A Novel Approach to Process Discovery with Enhanced Loop Handling (Extended version)

No Author Given

No Institute Given

Abstract. Automated process discovery from event logs is a key component of process mining, allowing companies to acquire meaningful insights into their business processes. Despite significant research, present methods struggle to balance important quality dimensions: fitness, precision, generalization, and complexity, but is limited when dealing with complex loop structures. This paper introduces Bonita Miner, a novel approach to process model discovery that generates behaviorally accurate Business Process Model and Notation (BPMN) diagrams. Bonita Miner incorporates an advanced filtering mechanism for Directly Follows Graphs (DFGs) alongside innovative algorithms designed to capture concurrency, splits, and loops, effectively addressing limitations of balancing as much as possible these four metrics, either there exists a loop, which challenge in existing works. Our approach produces models that are simpler and more reflective of the behavior of real-world processes, including complex loop dynamics. Empirical evaluations using real-world event logs demonstrate that Bonita Miner outperforms existing methods in fitness, precision, and generalization, while maintaining low model complexity.

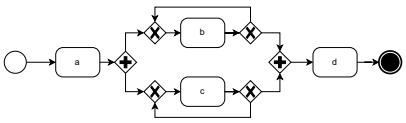
Keywords: Process Mining · Automated Process Discover · Event Log · BPMN · Loop Handling · Concurrency.

1 Introduction

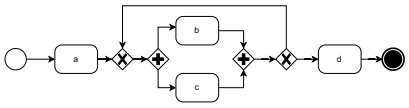
Process discovery [4] is one of the most prominent process mining techniques. It enables the automatic discovery of a process model that captures and explains the behavior recorded in event logs. The process model is typically represented in Petri nets [15], but other formats are also possible (e.g., in the standard Business Process Model and Notation - BPMN [9]). A discovered process model must accurately reflect the behavior observed in or inferred from the event log. Specifically, the process model should (i) parse the traces contained within the log, (ii) parse traces not present in the log but likely to belong to the process that generated the log, and (iii) refrain from parsing traces unrelated to the process. These properties are known as fitness, generalization, and precision, respectively. Furthermore, the model should be as simple as possible, a characteristic often quantified through complexity measures [8].

Despite extensive research [4,16], achieving a balance across the four key quality dimensions — fitness, precision, generalization, and complexity — remains a challenge. When applied to real-life event logs, most automated process discovery techniques, such as the Heuristics Miner [17] and its derivatives, often produce large, spaghetti-like models that are behaviorally incorrect (e.g., prone to deadlocks). Another state-of-the-art technique, the Inductive Miner [12], tends to generate block-structured models that are behaviorally sound with high fitness but exhibit low precision. Additionally, the SplitMiner approach [5] is capable of producing accurate and simple process models; however, these models become complex and unsound in the presence of loops, complicating the observed behavior in the event log and reducing the accuracy and structure of the results.

We provide a basic example to illustrate a limitation addressed by existing methods, shown in Figure 1. Consider an event log for the BPMN model represented in Figure 1b, where there exists a **loop** that includes the **parallel** blocks b and c. Current state-of-the-art algorithms are unable to accurately discover this BPMN pattern with a loop over parallel blocks. Instead, they produce a model like the one shown in Figure 1a, where these algorithms incorrectly represent b and c with individual self-loops. This misrepresentation significantly impacts the model's accuracy, reducing its usability for real-world process analysis.



(a) Discovered Model by Split Miner, Inductive Miner, Heuristics Miner.



(b) BPMN to Discover

Fig. 1: A simple example of the limitation of existing works.

To address this limitation and bridge the identified gap, this paper introduce Bonita Miner, a novel process discovery method to generate simpler process models while still achieving a balance among fitness, precision, and generalization. The approach leverages a novel Depth-First Algorithm (DFA) for constructing splits and joins, coupled with advanced filtering of the directly-follows graph (DFG) derived from event logs. These constructs effectively capture concurrency, choices, causal relationships, and even loops within the DFG. The DFA reduces model complexity while maintaining a balance among evaluation metrics, making the method suitable for processes of varying complexity, including those with loops. We also treat the loops as blocks (with one or multiple sources and one or multiple targets).

We empirically demonstrate that Bonita Miner surpasses three state-of-the-art baselines by evaluating its performance on BPIC real-life and synthetic loops

event logs using eight comprehensive metrics that measure the four key quality dimensions, even in the presence of complex loop structures.

The remainder of the paper is structured as follows. Section 2 reviews the existing process discovery methods. Section 3 introduces the Bonita Miner method, followed by Sections 4 and 5, which detail the proposed approach. Section 6 discusses the empirical evaluation of Bonita Miner, and finally, Section 7 concludes the paper and outlines potential directions for future research.

2 Related work

The section provides an overview of existing automated process discovery methods and discusses their limitations.

The α-algorithm [2] is a simple automated process discovery method based on the concept of DFG. While appealing due to its simplicity, the α-algorithm is not applicable to real-life event logs since it assumes the log to be complete and is too sensitive to infrequent behavior. The Heuristics Miner [18] addresses these limitations and consistently performs better in terms of accuracy on incomplete and noisy logs. To handle noise, the Heuristics Miner relies on a relative frequency metric between pairs of event labels. However, while this technique demonstrates higher fitness, it faces the challenge of lower precision.

Structured process models are generally more understandable than unstructured ones [10,11]. Moreover, structured models are sound, provided that the gateways at the entry and exit of each block match. Given these advantages, several algorithms have been designed to discover structured process models, represented for example as process trees. The Inductive Miner [13] uses a divide-and-conquer approach to discover process structured models, achieving high fitness. However, it tends to over-generalize the behavior observed in the log whenever the process model to be discovered is unstructured, generating undesired *flower models*[**?**].

The Structured Miner [3] addresses this limitation by relaxing the requirement of always producing a structured process model, in favor of achieving higher accuracy. Split Miner [5] is another method that addresses the balance between fitness, precision, and generalization. It focuses on filtering directly-follows graphs and discovering split gateways to produce simple, accurate, and deadlock-free process models. However, Split Miner is limited in handling loops effectively, which can lead to challenges in processes with loop-intensive behavior.

To reduce these challenges, an improved approach is proposed, which enhances the handling of loops while maintaining a balance across other quality dimensions. This approach ensures better adaptability to loop-intensive scenarios, thereby improving model quality and applicability.

3 Approach Overview

As mentioned in the introduction, the goal of the algorithm introduced in this paper is to ensure the balancing of key metrics even in the presence of loops. For

example, it successfully identifies the pattern represented in the BPMN model of Figure 1b, achieving a model accuracy of 1. The proposed approach decomposes the process discovery task in two main steps, as shown in Figure 2.

Graph filtering and analysis: taking an event log as input, this step first transforms it into a graph. Then, by analyzing both the graph and the event log, concurrent activities, choices, and part of the loops are removed from the graph. The benefits of processing these behaviors separately from the graph is twofold: it enables (i) handling complex loops (e.g., from one or multiple sources to one or multiple targets) and (ii) generating the BPMN model with a single-shot approach. with existing methods that require multiple iterations over the process model to achieve correct semantics, particularly for handling complex behaviors such as nested splits and joins.

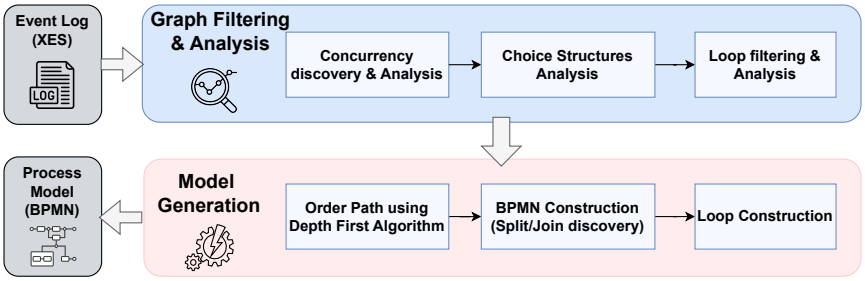


Fig. 2: Bonita Miner Overview

Model Generation: after filtering the graph to remove loops and concurrency, the next step involves ordering all paths in the graph using a depth-first algorithm. This approach ensures that joins are introduced alongside splits, constructing blocks of split/join gateways without enforcing strict block-structuredness. The resulting model is a BPMN diagram[1], where elements, including gateways (and nested gateways), are iteratively added based on insights from the earlier analysis. This step enables the construction of the BPMN diagram without loops in a single iteration, eliminating the need for post-processing or replacing gateways to achieve correct model semantics. Loops are constructed separately, employing a strategy that reduces complexity while maintaining a balance in quality metrics. This approach contrasts with existing methods, which typically handle loops alongside other tasks, resulting in increased complexity.

These steps enable the construction of a simplified model with low complexity, while achieving a high balance of fitness, precision, and generalization.

[1] This approach is not limited to BPMN; the same steps can also be applied to generate a Petri net.

4 Preliminary

In this section, we give some definitions related to event logs and process models that will be used in the subsequent sections.

A process model is a set of connected entities that formally represent process behavior. Since we use BPMN as the modeling language, entities can include flow objects (activities, events, and gateways), swimlanes (pools and lanes), artifacts (e.g., data objects, text annotations, or groups), and connecting objects (sequence flows, message flows, and associations). In this work, we limit the scope to entities shared among different languages, specifically flow objects represented by *activities*, *start/end events*, and *gateways* (including parallel and exclusive gateways), and connecting objects represented by *sequence flows*.

Definition 1 (Event Log). *Given a set of events $\mathcal{E}$, an event log $\mathcal{L}$ is a multiset of traces as $\mathcal{T}$, where a trace $t \in \mathcal{T}$ is a sequence of events $t = \langle e_1, e_2, \ldots, e_n \rangle$, with $e_i \in \mathcal{E}$, $1 \leq i \leq n$. Additionally, each event has a label $l \in \mathcal{L}$ and it refers to a task/activity executed within a process, we retrieve the label of an event with the function $\lambda : \mathcal{E} \to \mathcal{L}$, using the notation $\lambda(e) = e^l$.*

For the remaining, we assume all the traces of an event log have the same start event and the same end event. This is guaranteed by a simple preprocessing of the event log, to be compliant with the third of the 7PMG in [14], i.e., "use one start event for each trigger and one end event for each outcome".

Given the set of labels $\mathcal{L} = \{s, a, b, c, d, e, f, h, i, j, k, l, end\}$, a possible log is: $\mathcal{L} = \{\langle s, a, b, end \rangle^{10}, \langle s, a, c, e, f, h, d, i, j, k, l, end \rangle^{10}, \langle s, a, c, e, f, h, d, j, i, k, l, end \rangle^{10}, \langle s, a, c, f, e, h, d, i, j, k, l, end \rangle^{10}, \ldots\}$; this log contains more than 10 distinct traces, each of them recorded 10 times.

Starting from a log, we construct a DFG in which each arc is annotated with a frequency, based on the following definitions.

Definition 2 (Directly-Follows Graph). *Given an event log $\mathcal{L}$, its directly follows graph is defined as a directed graph $DFG = (V, E)$, where:*

- *V represents a finite set of vertices, each corresponding to an activity or start/end event, and having a unique label $l \in \mathcal{L}$.*
- *$E \subseteq V \times V$ represents a set of directed edges, where each edge $(u, v) \in E$ signifies that v can be executed right after the completion of u, theoretically $\forall(\lambda(a) = u \text{ and } \lambda(b) = v), \exists t = \langle e_1, e_2, \ldots, e_n \rangle \in \mathcal{T}$ where $a = e_i$ and $b = e_{i+1}$.*
- *$v_s \in V$ is the starting vertex, iff $\nexists(v, v_s) \in E$.*
- *$v_e \in V$ is an ending vertex, iff $\nexists(v_e, v) \in E$.*
- *For $e = (u, v) \in E$, we use the notations $e.source = u$ and $e.target = v$.*
- *For $v \in V$, the direct successors of v is $\text{Succ}_{DFG}(v) = \{u \in V \mid (v, u) \in E\}$ and the direct predecessors of is $\text{Pred}_{DFG}(v) = \{u \in V \mid (u, v) \in S\}$. When it is clear from context, the subscript DFG is omitted.*
- *For $(u, v) \in E$, we denote $\|u \to v\| = \sum |(u, v)|$ that represent the frequency of the edge from u to v, where $\forall(\lambda(a) = u \text{ and } \lambda(b) = v), \exists t = \langle e_1, e_2, \ldots, e_n \rangle \in \mathcal{T}$ where $a = e_i$ and $b = e_{i+1}$.*

– *For* $(u, v) \in E$*, we denote by short loop* $u \circlearrowleft v$ *where* $\exists(\lambda(a) = u$ *and* $\lambda(b) = v), \exists t = \langle e_1, e_2, \ldots, a, b, \ldots, b, a, \ldots, e_n\rangle \in \mathcal{T}$*.*

Let $DFG = (V, E)$ be a DFG. We define paths, cycles and branching relations are as follows:

Definition 3 (Path). *A path from* $u \in V$ *to* $v \in V$*, denoted as* $P_{u,v} = \langle(v_1, v_2), \ldots, (v_{n-1}, v_n)\rangle$*, is the sequence of unique edges leading from* u *to* v *where* $\forall 1 \leq i, j \leq n-1, i \neq j, n \geq 2$*,* $(v_i, v_{i+1}) \in E$*,* $v_1 = u, v_n = v$*, and* $\nexists(v_i, v_{i+1}) = (v_j, v_{j+1})$*. We denote by:*

– $P^v_{u,v} = \{v_1, \ldots, v_n\}$*: the set of the path vertices.*
– $P^s_{u,v} = (v_1, v_2)$*: the first edge of the path.*
– $P^e_{u,v} = (v_{n-1}, v_n)$*: the last edge of the path.*
– $\mathbb{P}_{u,v}$*: the possibly empty set of all paths from* u *to* v*.*

Definition 4 (Cycle). *A cycle* C_u *is a path* $P_{u,u} = \langle(u, v_1), \ldots, (v_n, u)\rangle$ *that starts and ends with* u *such that* $\forall 1 \leq i \leq n, v_i \neq u$*. We denote by* $\mathbb{C}_u$ *the possibly empty set of all cycles of* u*.*

Definition 5 (BPMN Process Model). *A BPMN process model is defined as* $M = (N, G, S)$*, where:*

– $N = V$ *is the set of vertices in* DFG*.*
– G *is a finite set of gateways. A gateway* $g \in G$ *can be parallel, represented by* $+$ *or exclusive represented by* $\times$*. We denote by* $type(g) \in \{\times, +\}$ *the function that returns the type of* g*.*
– $S \subseteq (N \times G) \cup (N \times G)$ *is a set of directed sequence flows. For each* $s = (u, v) \in S$*, we use the notations* $s.source = u$*, and* $s.target = v$*.*
– *For* $u \in N \cup G$*, the direct successors of* u *is* $\mathrm{Succ}_M(u) = \{v \in N \cup G \mid (u, v) \in S\}$ *and the direct predecessors is* $\mathrm{Pred}_M(u) = \{v \in N \cup G \mid (v, u) \in S\}$*. When it is clear from context, subscript* M *is omitted.*
– *For* $u \in N \cup G$*, the transitive successors of* u *is* $\mathrm{TSucc}(u) = \{v \in N \mid \exists g_1, \ldots, g_k \in G, (u, g_1), (g_k, v), (g_i, g_{i+1}) \in S\}$ *and the transitive predecessors of* u *is* $\mathrm{TPred}(u) = \{v \in N \mid \exists g_1, \ldots, g_k \in G, (v, g_1), (g_k, u), (g_i, g_{i+1}) \in S\}$*. For* $u \in N$*,* $\mathrm{TSucc}(u) = \mathrm{Succ}_G(u)$ *and* $\mathrm{TPred}(u) = \mathrm{Pred}_G(u)$*.*

5 Bonita Miner

In this section, we illustrate the details of our approach. We begin by running the example in Section 5.1, followed by filtering the concurrent activities in Section 5.2. Next, we identify the loops in Section 5.3. We then delve deeper into identifying the activities that represent transitivity after the gateways, including the identification of loop blocks in Section 5.4. This is followed by the construction of split and join patterns in the model in Section 5.5, and finally, the loop construction in the BPMN in Section 5.6.

5.1 Running Example

Figure 3 represents a BPMN process model. After activity (a), there are nested gateways: an XOR gateway connected to activity (b) and an AND gateway that connects to two activities, (c) and (d). Additionally, there are two nested join gateways before activity (h). Moreover, there is a loop on a parallel block encompassing activities (e) and (f). Since these patterns are present in the example, this model serves as a running example to illustrate the steps of the algorithm.

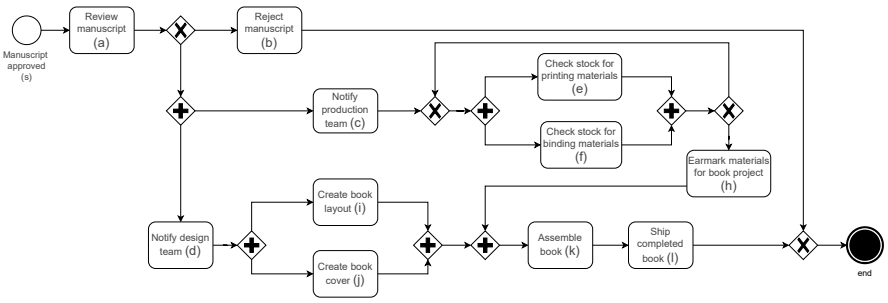


Fig. 3: The running example is used to explain the Bonita Miner workflow

5.2 Concurrency discovery

Identifying concurrency and filtering it effectively is crucial for simplifying the DFG. With a simplified DFG, we can extract branching relations and the dependent relations between activities. Additionally, concurrency discovery helps construct AND gateways (including both simple and nested gateways) during one-step generation. An event log is the input of our algorithm, and it can be converted into a DFG (Definition 5), as illustrated in Figure 4. To enhance the readability of the DFG, we add two arrows on each relation to represent the existence of two connections between a pair of activities. For example, $c \leftrightarrow d$ indicates that there exists a relation $c \rightarrow d$ as well as a relation $d \rightarrow c$. For the sake of clarity, we represent only a subset of these relations (we remove the parallelism that arises from the activities (e), (f), (h), (i), (j), and (k)). All the red relations in the DFG represent parallelism. The first step is to detect all concurrent relations, formally defined in Definition 6.

Definition 6 (Concurrent Relation). *Let an event log $\mathcal{L}$, a set of traces $\mathcal{T}$, and a directed graph* DFG $= (V, E)$ *be given. We postulate that a and b are concurrent if and only if the following two conditions hold:*

1. $\exists e, e' \in E$ *such that* $a, b \in V$ *and* $(a, b) = e$ *and* $(b, a) = e'$
2. $\exists t \in \mathcal{T}$ *such that* $\exists a, b \in t$

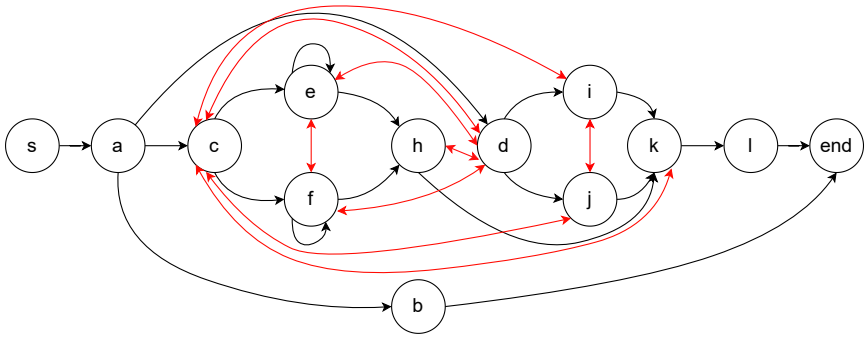


Fig. 4: Part of the DFG of the event log of the running example

After identifying all concurrent relations, we remove them except for those representing short loops, as we are focusing on parallelism in this step. This step is very important for filtering the graph and subsequently identifying the gateway type in the next step. In the following definitions and sections, we use the term DFG to refer to a filtered graph that excludes concurrency relations (see Figure 5).

Branching parallel relations are those derived from concurrency relations that share the same predecessor. This characteristic helps in detecting parallel split gateways (AND gateways), particularly by identifying the activities that will transitively follow a gateway. For example, $\{c, d\}$ in the filtered DFG shares the same predecessor (a); hence, it is added to the set of branching parallel relations. However, other relations, such as $\{c, e\}$, are not included because they do not have the same predecessor. As a result, the branching parallel relations are: $\{\{c, d\}, \{i, j\}, \{e, f\}\}$.

Formally, the branching parallel relations is defined as follow:

Definition 7 (Branching Parallel Relations). *Let* $\text{DFG}' = (V', E')$ *be the filtered DFG, where concurrency relations have been removed except for short loops. The set of branching parallel relations is denoted as:*

$$BP = \{\{a, b\} \mid a, b \text{ are concurrent and } \text{Pred}_{\text{DFG}'}(a) = \text{Pred}_{\text{DFG}'}(b)\}.$$

5.3 Loop filtering

Although loops contribute to increased complexity and reduced readability in a process model, we treat loops in our approach as blocks, which may have one or multiple sources and one or multiple targets. This treatment and construction of loops as blocks help to reduce complexity significantly. This step involves removing the loops from the *DFG* by detecting and filtering the dependency relations in E that break the cycles which results in an acyclic DFG. For example, given the DFG in Figure 5, there are four cycles $C_{e1} = \langle (e, e) \rangle$, $C_{e2} = \langle (e, f), (f, e) \rangle$, $C_{f1} = \langle (f, f) \rangle$ and $C_{f2} = \langle (f, e), (e, f) \rangle$.

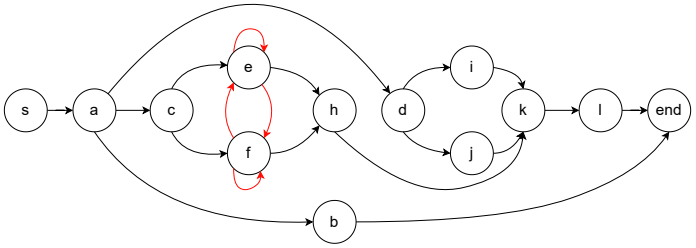


Fig. 5: DFG, after filtering the parallelism, of the running example with cycles

Breaking cycles involves removing specific edges, known as *looping edges*, which are the last edges in the paths that create the looping behavior. For example, given the cycle $C_{e2} = \langle (e, f), (f, e) \rangle$, the edge (f, e) creates the loop. It is characterized by the property that, following the path from the graph's start to the edge's source vertex, no other vertices in the cycle are revisited before reaching this edge. In this example, the looping edges are (e, e), (f, e), (f, f), and (e, f). Thus, all the looping edges are removed to break the loops.

The DFG in Figure 6 provides another example. The cycles in this example are $C_a^1 = \langle (a, b), (b, c), (c, a) \rangle$, $C_c^1 = \langle (c, a), (a, b), (b, c) \rangle$, and $C_b^1 = \langle (b, c), (c, a), (a, b) \rangle$. To break the loop, following the path from the graph's start to the edge's source vertex of the cycles, the looping edge is characterized by the property that no other vertices in the cycle are revisited before reaching this edge. Applying the path condition in Definition 8, only C_a^1 is valid. Therefore, $L = \{(c, a)\}$.

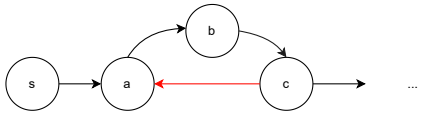


Fig. 6: An example of a long loop.

These are formalized in Definition 8 as follows:

Definition 8 (Looping edge). *Let C_u be a cycle of $u \in V$. The looping edge e^{loop} is an edge $e \in C_u^E$ such that $P^V_{v_s, C_u^S.source} \cap \left(C_u^V \setminus \{C_u^S.source\}\right) = \emptyset$.*

After filtering all the looping edges, an acyclic DFG, denoted as DFG_A, is obtained and used in the next step to discover the split and join gateways. The result after extract looping edge is the removal of the red edges from the Graph in the Figures 5 and 6.

5.4 Synthesis of Control Flow Elements

In the following subsections, we illustrate the analysis of choices, concurrency, and loops to develop a specific format that aids in the construction phase of the BPMN. This format is designed to detect gateway types, especially in the presence of nested gateways, and to preprocess loops by constructing blocks before adding them to the BPMN.

Exclusive & Concurrency Relations Analysis In this step, we extract all the choices between the activities, where the choices can be represented as branching without the existence of parallelism (Definition 7). For example, in Figure 4, after (a), there are three activities: (b), (c), and (d). Since (c) and (d) are in parallel, we can deduce that (b) and (c), as well as (b) and (d), are in choice due to the transitivity of an XOR gateway. These are formalized in Definition 9 as follows:

Definition 9 (Branching Exclusive Relations). *Let BP be the set of Branching Parallel relation. The BE_u is a set of exclusive branching after $u \in V$, where $\forall\{a,b\} \in BE_u, \exists(u,a) \in E$ and $(u,b) \in E$ where $a \neq b$ and $\{a,b\} \notin BP$.*

We denote $BE = \bigcup_{v \in V} BE_v$ to represent all the branching choice in the DFG_A.

After extracting the branching exclusive relations, we analyze these relations to explore transitivity after a gateway between the activities and merge certain branching relations. This helps our algorithm detect the nesting gateways in our main algorithms. The following Algorithm 1 represents the process of merging these relations. Before starting, we arrange the subsets of BE into a list based on the most frequent vertices in the subsets. This step is crucial to account for all possible relations, especially in cases involving multiple nested gateways (e.g., an XOR gateway followed by an AND gateway, which is then followed by another XOR gateway). Starting with the most frequent vertices ensures that all gateways are detected effectively.

Algorithm 1 Merge Exclusive

Input: Acyclic DFG DFG_A, Branching Exclusive Relation BE
Output: Merged Exclusive denoted by $\#$
create a empty set $\#$
for all $b \in BE$ **do**
  for all $b' \in BE \setminus \{b_1\}$ **do**
    if $b \cap b' \neq \emptyset$ **then**
      if $Succ(b.source) \cap Succ(b.target) \cap Succ(b'.source) \cap Succ(b'.target) \neq \emptyset$ **then**
        $\# \leftarrow \# \cup \{\{b.source, b.target, b'.source, b'.target\}\}$
  $BE \leftarrow BE \setminus \{b\}$

For example, from the DFG_A in Figure 5, $BE = \{\{b, c\}, \{b, d\}\}$. Since b is the command vertex (line 6) and all the vertices b, c, and d share the same successor a (line 7), the result is $\# = \{\{b, c, d\}\}$. We denote $\#(v_1, v_2)$ is the smaller subset that contains $v_1 \in V$ and $v_2 \in V$.

Similar to the Merge Exclusive (Algorithm 1), the Merge Parallel is based on the Branching Parallel Relations (Definition 7). The same algorithm is then applied to merge the parallel relations, and we denote the result by $\|$. After applying the algorithm, $\| = \{\{c, d\}, \{i, j\}, \{e, f\}\}$. We denote $\|(v_1, v_2)$ is the smaller subset that contains $v_1 \in V$ and $v_2 \in V$, and $\|(v)$ to represent all the subset that contain v.

Loop Structures Analysis After extracting loops (as described in Definition 8), we preprocess the loops to form blocks of loops, which helps reduce unnecessary gateways and relations used to represent the loops. To achieve this, two merge algorithms are applied to combine the loops based on their sources and targets. These algorithms assist in identifying the start and end points of the block where the loop should be constructed.

The "Merge Loop by sources" algorithm 2 is designed to merge loops based on an acyclic dependency graph based on their structure and relationships, and based on the looping edge defined in 8. This merge groups loops that share the same source and whose targets have common successors (line 7-9). In the example, the looping edge $L = \{(f, f), (e, e), (f, e), (e, f)\}$, and the result from this algorithm is $ML_s = \{(f, \{f, e\}), (e, \{e, f\})\}$, which is the input of the second algorithm 3.

Algorithm 2 Merge Loop by sources

1: **Input:** Acyclic DFG DFG_A, looping edge L
2: **Output:** Merged loops ML_s
3: create a list ML_s
4: **for all** $l \in L$ **do**
5: create a set $targets \leftarrow \{l.target\}$
6: **for all** $l' \in L \setminus \{l\}$ **do**
7: **if** $l.source == l'.source \wedge \text{Succ}_{DFG_A}(l.target) == \text{Succ}_{DFG_A}(l'.target)$ **then**
8: $targets \leftarrow targets \cup \{l'.target\}$
9: $L \leftarrow L \setminus \{l'\}$
10: $ML_s \leftarrow ML_s \cup (l.source, targets)$
11: $L \leftarrow L \setminus \{l\}$

The "Merge Loop" by targets algorithm 3 is designed to consolidate loops that have already been grouped based on their sources. The goal is to further merge these loops by examining their targets and associated predecessors. This merge ML, is derived from ML_s by combining sources if they have common predecessors (lines 7-9). In the example, the extracted $ML_s = \{(f, \{f, e\}), (e, \{e, f\})\}$,

and the result from this algorithm is $ML = \{(\{f, e\}, \{f, e\})\}$. This indicates the existence of a loop for the entire block of $\{e, f\}$.

Algorithm 3 Merge Loop

```
1: Input: Acyclic DFG DFG_A, Loop merged by sources ML_s
2: Output: Merged loops ML
3: create a list ML
4: for all l ∈ ML do
5:    create a set sources ← {l.source}
6:    for all l' ∈ ML_s \ {l} do
7:       if l.target == l'.target ∧ Pred_DFG_A(l.source) == Pred_DFG_A(l'.source)
   then
8:          sources ← sources ∪ {l'.source}
9:          ML_s ← ML_s \ {l'}
10:   ML ← ML ∪ (sources, l.target)
11:   ML_s ← ML_s \ {l}
```

5.5 Split & Join Discovery

Accurately discovering split and join gateways while keeping the model simple is challenging. To simplify model generation, we construct blocks of split/join gateways without enforcing the block-structuredness property. Algorithm 4 outlines the high-level steps. It takes as input the acyclic DFG DFG_A and the branching parallel relations $\|$ and returns as output a BPMN model M.

Initially, we traverse all paths in DFG_A from start to all possible ends, ordering them with a depth-first algorithm based on the longest path. This ensures joins are introduced alongside splits, focusing on identifying the maximum number of gateways by analyzing the longest paths first. The result is a list of ordered edges derived from these paths (Line 5). In our example of Fig. 5 (excluding the red looping edges that have been filtered), the ordered depth-first traversal returns the paths ordered $[P_1, P_2, P_3, P_4, P_5]$ where $P_1 = \langle(s, a), (a, c), (c, e), (e, h), (h, k), (k, l), (l, end)\rangle$, $P_2 = \langle(s, a), (a, c), (c, f), (f, h), (h, k), (k, l), (l, end)\rangle$, $P_3 = \langle(s, a), (a, d), (d, i), (i, k), (k, l), (l, end)\rangle$, $P_4 = \langle(s, a), (a, d), (d, j), (j, k), (k, l), (l, end)\rangle$, and $P_5 = \langle(s, a), (a, b), (b, end)\rangle$. The ordered list of unique edges E_O extracted from these paths, respecting their order, is $E_O = [(s, a), (a, c), \ldots, (c, f), (f, h) \ldots]$.

All the edges of P_1 are added in temporal order in the process model. Lines 9-11 add the source and target elements that are not already added to M. From P_2, the dependency (c, f) should be added, where c already exists and has a relation. Therefore, the split algorithm is called according to line 13 (detailed in Section 5.5, Figure 7a). This is followed by (f, h), where the join algorithm is invoked as specified in line 15 (detailed in Section 5.5, Figure 8a). From path P_3, for (a, d), the split algorithm is called (Figure 7b). This is followed by (d, i),

Algorithm 4 BPMN Construction

```
1: Input: Acyclic DFG DFG_A = (V, E), parallel vertices ||, exclusive vertices #
2: Output: Generated BPMN Model M = (N, G, S)
3: Initialize M
4: Create a gateway reference GR = null
5: E_O = OrderedDF(DFG_A)
6: for all e ∈ E_O do
7:    source ← e.source
8:    target ← e.target
9:    if ∄s', s'' ∈ S | s'.source = source ∧ s''.target = target then
10:      N ← N ∪ {source, target}
11:      S ← S ∪ {(source, target)}
12:   else if ∃s ∈ S | s.source = source ∧ target ∉ N then
13:      call AddSplitGateways(DFG_A, M, ||, #, source, target, GR)
14:   else
15:      call AddJoinGateways(M, source, target, GR)
```

which is added in line 13. Then, the join algorithm is called to add the relation between (i, k). For P_4, (d, j) is handled by calling the split algorithm, followed by the join algorithm for (j, k) (Figure 8b). Finally, from P_5, (a, b) is added by calling the split algorithm, and (b, end) is added directly (lines 9-11).

Split Operation The split operation in Algorithm 5 has two parts. The first adds a simple gateway (Figure 7a). The second, a recursive operation, nests gateways (Figure 7b).

(a) a simple split (b) a nested split

Fig. 7: Add split gateways to BPMN.

Figure 7a illustrates the addition of a new simple gateway with the dependency relation (c, f). Since c is an existing source with a relation in the process model (e), the split algorithm is applied. The first step involves checking the successor of c, which is e (lines 3-7). If the successor is an activity, a split gateway is added, with c as the incoming connection and e and f as the outgoing connections. To determine the gateway type, the relation between e and f is checked. This relation must exist within the smaller subset of # and ||. Based on the results, if the relation is in ||, the gateway will be +; otherwise, it will be $\times$. If the successor is a gateway, the advanced split algorithm is called to add a nested gateway (lines 9-10).

Algorithm 5 AddSplitGateways

1: **Input:** Acyclic DFG DFG_A, Process Model M, parallel vertices $\|$, exclusive vertices $\#$, Entity $source$, Entity $target$, gateway reference GR
2: $target_M \leftarrow s.target \mid \exists s = (source, target_M) \in S$
3: **if** $type(target_M) \notin G$ **then**
4: $g_{new} \leftarrow Gateway(\text{“}+\text{”})$ if $\exists\{target_M, target\} \in \| \ \wedge \ (\#(target_M, target) = \emptyset$ or $\|\|(target_M, target)\| < \|\#(target_M, target)\|)$ else $Gateway(\text{“}\times\text{”})$
5: $G \leftarrow G \cup \{g_{new}\}$
6: $S \leftarrow S \setminus \{(source, g_{current}\}$
7: $S \leftarrow S \cup \{(source, g_{new}), (g_{new}, target), (g_{new}, target_M)\}$
8: **else**
9: $s \leftarrow (source, target_M)$
10: $AddNestingSplitGateway(DFG_A, M, \|, \#, GR, s, source, target)$

The advanced algorithm in Algorithm6 is utilized to add a split gateway after the source, which can result in nested gateways. The algorithm involves three main steps: (i) adding the target as a successor to the existing gateway (lines 4-6), (ii) inserting a new split gateway before the existing gateway (line 7-18), and (iii) adding a new gateway after the existing one using the nesting split algorithm (lines 20-24).

For example, in Figure 7b, the dependency relation is (a, b). The recursive algorithm begins by identifying the successor of a, which is the gateway (1). Next, all successors with the transition (3) are searched, resulting in the extracted list $\{c, d\}$. The relationship between the target b and the extracted list $\{c, d\}$ is then analyzed. Since there is no parallelism in $\|$ that includes (b, c) or (b, d), it is concluded that b is in an exclusive decision with these activities: $(b, c) \Rightarrow$ exclusive; $(b, d) \Rightarrow$ exclusive. Because this relation is exclusive and not of the same type as (3), a new gateway (7) is added before the existing gateway (lines 13-18).

Algorithm 6 AddNestingSplitGateway

Input: Acyclic DFG DFG_A, Process Model M, parallel vertices $\|$, exclusive vertices $\#$, gateway reference GR, gateway type $gatewayType$, current relation s, Entity source $source$, Entity target $target$
$g_{current} \leftarrow s.target$
$succT_N \leftarrow SuccT_M^N(g_{current})$
if $succT_N \in ||(target) \ \wedge \ type(g_{current}) =$ " + " **then**
  $S \leftarrow S \cup \{(g, target)\}$
  $GR \leftarrow g_{current}$
else if $succT_N \in ||(target) \ \wedge \ type(g_{current}) =$ " × " **then**
  $g_{new} \leftarrow Gateway($" + "$)$
  $G \leftarrow G \cup \{g_{new}\}$
  $S \leftarrow S \setminus \{(source, g_{current}\}$
  $S \leftarrow S \cup \{(source, g_{new}), (g_{new}, g_{current}), (g_{new}, target)\}$
  $GR \leftarrow g_{new}$
else if $succ_N \cap ||(target) \leftarrow \emptyset$ **then**
  $g_{new} \leftarrow Gateway($" × "$)$
  $G \leftarrow G \cup g_{new}$
  $S \leftarrow S \setminus \{(source, g_{current}\}$
  $S \leftarrow S \cup \{(source, g_{new}), (g_{new}, g_{current}), (g_{new}, target)\}$
  $GR \leftarrow g_{new}$
else
  $succ_N \leftarrow Succ_M^N(g_{current})$
  for all $v \in succ_N$ **do**
    if $v \in G$ **then**
      $s \leftarrow (g_{current}, v)$
      $AddNestingSplitGateway(DFG_A, M, \|, \#, GR, s, source, target)$

Join Operation Similar to the split operation, the join operation presented in Algorithm 7 consists of two main parts: adding a simple join gateway and adding a nested join gateway.

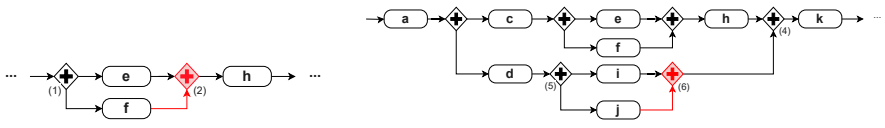
Algorithm 7 AddJoinGateways

Input: Process Model M, gateway reference GR, Entity $source$, Entity $target$
$source_M \leftarrow s.source \mid \exists s = (source_M, target) \in S$
if $type(source_M) \notin G$ **then**
  $g_{new} \leftarrow Gateway(type(GR))$
  $G \leftarrow G \cup \{g_{new}\}$
  $S \leftarrow S \setminus \{(source, g_{current}\}$
  $S \leftarrow S \cup \{(source, g_{new}), (source_M, g_{new}), (g_{new}, target)\}$
else
  $s \leftarrow (source_M, target)$
  $AddNestingJoinGateway(M, GR, s, source, target)$

Figure 8a illustrates the addition of a simple join. In this example, the dependency relation is (f, h). Since f is a source already added and h is a target already incorporated into the process model, the join algorithm is applied. The first step involves checking the predecessor of h, which is e (lines 3-7). As the predecessor is an activity, a join gateway is directly added after h. The incoming connections to this gateway are e and f, and the outgoing connection is h (lines 9-10). The type of the gateway matches the last gateway added into the BPMN.

The advanced algorithm in Algorithm 8 adds a join gateway when a gateway already exists before the target, potentially resulting in nested gateways. It involves three main steps: (i) adding the target as the predecessor of the existing gateway (lines 3-5), (ii) adding a new join gateway after the existing gateway (lines 6-31), and (iii) recalling the complex join algorithm to add a new gateway before the existing one (line 32).

In Figure 8b, the dependency relation is (j, k), with the last added split being (5). The algorithm checks for a backward path from the selected gateway (4) that intersects the split gateway (5). If all predecessors intersect the split gateway, the temporal relation between the gateways is validated. If they are of the same type, a new relation is added from the target to the selected gateway; otherwise, a new gateway is added after the selected gateway. If at least two predecessors have a relation, a new join gateway matching the split gateway's type is added after these predecessors and before the selected gateway. Otherwise, the nesting join algorithm is applied to all predecessor gateways. In this example, (4) has h and i as predecessors, with a backward path from i to (5). Therefore, a new join gateway of the same type as (5) is added after i (lines 23-30).



(a) a simple join (b) a nested join

Fig. 8: Add join gateways to BPMN.

Algorithm 8 AddNestingJoinGateway

Input: Process Model M, gateway reference GR, Entity $source$, Entity target $pred$, Entity $target$, current relation s
$pred_M \leftarrow Pred_M(pred)$
if $\forall v \in pred_M, \exists P_{GR,v}$ **then**
  if $type(pred) == type(GR)$ **then**
    $S \leftarrow S \cup \{(source, pred)\}$
  else
    $g_{new} \leftarrow Gateway(type(GR))$
    $G \leftarrow G \cup \{g_{new}\}$
    $S \leftarrow S \setminus \{s\}$
    $S \leftarrow S \cup \{(source, g_{new}), (pred, g_{new}), (g_{new}, s.target)\}$
else if $\forall v \in pred_M, \nexists P_{GR,v}$ **then**
  $g_{new} \leftarrow Gateway(type(GR))$
  $G \leftarrow G \cup \{g_{new}\}$
  $S \leftarrow S \setminus \{s\}$
  $S \leftarrow S \cup \{(source, g_{new}), (pred, g_{new}), (g_{new}, s.target)\}$
else if $\|\forall v \in pred_M, \exists P_{GR,v}\| \geq 2$ **then**
  $g_{new} \leftarrow Gateway(type(GR))$
  $G \leftarrow G \cup \{g_{new}\}$
  for all $v \in pred_M$ **do**
    $S \leftarrow S \setminus \{(v, pred)\}$
    $S \leftarrow S \cup \{(v, g_{new})\}$
  $S \leftarrow \{(g_{new}, pred)\}$
else
  $pred_{new} \leftarrow Pred_M(v) | v \in pred_M$
  $s = (pred_{new}, pred) \in S$
  if $type(pred_{new}) \in G$ **then**
    $g_{new} \leftarrow Gateway(type(GR))$
    $G \leftarrow G \cup \{g_{new}\}$
    $S \leftarrow S \setminus \{s\}$
    $S \leftarrow S \cup \{(source, g_{new}), (pred_{new}, g_{new}), (g_{new}, pred)\}$
  else
    call $AddNestingJoinGateway(M, GR, source, pred_{new}, target, s)$
$GR \leftarrow null$

5.6 Loop Construction

We start in the loop construction by the block construction. During the generation of the BPMN, it is not possible to establish all relationships because the loop relation is removed, resulting in some incomplete blocks. For example, after removing the entities in red in Figure 9, the result is contracted before adding the loop. In the loop block construction algorithm (Algorithm 9), the goal is to complete the loop block (gateway (3)) before adding the loops as described in algorithm 10.

Since in the example used in the paper, the blocks of e, f are completed where the source block is gateway (1) and the target block is gateway (2) (cf. Figure

Algorithm 9 Loop Block Construction

1: **Input:** Acyclic DFG DFG_A, Process Model $M = (N, G, S)$, Merged loops ML, parallel vertices $\|$, exclusive vertices $\#$
2: **for all** $l = (sources, targets) \in ML$ **do**
3: **if** $\|sources\| \neq 1 \ \wedge \ \nexists (u, v) \in S \mid (v \in N \ \wedge \ u \in sources)$ **then**
4: $g_{split} \leftarrow LCA_M(sources)$
5: $G \leftarrow G \cup \{g_{join}\} \mid type(g_{join}) = type(g_{split})$
6: $S \leftarrow S \cup \{(s, g_{join}\} \mid s \in sources$
7: **if** $\|targets\| \neq 1 \ \wedge \ \nexists (v, u) \in S \mid (v \in N \ \wedge \ u \in targets)$ **then**
8: $N \leftarrow N \cup \{temp\}$
9: **for all** $t \in targets$ **do**
10: call $AddSplitGateways(DFG_A, M, \|, \#, temp, t, \emptyset)$
11: $N \leftarrow N \setminus \{temp\}$

10), we add other example (cf. Figure 9) to well explain the algorithm, where only one loop exists, which is $(\{e, f\}, \{b\})$. Since there are two sources, $\{e, f\}$, it is necessary to validate the completion of their block (line 3). If the block is completed, no action is required; otherwise, the block should be constructed based on the split. To accomplish this, by applying the lowest common ancestor algorithm [6] on the process model, the split gateway can be identified as the last ancestor e and f (line 4). Subsequently, a new join gateway should be added with the same type of split(lines 5-6).

Next, the targets are checked to determine if they construct all the blocks. If more than one target exists, the existence of gateways before the targets is validated (line 7). If gateways do not exist, a temporary activity is added to construct this block by adding connections and applying the split operation from the temporary activity to the targets (line 8-10). Finally, this activity is removed (line 11), and the process continues to the final algorithms to add loops to the process model 10.

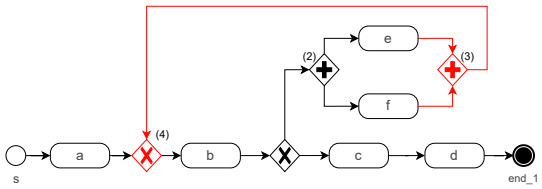


Fig. 9: Complete a loop block and add it to the BPMN model.

The final step involves adding the loop between the source and target blocks presented in Algorithm 10. A block is considered an activity if there is one element for the source or target, and it is considered a gateway if there is more than one source or target. The process iterates through all the Merged Loop

results, leading to four scenarios: (i) If both source and target blocks have a sequence flow to other entities, a new gateway is added after the source block and another before the target block. These two gateways are then connected (line 7). (ii) If the source block has a sequence flow but the target block does not, a new gateway is added after the source block, followed by a connection from the new gateway to the target block (line 9). (iii) If the target block has a sequence flow but the source block does not, a new gateway is added before the target block, followed by a connection from the source block to the gateway (line 12). (iv) If neither the source nor the target block has a sequence flow, a simple connection is created between these blocks (line 14).

Algorithm 10 Loop Operation

1: **Input:** Process Model M, merge loops ML
2: **for all** $l \in ML$ **do**
3: $sourceBlock \leftarrow \bigcap\{Succ_M(s) | s \in l_s\}$ if $|l_s| > 1$ else l_s
4: $targetBlock \leftarrow LCA_M(l_t)$ if $|l_t| > 1$ else l_t
5: **if** $\exists(sourceBlock, s) \in S$ **then**
6: **if** $\exists(s, targetBlock) \in S$ **then**
7: $AddTwoLoopGateways(sourceBlock, targetBlock)$
8: **else**
9: $AddLoopAfterSourceBlock(sourceBlock, targetBlock)$
10: **else**
11: **if** $\exists(s, targetBlock) \in S$ **then**
12: $AddLoopBeforeTargetBlock(sourceBlock, targetBlock)$
13: **else**
14: $S \leftarrow S \cup \{(sourceBlock, targetBlock)\}$

In the paper example, Figure 10 represents the loop operation where the $\{e, f\}$ block is in a loop, meaning this block is both the source and the target, with a gateway as both the source and target blocks. The source block is gateway (2), and the target block is gateway (1). These blocks are connected; therefore, new gateways are added, one after the source (8) and another before the target (9), and a connection from (8) to (9) is added (line 14 in the algorithm).

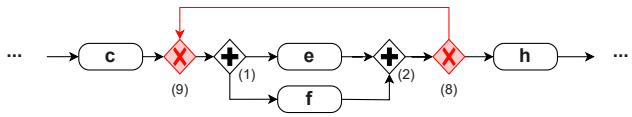


Fig. 10: Add a loop to the process model.

In other case (Figure 9), after constructing the loop block (adding the gateway (3)), the source block is disconnected, but b which is the target block is

connected to other entities. Therefore, a new × gateway (4) is added before b, with a connection from (3) to (4) (lines 11-12 in the algorithm).

6 Evaluation

We implemented Bonita Miner (hereafter BM) as a standalone Java application [2]. The tool takes as input an event log in XES format and produces a BPMN process model as output. Using this implementation, we conducted an empirical comparison of BM against three existing methods using a set of publicly available logs.

6.1 Datasets

We used a collection of real-life event logs, including logs from the annual Business Process Intelligence Challenge (BPIC), as well as other logs such as the Road Traffic Fines Management Process (RTFMP) and the SEPSIS Cases log as presented in Table 1. For the BPIC logs, we applied a filtering method to remove infrequent traces (5%) before applying each of the discovery methods. The dataset is heterogeneous in the number of traces (44 to 150,370), in the number of distinct traces (15 to 2023), in the number of event classes (11 to 55) and in the trace length (2 to 185 events).

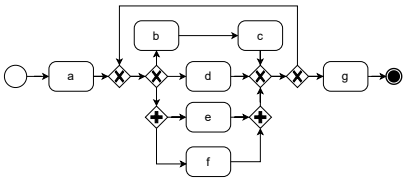
Model	Total traces	Distinct traces	Total events	Distinct events	Trace length		
					Min	Avr	Max
$BPIC2012_f$	9333	612	95348	23	3	10	55
$BPIC2013_{cp_f}$	1236	35	4343	4	2	3	13
$BPIC2013_{inc_f}$	5620	178	31740	4	2	5	18
$BPIC2013_{op_f}$	693	30	1569	3	1	2	9
$BPIC2015_{1_f}$	44	15	473	55	2	10	38
$BPIC2015_{2_f}$	8	4	88	15	7	11	13
$BPIC2015_{3_f}$	76	16	650	51	3	8	38
$BPIC2015_{4_f}$	7	3	73	14	4	10	13
$BPIC2015_{5_f}$	6	3	64	18	5	10	15
$BPIC2017_f$	17602	2023	501041	24	10	28	63
$BPIC2017_{o_f}$	40754	6	184037	7	3	4	5
$BPIC2020_{DD_f}$	9899	8	51479	10	1	5	9
$BPIC2020_{ID_f}$	5989	293	65739	32	3	10	24
$BPIC2020_{PL_f}$	5972	385	65575	44	3	10	30
$BPIC2020_{PTC_f}$	1991	94	17040	29	1	8	16
$BPIC2020_{RP_f}$	6521	11	34328	11	1	5	11
$RTFMP$	150370	231	561470	11	2	4	20
$SEPSIS$	1050	846	15214	16	3	14	185

[2] Available at: `https://github.com/anonymous4conferences/BonitaMiner`

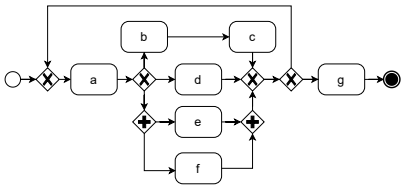
Table 1 continued from previous page

Table 1: Statistics of the event logs employed

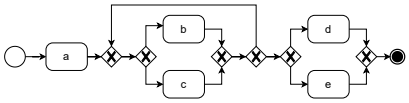
In addition to these logs, we generated event logs for the BPMN models containing loops, as shown in Figure 11, which are the focus of this paper. These patterns are used for different loop block, like multi source to multi target in BPMN (L1) 11a, or multi source one target in BPMN (L2) 11b, also the representation of the Block loops followed by split like in L3 11c and L4 11d.



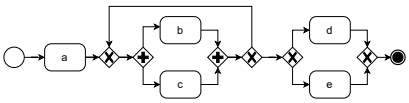
(a) (L1): BPMN with loop block, the loop is for a block that contains nesting gateways and parallelism.



(b) (L2): BPMN with loop block, the loop is from different sources to one target that contains nesting gateways and parallelism.



(c) (L3): Loops block on XOR gateway followed by split gateway.



(d) (L4): Loops block on AND gateway followed by split gateway.

Fig. 11: Different Loops Patterns

6.2 Experimental Setup

We selected three state-of-the-art discovery methods as baselines: Inductive Miner (IM), Heuristics Miner (HM), and Split Miner (SM). IM and HM were implemented using the PM4PY library [7].

For the evaluation of default parameters, we assessed the quality of the produced models across various quality dimensions. These included fitness, precision, and F-score as proxies for accuracy, as well as generalization, size, control-flow complexity (CFC), and structuredness (struct.) as proxies for complexity [1], along with execution time in seconds, excluding the time required to load the event log into memory. The size metric represents the number of elements in the process model, such as nodes (tasks, events, gateways) and edges, while CFC measures the structural complexity of the control-flow, considering constructs

like splits and joins (XOR, AND). Each metric was computed on BPMN models, except for fitness, precision, and generalization, which were measured on Petri nets due to the limitations of the measuring tools, which work exclusively on Petri nets. We converted BPMN models to Petri nets using the PM4PY library and also used PM4PY to calculate fitness, precision, and generalization.

All experiments were conducted on a system equipped with an 11th Gen Intel Core i7 processor running at 2.30 GHz, paired with 16 GB of RAM.

6.3 Results & Discussion

The evaluation of BM revealed its superior performance across multiple dimensions. The results, summarized in Table 2, show that BM consistently outperformed baseline methods across synthetic and real-life datasets. For instance, BM achieved perfect fitness (1.00) on most synthetic logs (L1–L4), indicating its robustness in reproducing observed behaviors. In real-life logs such as BPIC 2017, BM maintained competitive fitness (0.93) and F-score (0.94), striking a commendable balance between accuracy and simplicity.

Log name	Discovery Method	Distinct traces			Generalization	Complexity			Time(s)
		Fitness	Precision	F-score		Size	CFC	Struct.	
L1	BM	**1.00**	0.91	**0.95**	**0.97**	**15**	**6**	**1.00**	0.091
	SM	0.89	0.87	0.88	0.95	23	13	0.65	**0.01**
	IM	0.78	**1.00**	0.88	0.88	39	30	0.103	0.19
	HM	**1.00**	0.6	0.75	0.95	31	21	1.00	0.21
L2	BM	**1.00**	0.96	**0.98**	**0.97**	**15**	**6**	**1.0**	**0.05**
	SM	0.81	**1.00**	0.90	0.95	**15**	7	0.80	0.09
	IM	0.96	0.96	0.96	0.89	17	12	0.176	0.24
	HM	**1.00**	0.96	**0.98**	0.96	**15**	**6**	**1.0**	0.24
L3	BM	**1.00**	0.99	**0.99**	**0.96**	**12**	**5**	**1.0**	**0.053**
	SM	0.82	0.98	0.90	0.95	15	7	1.0	0.06
	IM	0.79	**1.00**	0.88	0.94	21	11	0.14	0.16
	HM	**1.00**	0.84	0.91	0.95	19	11	1.00	0.15
L4	BM	**1.00**	0.99	**0.99**	**0.97**	**12**	**4**	**1.0**	0.09
	SM	**1.00**	0.84	0.91	0.96	15	7	**1.0**	**0.06**
	IM	0.79	**1.00**	0.88	0.95	21	10	0.14	0.17
	HM	**1.00**	0.84	0.91	0.96	15	7	**1.00**	0.16
$BPIC2012_f$	BM	0.93	**0.82**	**0.87**	0.85	**48**	**34**	0.67	4.2
	SM	0.82	**0.82**	0.82	0.86	95	62	0.32	**1.46**
	IM	**1.00**	0.28	0.44	**0.96**	86	58	**1.00**	3.69
	HM	0.98	0.36	0.52	0.92	49	45	-	3.29
$BPIC2013_{cp_f}$	BM	0.91	0.88	0.89	0.89	**12**	**6**	**1.00**	0.18
	SM	0.85	**0.94**	0.90	0.88	**12**	**6**	**1.00**	0.04
	IM	**1.00**	0.63	0.77	**0.91**	18	11	1.00	0.27
	HM	**1.00**	0.90	**0.94**	0.89	11	6	**0.73**	**0.28**
$BPIC2013_{inc_f}$	BM	0.87	0.98	0.92	0.90	**13**	11	-	0.59

Table 2 continued from previous page

	SM	0.98	**0.99**	**0.98**	**0.95**	16	**10**	**1.00**	**0.08**
	IM	**1.00**	0.61	0.76	0.94	16	**10**	**1.00**	1.9
	HM	**1.00**	0.96	0.98	0.94	**13**	**10**	0.85	2.05
$BPIC2013_{op_f}$	BM	0.98	**0.99**	**0.99**	**0.93**	**11**	**7**	0.64	0.03
	SM	0.83	0.96	0.89	**0.93**	13	**7**	**1.00**	**0.02**
	IM	**1.00**	0.75	0.86	0.91	23	16	**1.00**	0.13
	HM	**1.00**	0.97	**0.99**	0.90	11	9	0.91	0.11
$BPIC2015_{1_f}$	BM	0.82	**0.91**	**0.87**	0.5	**90**	**34**	0.36	**0.03**
	SM	0.77	0.85	0.81	0.49	172	99	0.23	**0.05**
	IM	0.92	0.79	0.85	0.48	116	63	0.31	0.16
	HM	**1.00**	0.30	0.46	**0.56**	121	58	**1.00**	0.25
$BPIC2015_{2_f}$	BM	0.92	**0.92**	0.92	0.52	**28**	12	-	**0.01**
	SM	0.97	0.88	0.92	0.46	33	17	0.15	0.04
	IM	**1.00**	0.64	0.78	**0.54**	30	**11**	**1.00**	0.1
	HM	**1.00**	0.88	**0.94**	0.49	33	16	0.15	0.1
$BPIC2015_{3_f}$	BM	0.84	**0.92**	**0.88**	0.53	**82**	**30**	0.28	**0.05**
	SM	0.72	0.68	0.70	0.49	117	52	0.20	0.15
	IM	**1.00**	0.49	0.66	**0.56**	102	46	**1.00**	0.18
	HM	0.90	0.86	**0.88**	0.52	91	39	0.25	0.16
$BPIC2015_{4_f}$	BM	0.89	**0.92**	**0.90**	**0.55**	**22**	**6**	**1.00**	**0.01**
	SM	0.89	**0.92**	**0.90**	**0.55**	**22**	**6**	**1.00**	0.02
	IM	**1.00**	0.71	0.83	0.53	26	8	**1.00**	0.1
	HM	0.88	**0.92**	**0.90**	**0.55**	**22**	**6**	**1.00**	0.1
$BPIC2015_{5_f}$	BM	0.96	**0.89**	**0.92**	0.38	**33**	**15**	0.67	**0.01**
	SM	**1.00**	0.85	**0.92**	**0.41**	36	16	0.64	0.04
	IM	**1.00**	0.47	0.64	**0.41**	36	16	**1.00**	0.12
	HM	**1.00**	0.85	**0.92**	0.38	36	18	0.61	0.12
$BPIC2017_f$	BM	0.93	0.96	**0.94**	0.80	**54**	50	0.69	15.68
	SM	0.83	**1.00**	0.91	0.80	142	116	0.23	**0.21**
	IM	0.94	0.85	0.89	0.92	62	**47**	0.45	25.24
	HM	**1.00**	0.22	0.36	**0.94**	91	69	**1.00**	30.92
$BPIC2017_{o_f}$	BM	**1.00**	**1.00**	**1.00**	**0.99**	15	8	0.27	5.2
	SM	**1.00**	**1.00**	**1.00**	**0.99**	15	8	0.27	**0.11**
	IM	**1.00**	0.80	0.89	0.99	**14**	**7**	**1.00**	11.02
	HM	0.91	**1.00**	0.95	0.79	18	10	0.22	11.79
$BPIC2020_{DD_f}$	BM	**1.00**	**0.99**	**1.00**	**0.97**	19	10	0.58	2.5
	SM	**1.00**	**0.99**	**1.00**	**0.97**	21	10	0.57	**0.05**
	IM	**1.00**	0.70	0.82	**0.97**	23	13	**1.00**	1.9
	HM	0.91	0.99	0.95	0.90	**18**	**8**	0.28	2.09
$BPIC2020_{ID_f}$	BM	0.89	0.93	0.91	0.84	**65**	**47**	0.19	2.96
	SM	0.85	0.75	0.79	0.78	149	125	0.05	**0.37**
	IM	**1.00**	0.25	0.40	**0.93**	112	76	**1.00**	3.26
	HM	0.91	0.94	**0.93**	0.72	100	89	0.10	3.44
$BPIC2020_{PL_f}$	BM	0.84	0.96	0.89	0.81	**85**	**60**	**0.51**	2.99

Table 2 continued from previous page

	SM	0.75	0.91	0.82	0.79	229	195	0.12	**0.55**
	IM	**1.00**	0.09	0.17	**0.90**	-	-	-	3.95
	HM	0.89	**0.97**	**0.93**	0.74	118	103	0.16	3.39
$BPIC2020_{PTC_f}$	BM	0.93	0.91	**0.92**	0.77	**62**	**44**	0.19	0.96
	SM	0.78	0.76	0.77	0.72	163	128	0.01	**0.28**
	IM	**0.98**	0.11	0.20	**0.89**	100	73	**1.00**	0.95
	HM	0.91	**0.94**	**0.92**	0.68	88	76	0.11	0.9
$BPIC2020_{RP_f}$	BM	**1.00**	**0.99**	**0.99**	0.96	21	11	0.48	1.96
	SM	**1.00**	**0.99**	**0.99**	**0.97**	24	11	0.46	0.05
	IM	**1.00**	0.78	0.88	**0.97**	24	15	**1.00**	1.5
	HM	0.92	**0.99**	0.95	0.90	**20**	**9**	0.10	1.62
$RTFMP_f$	BM	0.98	**1.00**	**0.98**	0.99	**13**	**6**	0.46	13.26
	SM	0.88	0.99	0.93	**1.00**	21	10	0.29	**0.29**
	IM	0.92	**1.00**	0.96	0.84	20	14	0.15	24.03
	HM	**1.00**	0.66	0.80	**1.00**	22	15	**1.00**	21.64
$SEPSIS_f$	BM	0.79	**0.99**	**0.88**	0.76	**26**	**16**	0.41	0.35
	SM	0.92	0.70	0.79	0.74	34	21	0.71	**0.02**
	IM	0.90	0.83	0.86	0.72	47	39	0.23	0.13
	HM	**1.00**	0.60	0.75	**0.86**	40	25	**1.00**	0.12

Table 2: Evaluation results

The generalization capabilities of BM stand out as a key strength. Across the majority of datasets, BM consistently achieved scores that matched or surpassed those of the baseline methods, demonstrating its ability to generalize beyond observed behaviors without succumbing to overgeneralization. For instance, in the RTFMP log, BM achieved an impressive generalization score of 0.99, while maintaining minimal structural complexity, highlighting its superior adaptability and precision.

Regarding model simplicity, BM frequently outperformed its competitors by generating smaller, more interpretable models with lower structural complexity. A notable example is the SEPSIS log, where BM produced a model with only 26 elements and a CFC of 16, significantly enhancing interpretability compared to SM, IM, and HM. Similarly, in synthetic loops such as L3 and L4, BM maintained high fitness and precision while preserving structural clarity, further showcasing its effectiveness in handling complex loop structures.

In terms of execution time, SM remains the fastest method for real-life event logs. However, BM demonstrates improved efficiency over IM and HM, striking a commendable balance between performance and computational demand.

Overall, the empirical results underscore BM's ability to deliver accurate, interpretable, and generalizable models. While other methods, such as SM and HM, exhibit specific strengths in certain scenarios, BM provides a comprehensive solution, excelling particularly in handling loops and generating well-rounded models across diverse datasets.

7 Conclusion & Future Work

This paper introduces Bonita Miner, a novel approach to process discovery that effectively balances key metrics—fitness, precision, generalization, and simplicity—even when dealing with complex loop structures. By leveraging the Depth-First Algorithm (DFA) for split and join gateway construction and advanced filtering of Directly-Follows Graphs (DFG), Bonita Miner simplifies the process model generation while maintaining behavioral accuracy. The proposed method addresses the limitations of existing techniques by producing simpler, more interpretable models that are robust to the intricacies of real-world processes, such as concurrency and loops. Empirical results demonstrate its superiority over state-of-the-art methods, particularly in its ability to generate behaviorally accurate and structurally simpler models.

Future enhancements to Bonita Miner could focus on several areas of development. One area is the optimization of execution time, as reducing the computational overhead for processing large-scale event logs would make the method more practical for industry-scale applications. Another promising direction is the incorporation of automatic deadlock detection and prevention mechanisms. This would not only enhance the robustness of the process models but also ensure their operational soundness in real-world scenarios.

References

1. van der Aalst, W.M.P.: Process Mining - Data Science in Action, Second Edition. Springer (2016)
2. van der Aalst, W.M.P., Weijters, T., Maruster, L.: Workflow mining: Discovering process models from event logs. IEEE Trans. Knowl. Data Eng. **16**(9), 1128–1142 (2004)
3. Augusto, A., Conforti, R., Dumas, M., Rosa, M.L., Bruno, G.: Automated discovery of structured process models: Discover structured vs. discover and structure. In: Conceptual Modeling - 35th International Conference, ER 2016, Gifu, Japan, November 14-17, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9974, pp. 313–329 (2016)
4. Augusto, A., Conforti, R., Dumas, M., Rosa, M.L., Maggi, F.M., Marrella, A., Mecella, M., Soo, A.: Automated discovery of process models from event logs: Review and benchmark. IEEE Trans. Knowl. Data Eng. **31**(4), 686–705 (2019)
5. Augusto, A., Conforti, R., Dumas, M., Rosa, M.L., Polyvyanyy, A.: Split miner: automated discovery of accurate and simple business process models from event logs. Knowl. Inf. Syst. **59**(2), 251–284 (2019)
6. Bender, M.A., Farach-Colton, M., Pemmasani, G., Skiena, S., Sumazin, P.: Lowest common ancestors in trees and directed acyclic graphs. J. Algorithms **57**(2), 75–94 (2005)
7. Berti, A., van Zelst, S., Schuster, D.: Pm4py: A process mining library for python. Software Impacts **17**, 100556 (2023)
8. Buijs, J.C.A.M., van Dongen, B.F., van der Aalst, W.M.P.: On the role of fitness, precision, generalization and simplicity in process discovery. In: On the Move to Meaningful Internet Systems: OTM 2012, Confederated International Conferences: CoopIS, DOA-SVI, and ODBASE 2012, Rome, Italy, September 10-14, 2012.

Proceedings, Part I. Lecture Notes in Computer Science, vol. 7565, pp. 305–322. Springer (2012)
9. Chinosi, M., Trombetta, A.: BPMN: an introduction to the standard. Comput. Stand. Interfaces **34**(1), 124–134 (2012)
10. Dumas, M., García-Bañuelos, L., Polyvyanyy, A.: Unraveling unstructured process models. In: Business Process Modeling Notation - Second International Workshop, BPMN 2010, Potsdam, Germany, October 13-14, 2010. Proceedings. Lecture Notes in Business Information Processing, vol. 67, pp. 1–7. Springer (2010)
11. Dumas, M., Rosa, M.L., Mendling, J., Mäesalu, R., Reijers, H.A., Semenenko, N.: Understanding business process models: The costs and benefits of structuredness. In: Advanced Information Systems Engineering - 24th International Conference, CAiSE 2012, Gdansk, Poland, June 25-29, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7328, pp. 31–46. Springer (2012)
12. Leemans, S.J.J., Fahland, D., van der Aalst, W.M.P.: Discovering block-structured process models from event logs - A constructive approach. In: Application and Theory of Petri Nets and Concurrency - 34th International Conference, PETRI NETS 2013, Milan, Italy, June 24-28, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7927, pp. 311–329. Springer (2013)
13. Leemans, S.J.J., Fahland, D., van der Aalst, W.M.P.: Discovering block-structured process models from event logs containing infrequent behaviour. In: Business Process Management Workshops - BPM 2013 International Workshops, Beijing, China, August 26, 2013, Revised Papers. Lecture Notes in Business Information Processing, vol. 171, pp. 66–78. Springer (2013)
14. Mendling, J., Reijers, H.A., van der Aalst, W.M.P.: Seven process modeling guidelines (7PMG). Inf. Softw. Technol. **52**(2), 127–136 (2010)
15. Weber, M., Kindler, E.: The petri net markup language. In: Petri Net Technology for Communication-Based Systems - Advances in Petri Nets. vol. 2472, pp. 124–144. Springer (2003)
16. Weerdt, J.D., Backer, M.D., Vanthienen, J., Baesens, B.: A multi-dimensional quality assessment of state-of-the-art process discovery algorithms using real-life event logs. Inf. Syst. **37**(7), 654–676 (2012)
17. Weijters, A.J.M.M., Ribeiro, J.T.S.: Flexible heuristics miner (FHM). In: Proceedings of the IEEE Symposium on Computational Intelligence and Data Mining, CIDM 2011, part of the IEEE Symposium Series on Computational Intelligence 2011, April 11-15, 2011, Paris, France. pp. 310–317. IEEE (2011)
18. Weijters, A.J.M.M., Ribeiro, J.T.S.: Flexible heuristics miner (FHM). In: Proceedings of the IEEE Symposium on Computational Intelligence and Data Mining, CIDM 2011, part of the IEEE Symposium Series on Computational Intelligence 2011, April 11-15, 2011, Paris, France. pp. 310–317. IEEE (2011)