	0.84	0.84	0.84
BPIC2013 - Incidents	0.77	0.77	0.77	0.77	0.94	0.94	0.94	0.93	0.80	0.80	0.80	0.84	0.86	0.86	0.86	**0.88**
BPIC2013 - Open	0.95	0.95	0.95	0.95	0.83	0.83	0.83	0.83	0.91	0.91	0.91	0.91	0.87	0.87	0.87	0.87
BPIC2017	0.45	**0.47**	0.33	0.44	0.98	0.93	0.92	0.94	0.52	0.54	0.51	0.59	0.68	0.69	0.66	**0.73**
BPIC2017 - Offer Log	0.91	**1.00**	**1.00**	**1.00**	1.00	1.00	1.00	1.00	0.90	1.00	1.00	1.00	0.94	**1.00**	**1.00**	**1.00**
BPIC2018	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
BPIC2019	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
BPIC2020 - Dom. Dec.	0.56	**0.63**	0.53	0.59	0.89	1.00	0.99	0.97	0.47	0.43	0.43	0.61	0.62	0.60	0.60	**0.75**
BPIC2020 - Int. Dec.	0.24	0.31	0.18	**0.33**	0.92	0.90	0.88	0.88	0.24	0.49	0.73	0.74	0.38	0.64	0.79	**0.80**
BPIC2020 - Permit	0.16	**0.17**	0.14	**0.17**	0.92	0.94	0.87	0.86	0.24	0.35	0.67	0.70	0.38	0.51	0.76	**0.77**
BPIC2020 - Prepaid	0.22	0.22	0.22	0.22	0.94	0.96	0.95	0.94	0.25	0.28	0.59	0.73	0.39	0.44	0.73	**0.83**
BPIC2020 - Request	0.45	0.51	0.53	**0.69**	0.97	1.00	1.00	0.98	0.45	0.46	0.40	0.63	0.62	0.63	0.57	**0.76**
Hospital Billing	0.67	0.67	**0.71**	0.67	0.96	0.98	0.88	0.87	0.64	0.73	0.89	0.86	0.77	0.84	**0.89**	0.86
Receipt Phase WABO	0.28	0.30	0.27	**0.31**	0.94	0.97	0.82	0.90	0.22	0.25	0.37	0.48	0.36	0.40	0.51	**0.62**
Road Traffic Fine	**0.74**	**0.74**	0.70	**0.74**	0.91	0.92	0.99	0.99	0.79	0.88	0.93	0.93	0.85	0.90	**0.96**	**0.96**
Sepsis Cases	**0.44**	**0.44**	0.37	**0.44**	0.94	0.92	0.85	0.86	0.36	0.34	0.66	0.60	0.52	0.50	**0.75**	0.71
Average	0.53	0.55	0.53	**0.57**	0.93	0.94	0.91	0.92	0.53	0.58	0.67	0.71	0.64	0.69	0.75	**0.79**

Baseline vs. POWL 2.0

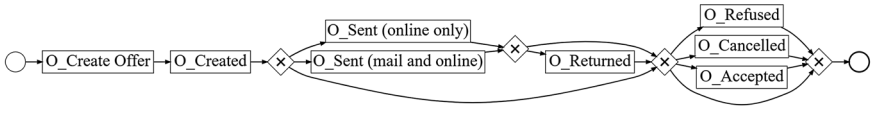
The results demonstrate a substantial improvement in model quality in general. In the setting without filtering, the improvements are modest due to the high presence of noise in real-life logs; the average f-score increases slightly from 0.53 for the baseline PM to 0.57 for the optimal cyclic strategy (PM_{C2}). The advantages of POWL 2.0 become much more pronounced in the noise filtering setting. Here, the baseline PM achieves an average f-score of 0.64. The introduction of choice graph mining strategies increases this to an average of 0.69 for PM_{AC}, 0.75 for PM_{C1}, and 0.79 for PM_{C2}. While fitness deviations between the different approaches are modest, the primary driver for this improvement is precision. For example, in the BPIC2020 - Prepaid Travel Cost log with filtering, precision jumps from 0.25 (Baseline) to 0.73 (PM_{C2}), resulting in an f-score increase from 0.39 to 0.83.

This improvement stems from the ability of choice graphs to model decision logic that does not fit into block-structured process tree operators. Figure 9 presents the process models discovered for the BPIC2017 - Offer Log in the setting without filtering, converted into BPMN. In this case, mining for choice graphs increased precision from 0.84 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 $PM_{AC} PM_{C1}$, and PM_{C2} 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.

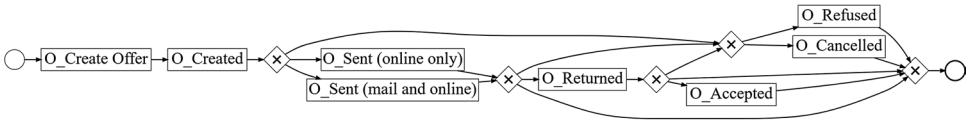
Comparison of choice graph mining strategies

In Sect. "Strategies for Mining for Choice Graphs", we hypothesized that Level 1 (PM_{C1}) might be too permissive, Level 3 (PM_{AC}) too strict, and Level 2 (PM_{C2}) would offer the optimal balance. The experimental results support this hypothesis, establishing Level 2 as the superior strategy on average.

The Acyclic strategy (PM_{AC}) improves over the baseline (average f-score 0.69 vs 0.64 with filtering, 0.55 vs 0.53 without filtering), but it lags behind the cyclic strategies in general. By forcing cyclic components into block-structured loops, it fails to capture the intricate "state-machine"-like cyclic behavior present in many logs, confirming that strict acyclicity is too limiting for general process discovery. Between the cyclic strategies, PM_{C2} (Level 2) performs better than PM_{C1} (Level 1) on average: f-score 0.79 vs 0.75 with filtering, 0.57 vs 0.53 without filtering. This confirms that by enforcing a

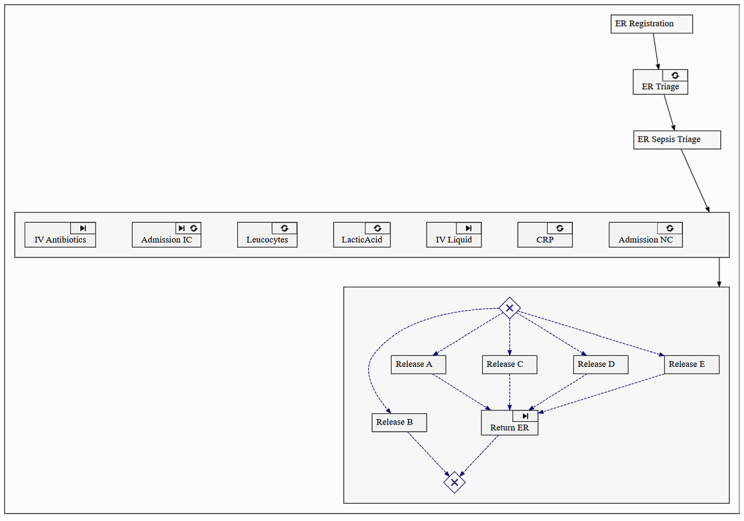


(a) By PM.

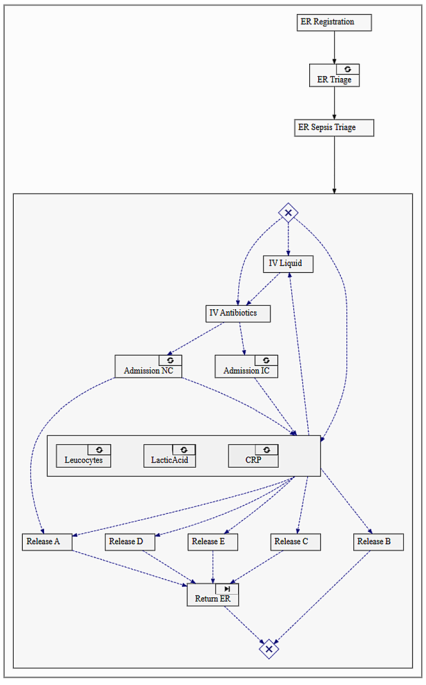


(b) By PM_{AC}, PM_{C1}, and PM_{C2}.

Fig. 9 Process models discovered for the BPIC2017 - offer log



(a) Model discovered by the acyclic strategy (PM_{AC}).



(b) Model with a cyclic choice graph discovered by PM_{C2}.

Fig. 10 Comparison of POWL 2.0 models for the *Sepsis cases* log (filtered). For compactness, we use the skip forward symbol to indicate that a sub-model is skippable and the repeat symbol to indicate that a sub-model can be repeated in a self-loop

stricter constraint for cycles in choice graphs, the robust fall-through mechanisms can often resolve ambiguous structures into more precise constructs.

To illustrate the benefits of mining cyclic choice graphs, we analyze the models discovered for the Sepsis Cases log with noise filtering. Figure 10 compares the model generated by the acyclic strategy (PM_{AC}) against the one generated by the Strict Short-Loop-Free strategy (PM_{C2}). This process contains an unstructured loop involving seven activities: CRP, IV Liquid, LacticAcid, Leucocytes, Admission IC, IV Antibiotics, and Admission NC. The acyclic strategy merges these activities into a single block, resulting

in an imprecise model that allows these activities to be executed in any order. In contrast, the cyclic strategy (PM_{C2}) discovers cyclic paths where specific dependencies are enforced. For example, in the cyclic model, "IV Liquid" must always be directly followed by "IV Antibiotics".

Discussion and practical implications

The experimental evaluation generally aligns with the hypothesis formulated in Sect. "Summary and Hypothesis". The results show that mining for cyclic choice graphs often yields superior models compared to the acyclic approach. Based on these studies, we can outline several practical consequences to guide users in selecting the appropriate strategy:

1. **Level 2 as a Default Configuration:** The Strict Short-Loop-Free strategy (PM_{C2}) provides a reliable default setting for unknown event logs. By conservatively delegating ambiguous cyclic structures to fall-through mechanisms, it balances the preservation of unstructured cycles with the avoidance of misinterpreting hidden concurrency, generally achieving the highest average f-score in our tests.
2. **Sensitivity to Noise Filtering:** The consequences of strategy selection become more apparent when noise filtering is applied. Filtering naturally removes lower-frequency edges, which can inadvertently erase the explicit indicators of concurrency (short loops). In such cases, more permissive strategies like Level 1 (PM_{C1}) have a higher risk of misinterpreting the remaining broken concurrency as decision loops. Level 2's strictness appears to mitigate this risk, making it more stable under standard filtering conditions.
3. **Context-Dependent Strategy Selection:** While Level 2 is a strong general-purpose choice, the optimal strategy depends on the specific use case and data characteristics. If a log is known to be highly complete with minimal missing behavior, the permissiveness of Level 1 may uncover valid, complex routing logic that Level 2 would otherwise merge. Conversely, if the underlying process is known to be highly standardized where complex, non-block-structured loops are simply not expected, then the acyclic strategy (PM_{AC}) becomes a safer choice. By enforcing block-structured loops, it prevents the algorithm from mistakenly interpreting noisy or incomplete concurrency as complex decision cycles, thereby keeping the model simple and structurally conservative.

In summary, the proposed hierarchy of choice graph strategies allows analysts to tailor the strictness of the discovery process. Rather than relying on a single rigid rule, users can adjust the algorithm based on the expected noise levels in their data and the anticipated complexity of the underlying process.

Conclusion

In this paper, we explored strategies for discovering complex, cyclic decision flows using the POWL 2.0 modeling language. Traditional discovery techniques often struggle to capture the flexibility of real-world processes, such as arbitrary jumps and unstructured loops, without compromising the formal guarantees of the model. By replacing the standard block-structured XOR and Loop operators with choice graphs, and extending the

Inductive Miner to discover them, we have enabled the generation of sound models that accurately reflect intricate decision logic.

A key challenge in discovering cyclic choice graphs is distinguishing between genuine decision loops and artifacts caused by hidden concurrency. We introduced and evaluated a hierarchy of mining strategies to resolve this ambiguity, ranging from permissive to strict. Our rigorous evaluation on 17 real-life event logs demonstrates that allowing for cyclic choice graphs significantly improves model quality, without sacrificing scalability.

Several promising directions for future work remain. First, we plan to explore further extensions to POWL to capture even more complex process structures, while carefully considering the trade-offs between expressiveness, scalability, and the preservation of formal guarantees. Furthermore, we plan to adapt our discovery approach for object-centric event data, enabling the modeling of processes involving multiple interacting object types. Finally, a critical area for improvement lies in enhancing the usability of the discovered models through interactive visualizations. This may include incorporating performance and conformance checking perspectives directly into the visual representation.

Author contribution
H.K. conceptualized the core methodology, conducted the experiments, and wrote the main manuscript text. G.P. and W.A. supervised the project, provided guidance, and reviewed the manuscript.

Funding
Open Access funding enabled and organized by Projekt DEAL. This work was supported by the Federal Ministry of Research, Technology and Space (BMFTR), Germany (grant 01IS23065).

Data availability
The dataset generated during the current study is available at https://doi.org/10.5281/zenodo.18009133.

Declarations

Ethical approval
Not applicable.

Competing interests
The authors declare no competing interests.

Received: 22 December 2025 / Accepted: 10 May 2026
Published online: 10 June 2026

References

Adriansyah A, Munoz-Gama J, Carmona J et al (2015) Measuring precision of modeled behavior. Inf Syst E-Bus Manage 13(1):37–67. https://doi.org/10.1007/S10257-014-0234-7

Adriansyah A, van Dongen BF, van der Aalst WMP (2011 August 29 - September 2) Conformance checking using cost-based fitness analysis. In Proceedings of the 15th IEEE International Enterprise Distributed Object Computing Conference, EDOC 2011, Helsinki, Finland, 55–64. https://doi.org/10.1109/EDOC.2011.12

Augusto A, Conforti R, Dumas M et al (2019a) Automated discovery of process models from event logs: review and benchmark. IEEE Trans Knowl Data Eng 31(4):686–705. https://doi.org/10.1109/TKDE.2018.2841877

Augusto A, Conforti R, Dumas M et al (2019b) Split miner: automated discovery of accurate and simple business process models from event logs. Knowl Inf Syst 59(2):251–284. https://doi.org/10.1007/S10115-018-1214-X

Bergenthum R, Desel J, Lorenz R et al (2007) Process mining based on regions of languages. In: Alonso G, Dadam P, Rosemann M (eds) Business Process Management, 5th International Conference, BPM 2007, vol 4714. Springer, Brisbane, Australia, 375–383. September 24–28, Proceedings, Lecture Notes in Computer Science. https://doi.org/10.1007/978-3-540-75183-0_27

Berti A, van Zelst SJ, Schuster D (2023) PM4Py: a process mining library for python. Softw Impacts 17:100556. https://doi.org/10.1016/J.SIMPA.2023.100556

Buijs JCAM, van Dongen BF, van der Aalst WMP (2012) A genetic algorithm for discovering process trees. In Proceedings of the IEEE Congress on Evolutionary Computation, CEC 2012, IEEE, Brisbane, Australia, 1–8. https://doi.org/10.1109/CEC.2012.6256458. 10–15 June 2012

Carmona J, Cortadella J, Kishinevsky M (2008) A region-based algorithm for discovering petri nets from event logs. In: Dumas M, Reichert M, Shan M (eds) Business Process Management, 6th International Conference, BPM 2008, vol 5240. Springer,

Milan, Italy, 358–373. September 2–4, 2008. Proceedings, Lecture Notes in Computer Science. https://doi.org/10.1007/978-3-540-85758-7_26
Dahari Y, Gal A, Senderovich A et al (2018) Fusion-based process discovery. In Advanced Information Systems Engineering - 30th International Conference, CAiSE 2018, Tallinn, Estonia, 291–307. June 11–15, 2018, Proceedings. https://doi.org/10.1007/978-3-319-91563-0_18
Desel J, Esparza J (1995) Free choice Petri nets. 40, Cambridge university press
Lectures on concurrency and Petri nets (2004) In: Desel J, Reisig W, Rozenberg G (eds) Advances in petri nets, lecture notes in computer science, vol 3098. Springer. https://doi.org/10.1007/B98282
Dumas M, García-Bañuelos L (2015) Process mining reloaded: event structures as a unified representation of process models and event logs. In: Devillers R, Valmari A (eds) Application and Theory of Petri Nets and Concurrency - 36th International Conference, PETRI NETS 2015, vol 9115. Springer, Brussels, Belgium, 33–48. June 21–26, 2015, Proceedings, Lecture Notes in Computer Science. https://doi.org/10.1007/978-3-319-19488-2_2
Favre C, Fahland D, Völzer H (2015) The relationship between workflow graphs and free-choice workflow nets. Inf Syst 47:197–219. https://doi.org/10.1016/J.IS.2013.12.004
Kourani H, Park G, van der Aalst WMP (2026) Hierarchical decomposition of separable workflow-nets. https://arxiv.org/abs/2602.15739
Kourani H, Park G, van der Aalst WMP (2025a) Unlocking non-block-structured decisions: inductive mining with choice graphs. In: Senderovich A, Cabanillas C, Vanderfeesten I et al (eds) Business Process Management - 23rd International Conference, BPM 2025, vol 16044. Springer, Seville, Spain, 144–161. August 31- September 5, 2025, Proceedings, Lecture Notes in Computer Science. https://doi.org/10.1007/978-3-032-02867-9_10
Kourani H, Schuster D, van der Aalst WMP (2023) Scalable discovery of partially ordered workflow models with formal guarantees. In 5th International Conference on Process Mining, ICPM 2023, Rome, Italy, 89–96. https://doi.org/10.1109/ICPM60904.2023.10271941. 23–27 Oct 2023
Kourani H, van Zelst SJ (2023) POWL: partially ordered workflow language. In Business Process Management - 21st International Conference, BPM 2023, Utrecht, The Netherlands, 92–108. September 11–15, 2023, Proceedings. https://doi.org/10.1007/978-3-031-41620-0_6
Kourani H, van Zelst SJ, Schuster D et al (2025b) Discovering partially ordered workflow models. Inf Syst 128:102493. https://doi.org/10.1016/J.IS.2024.102493
Leemans SJJ (2022) Robust process mining with guarantees - process discovery, conformance checking and enhancement. In: Lecture notes in business information processing, vol 440. Springer. https://doi.org/10.1007/978-3-030-96655-3
Leemans SJJ, Poppe E, Wynn MT (2019) Directly follows-based process mining: exploration & a case study. In International Conference on Process Mining, ICPM 2019, IEEE, Aachen, Germany, 25–32. https://doi.org/10.1109/ICPM.2019.00015. 24–26 June 2019
Leemans SJJ, Tax N, Ter Hofstede AHM (2018) Indulpet miner: combining discovery algorithms. In: Panetto H, Debruyne C, Proper HA et al. On the move to meaningful internet systems. OTM 2018 conferences - confederated international conferences: CoopIS, C&TC, and ODBASE 2018, Valletta, Malta, Proceedings, Part I, Lecture Notes in Computer Science, vol 11229. Springer, pp 97–115. https://doi.org/10.1007/978-3-030-02610-3_6. 22–26 Oct 2018
Pollack M (1960) Letter to the editor—the maximum Capacity through a network. Oper Res 8(5):733–736. https://doi.org/10.1287/opre.8.5.733
Slaats T, Schunselaar DMM, Maggi FM et al (2016) The semantics of hybrid process models. In: Debruyne C, Panetto H, Meersman R et al. (eds) On the Move to Meaningful Internet Systems: OTM 2016 Conferences - Confederated International Conferences: CoopIS, C&TC, and ODBASE 2016, Rhodes, Greece, 531–551. Proceedings. https://doi.org/10.1007/978-3-319-48472-3_32. 24–28 Oct 2016
van Beest Nrtp, Dumas M, García-Bañuelos L et al (2015) Log delta analysis: interpretable differencing of business process event logs. In: Motahari-Nezhad HR, Recker J, Weidlich M (eds) Business Process Management - 13th International Conference, BPM 2015, vol 9253. Springer, Innsbruck, Austria, 386–405. https://doi.org/10.1007/978-3-319-23063-4_26. August 31- September 3, 2015
van der Aalst Wmp, Masellis RD, Francescomarino CD et al (2023) Discovering hybrid process models with bounds on time and complexity: when to be formal and when not? Inf Syst 116:102214. https://doi.org/10.1016/J.IS.2023.102214
van Dongen Bf, de Medeiros Aka, Wen L (2009) Process mining: overview and outlook of petri net discovery algorithms. Trans Petri Nets Other Model Concurr 2:225–242. https://doi.org/10.1007/978-3-642-00899-3_13
von Rosing M, White S, Cummins F et al (2015) Business process model and notation - BPMN. In: von Rosing M, von Scheel H, Scheer A (eds) The complete business process handbook: body of knowledge from process modeling to BPM, vol I. Morgan Kaufmann/Elsevier, Massachusetts, USA, pp 429–453. https://doi.org/10.1016/B978-0-12-799959-3.00021-5

Publisher's Note