Journal of Intelligent Information Systems
https://doi.org/10.1007/s10844-025-00997-7

RESEARCH

I-PALIA, An algorithm for discovering BPMN processes with duplicated tasks

Carlos Fernandez-Llatas[1,3] · **Andrea Burattin**[2]

Received: 12 March 2024 / Revised: 2 October 2025 / Accepted: 3 October 2025

Abstract
Process mining encompasses a range of methods designed to analyze event logs. Among these methods, control-flow discovery algorithms are particularly significant, as they enable the identification of real-world process models, known as *in-vivo* processes, in contrast to anticipated models. An obstacle faced by control-flow discovery algorithms is their limited ability to recognize duplicated activities, which are activities that occur in multiple locations within a process. This issue is particularly relevant in the healthcare sector, where numerous instances of duplicated activities exist in processes but remain undetected by conventional algorithms. This article introduces a novel concept for a control-flow discovery algorithm capable of effectively revealing duplicated activities. The effectiveness of this technique is demonstrated through experimentation on a synthetic dataset. Moreover, the algorithm has been implemented and its source code is available as open-source software, accessible both as a ProM plugin and a Java Maven dependency.

Keywords Process mining · Control-flow discovery · BPMN · Duplicated activities

Carlos Fernandez-Llatas and Andrea Burattin contributed equally to this work.

✉ Carlos Fernandez-Llatas
cfllatas@itaca.upv.es

Andrea Burattin
andbur@dtu.dk

[1] SABIEN-ITACA, Universitat Politècnica de València, Camino de Vera S/N, Valencia 46022, Valencia, Spain

[2] DTU Compute, Technical University of Denmark, Lyngby, Denmark

[3] Department of Clinical Science, Intervention and Technology,, Karolinska Institutet, Stockholm 17177, Sweden

Published online: 20 October 2025

1 Introduction

Process mining is the scientific discipline aiming at connecting process models and recordings of activity executions (Aalst, 2016). In particular, *control-flow discovery* techniques pertain to the synthesis of models that are capable of explaining in a compact way all (or most of) the executions reported in an event log.

The ultimate goal of process mining techniques is to improve processes and how these are actually deployed in the real world. Since these models are supposed to improve actual processes, it is essential that the models identified are as reliable and *good* as possible.

When considering the control-flow discovery techniques, the main challenge they need to face consists of extracting a model that is the most suitable representation possible. Defining *most suitable representation* is a challenge in itself and, in the literature, numerical approaches to quantify this dimension have been proposed, in particular *fitness*, *precision*, *generalisation* and *simplicity* (Aalst, 2016). Fitness indicates that a model should be able to replicate the log it has been generated from; precision quantifies how much more behavior (w.r.t. the starting log) the mined model permits; generalization tries to capture to what extent behavior not observed in the log is present in the model; and, finally, simplicity verifies that the model should be as simple as possible, to foster understandability. All these metrics should be maximized in order to obtain good results.

While many algorithms for control-flow discovery have put a lot of focus on optimizing fitness and precision (Leemans et al., 2014a; Aalst et al., 2006; Augusto et al., 2019), less emphasis has been put on the simplicity dimension, in particular regarding the type of supported behavior. Specifically, as mentioned in the Process Mining Manifesto (Daniel et al., 2011) (as guiding principle GP3), control-flow discovery algorithms should be able to identify basic control-flow constructs (Russell et al., 2006; Aalst et al., 2003), such as concurrency and choice. In many situations, a limiting factor towards better simplicity is the problem of *duplicated activities*. This problem stems from the fact that most control-flow discovery algorithms are not able to produce process models where the same activity occurs more than once.

In many scenarios, it is easy to find processes with duplicated tasks. For instance, within the healthcare sector, it's common to encounter activities that occur repeatedly. Instances such as revisiting medical appointments, undergoing assessment procedures like laboratory tests and medical imaging, or undergoing cyclic treatments like dialysis or chemotherapy are recurrent events throughout a patient's journey within the care process. These activities carry significant importance in representing the overall process. To illustrate, the sequencing of treatments can vary significantly based on the timing of assessments. For instance, in the context of cancer treatments, administering chemotherapy before and/or after surgery can yield distinct results. The initial treatment aims to shrink the tumor size and streamline the surgical procedure, while the subsequent treatment focuses on preventing the proliferation of harmful cells, introducing differing objectives and complexities that impact process delineation. This problem can be found in many other process scenarios like manufacturing, logistics, and auditing.

When applying conventional process discovery algorithms to these scenarios, the ability to differentiate between these distinct behaviors becomes compromised. In this example, as well as in numerous other cases within the healthcare sphere, the utilization of duplicated nodes becomes imperative. Not only do they contribute to a lucid depiction of the process,

but they also facilitate comprehension of the preparatory and follow-up stages surrounding these recurrent activities.

As a simple example of a process that shows duplicated activities, consider the process model reported in Fig. 1. In this normative process, you see that the activity of contacting a potential customer takes place both at the beginning and at the end of the process. While the two activities may involve slightly different tasks, for the sake of modelling it is legitimate that they have the same label. As a consequence, event logs recording executions of this process will show the presence of two activities called *Contact potential customer* which, however, should not be collapsed into the same activity in the model: though these have the same label, their mining is distinguishable based on the position where they occur; hence, these should appear two times.

In this paper, we present a new discovery algorithm capable of extracting process models, represented as BPMN (OMG, 2009), which is based on the control of the generalization in the process of log inference using Grammar Inference techniques. The algorithm can identify all the basic workflow patterns (i.e., sequence, parallel split, synchronization, exclusive choice, and simple merge) with the addition of duplicated activities. This algorithm is inspired by the PALIA algorithm, which uses grammar inference approaches for discovering processes taking the complete schedule of the log events for identifying parallelism (Fernández-Llatas et al., 2010; Rojas et al., 2017). This paper extends (Fernandez-Llatas and Burattin, 2023) by:

- Improving the state of the art with respect to duplicated task mining;
- Extending the algorithm with different abstraction capabilities;
- Extending the description of the algorithm and the data structure;
- Presenting a new implementation of the approach in a real system.

The rest of the paper is structured as follows: Section 2 introduces the background and the state of the art of the proposed technique; Section 3 states some preliminary concepts for the definition of the algorithm; Section 4 presents the actual algorithm proposed in the paper; Section 5 describes the implementation of the technique; Section 6 presents some performance results of the algorithm against some extreme scenarios; Section 7 presents some discussion points and best practices in the application of the algorithm; and finally Section 8 concludes the paper.

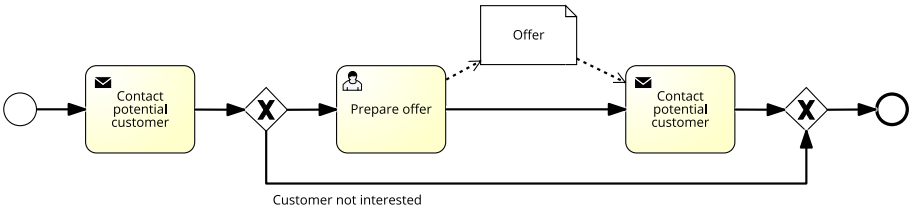


Fig. 1 Example of process with duplicated activities

2 Background and state of the art

There were many attempts in the literature to create solutions that can deal with the problem of duplicated tasks (Duan & Wei, 2020). In the literature on control-flow discovery (Augusto et al., 2019), many algorithms have been developed capable of identifying all the basic workflow patterns. Among these, the Alpha miner is typically recalled as one of the first algorithms explicitly tackling the control-flow discovery problem. More advanced algorithms, such as the Heuristics Miner, the Fuzzy Miner, the Split Miner, and the Inductive Miner, have gained much popularity due to the quality of the output they produce and their performance. However, none of these is actually capable of producing duplicated activities. The algorithms that are able to achieve this are very few, including the $\alpha*$-algorithm (Calders et al., 2009), which nonetheless has very restrictive assumptions on the event log. Fodina (Calders et al., 2009) can discover duplicated activities by pre-processing the event log only based on some heuristics. Genetic Process Mining (Medeiros, 2006), Evolutionary Tree Miner (ETM) (Buijs et al., 2014), AGNES (Goedertier et al., 2009) are evolutionary algorithms that, in principle, can discover duplicated activities at the expense of extreme computational complexity. Some techniques apply heuristic approaches for identifying duplicated activities and provide relabelling (Lu et al., 2016) that could feed existing Process Mining Discovery algorithms. SLAD (Vázquez-Barreiros et al., 2016) is another approach that post-processes the mined model for duplicated activities. Also in this case, however, the algorithm exploits some heuristics to simplify the model by duplicating some behaviour.

There are works that generate stochastic process models based on grammatical inference, such as those relying on the inference of stochastic deterministic finite automata (SDFA) (Alkhammash et al., 2024). These approaches are capable of capturing the sequential structure and the probability of occurrence of observed traces, resulting in models that reflect the statistical distribution of the data. However, an important limitation of these techniques is that, due to the inherently sequential nature of automata, they are unable to identify or represent complex behavioral patterns such as parallelism. As a result, the discovered models typically only describe strict precedence relationships between activities, without adequately capturing the existence of activities that can be performed in parallel or independently within the process.

As a basis for the paper, we should provide some definitions. Classic PALIA algorithm (Fernández-Llatas et al., 2010; Rojas et al., 2017) uses as a representation model, a Timed Parallel Automaton (Fernandez-Llatas et al., 2011). This model has an expressiveness equivalent to a safe Petri Net (Peterson, 1977) and has a Regular complexity (Fernandez-Llatas et al., 2011) based on the concept of Parallel Finite Automaton (Stotts & Pugh, 1994). For this paper, a TPA is defined as follows:

Definition 1 (Timed Parallel Automaton (TPA)) A Timed Parallel Automaton (TPA) (Fernandez-Llatas et al., 2011) is a tuple $\mathcal{T} = \{N, Q, \gamma, \delta, S, F\}$ where:

- N is a finite set of Nodes, where a Node n is a graphical representation of the action a,
- Q is a finite set of states where $q \subseteq N^+ \forall q \in Q$,
- $\gamma : N^+ \rightarrow N^+$ is the Node Transition function, where $\gamma = (\{n_0^s, .., n_i^s\}, \{n_0^e, .., n_j^e\})$. Γ is the set of γ transitions;

- $\delta : Q \rightarrow Q$ is the State transition function where $\delta = (q^s, q^e)$. Δ is the set of δ transitions where $(\forall \delta \in \Delta \; \exists \; \gamma \in \Gamma \mid \{n_0^s, \ldots, n_i^s\} = q^s \wedge \{n_0^e, \ldots, n_i^e\} = q^e)$ where γ is the associated transition of δ;
- $S \subseteq N$ is the set of Starting Nodes;
- $F \subseteq N$ is the set of Final Nodes.

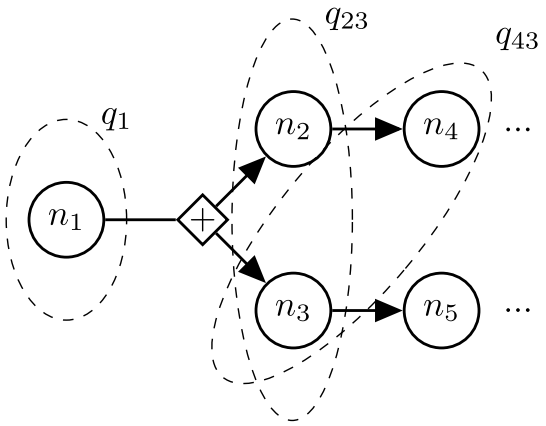
A TPA has a double transition function, defining Nodes (N) and States (Q). N^+ is the set of all combinations of $n \in N$ with at least one n. The transitions between nodes can be n to n multiple for defining parallelism. For example, a transition $\gamma_1 = (\{n_1\}, \{n_2, n_3\})$ represents a parallel split from the node n_1 to the nodes n_2 and n_3. In the same way, a transition $\gamma_2 = (\{n_1, n_2\}, \{n_3\})$ represent a parallel synchronization from the nodes n_1 and n_2 to the node n_3. Exclusive Choice can be represented using several γ transitions. For example, having $\gamma_1 = (\{n_1\}, \{n_2\})$ and $\gamma_2 = (\{n_1\}, \{n_3\})$ we can represent that, from n_1 is possible to go to n_2 or n_3.

The States function (Δ) captures the regular language behavior underlying the interpretation of the automaton. This means that, on its own, it can represent the sequential behavior that occurs between the parallelisms described by the (Γ) function. In the (Δ set), the parallelism is represented at the state level. A State q is a set of Nodes that represents the actions that are active in the state q in an analogous way to a Petri-Net marking. Δ and Γ are complementary. The state transitions (δ) keep the regular complexity due to their simplicity (N to N) where the Node Transitions (Γ) represent parallel situations (N^+ to N^+). In practice, there is only one state active in a moment in time, which supposes that it could be several nodes active. Node transitions represent the single pass from one node to another.

A visual example of how the double transition function works for representing parallel patterns can be shown in Fig. 2. The TPA can be formally described as can be seen at the bottom of the Figure[1] In this example, N represents the set of nodes available in the automaton, while Q represents the set of possible nodes that could be active at the same time, in a similar way to Petri Nets marking. In this line, the function Γ defines the possible transitions between the nodes $\gamma_1 = (\{n_1\}, \{n_2, n_3\})$ states the parallel split sequences headed by nodes n_2 and n_3. Analogously δ_1 represents the transition in a single transition from the state q_1 to state q_{23} and γ_1 is the associated transition of δ_1. By continuing the sequence using transition γ_2, the node n_2 finishes, and the node n_4 starts. Similarly, the transition δ_2 has the same execution information but keeping in mind that the node n_3 is still active. Using this double transition function, it is possible to represent the many-to-many transition information in the Γ function, but having global control of the execution of parallel nodes in Δ only with one-to-one transitions, keeping in Δ function a regular languages level complexity.

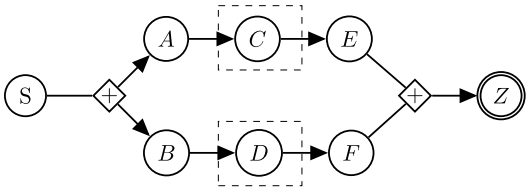
This double transition function allows representing complex workflow patterns (Fernandez-Llatas et al., 2011). For instance, Fig. 3 represents an example of an interleaved parallel routine. In this pattern, some parallel sequences are executed, but some actions are not allowed to be executed simultaneously. For example, the dashed rectangles in the figure represent sequences of protected actions. In this line, nodes C, and D, can't be executed at the same time. This pattern could be represented formally as a TPA as defined at the bottom of the Figure. As can be seen in the formal representation, the Γ Function reflects all the

[1] The notation has been simplified and only the states required for the understandability of the example have been represented



$$
\begin{aligned}
N =& \{n_1, n_2, n_3, n_4, n_5, ...\} \\
Q =& \{q_1, q_{23}, q_{43} ...\} \\
\Gamma =& \{(\{n_1\}, \{n_2, n_3\}), (\{n_2\}, \{n_4\}), (\{n_3\}, \{n_5\}) ...\} \\
\Delta =& \{(q_1, q_{23}), (q_{23}, q_{43}), ...\}
\end{aligned}
$$

Fig. 2 Example of visual representation of Γ and Δ functions



$$
\begin{aligned}
N =& \{S, A, B, C, D, E, F, Z\} \\
Q =& \{q_S, q_{AB}, q_{CB}, q_{EB}, q_{AD}, q_{AF}, q_{EF}, q_Z\} \\
\Gamma =& \{(\{S\}, \{A, B\}), (\{A\}, \{C\}), (\{C\}, \{E\}), (\{B\}, \{D\}), (\{D\}, \{F\}), (\{E, F\}, \{Z\})\} \\
\Delta =& \{(q_S, q_{AB}), (q_{AB}, q_{CB}), (q_{CB}, q_{EB}), (q_{CB}, q_{EB}), (q_{EB}, q_{ED}), \\
& (q_{ED}, q_{EF}), (q_{AB}, q_{AD}), (q_{AD}, q_{AF}), (q_{AF}, q_{CF}), (q_{CF}, q_{EF}), (q_{EF}, q_Z)\}
\end{aligned}
$$

Fig. 3 Interleaved parallel routine represented with TPA

transitions described on the graphical TPA. However, Δ function skips the prohibited transitions like (q_{AD}, q_{CD}) or (q_{CB}, q_{CD}).

In a TPA, Δ orchestrates the actions from a high level, and Γ defines all the possible many-to-many transitions at a low level. Both are complementary, on one hand, without Δ, Γ is unable to know globally the allowed transitions, and on the other hand, without Γ, Δ can't be safe in identifying the next node to activate. This is clear in self-loops. For example,

in a Self Loop (q_{AB}, q_{AB}), without the Γ associated transition, Δ is unable to know if there is a self-loop in node A or in node B.

3 Preliminary concepts

The main objective of this paper is to build a discovery algorithm that can infer TPAs from log events. This requires the ability infer the double transition function. However, it is possible to build an algorithm that, given a Γ set, it constructs a basic Δ set of states with some limitations. This will allow the discovery algorithm to be focused on the inference of Γ transitions, assuming that Δ can be extracted afterwards.

In this section, we will present the algorithm that we have devised to infer the Δ function from Γ and some preliminary definitions such as, domains, ranges or event logs for the formalization of the algorithm.

Definition 2 The Domain of a function $\gamma = (\{n_0^s, .., n_i^s\}, \{n_0^e, .., n_j^e\}) \in \Gamma$ is the set starting nodes of the function.

$$Domain(\gamma) = \{n_0^s, .., n_i^s\}$$

Definition 3 The Range of a function $\gamma = (\{n_0^s, .., n_i^s\}, \{n_0^e, .., n_j^e\}) \in \Gamma$ is the set of ending nodes of the function.

$$Range(\gamma) = \{n_0^e, .., n_j^e\}$$

Require: N,Γ
Ensure: Q,Δ
1: $Q_{nv} \leftarrow InitialStates(N)$
2: **while** $Q_{nv} \neq \emptyset$ **do**
3: $q \leftarrow Q_{nv}[0]$
4: $Q_{nv} \leftarrow Q_{nv}[1:]$
5: **for all** $\gamma \in \Gamma | Domain(\gamma) \subseteq q$ **do**
6: $q' \leftarrow q - Domain(\gamma) + Range(\gamma)$
7: **if** $q' \in Q$ **then**
8: $Q_{nv} \leftarrow Q_{nv} \cup \{q'\}$
9: **end if**
10: $\delta \leftarrow (q, q', \gamma)$
11: $\Delta \leftarrow \Delta \cup \{\delta\}$
12: **end for**
13: **end while**

Algorithm 1 Δ State transitions inference.

The "Δ State Transitions Inference Algorithm" is presented in Algorithm 1. This algorithm traverses the nodes from the initial nodes through the γ transitions that are the origin (Domain). In each iteration, the algorithm creates the possible states (q') possible at any moment by analyzing all the possible paths in the TPA. When a not visited state is reached, this is annotated in the set of not visited states (Q_{nv}). The algorithm stops when all the states

are visited and analysed. As a result, the Algorithm returns in Q all the possible states and in Δ all the possible one-to-one state transitions.

In case of no parallelism, Γ and Δ are equivalent; all the states have a single node. The difference resides when Γ has parallel nodes. In that case, Δ can be seen as the explosion of Γ transitions derivations defining the Q as all the possible active nodes in parallel in a moment in time and Δ equivalent to Γ representing the transitions between the node sets.

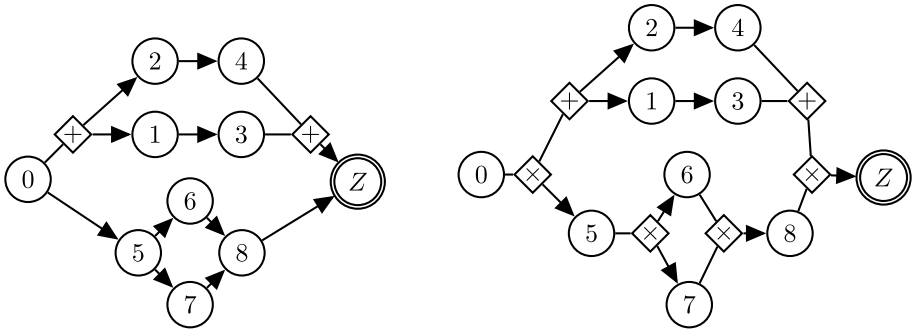
This algorithm is simple but also limited. This algorithm is not able to deal with more complex representations like Interleaved Parallel Routines or milestones. This is because it only captures strict sequential relationships between activities and lacks mechanisms to explicitly represent concurrency or conditional dependencies. As a result, patterns involving parallelism, interleaving, or milestone constraints cannot be properly distinguished or modeled, and are instead flattened into purely sequential or alternative paths. On the one hand, as it is not expected that the discovery algorithm could discover this kind of structure, we consider this algorithm enough to represent the final model. On the other hand, the simplicity of this algorithm allows a simple graphical TPA representation that defines parallel gateways in many-to-many Γ transitions and assumes that all single arrows are exclusive Or (XOR) gateways. Due to that, it is easy to translate from a TPA under these conditions to a BPMN model. Figure 4 presents an example of how a TPA can be translated to a BPM notation format. In the example, the parallel sequences have the same notation, and the single arrows are decorated with XOR gateways.

In the following, we present the definitions of Event and Event Log:

Definition 4 An Event is a tuple $e = (a, \pi)$ where $a \in \mathcal{A}$ is the set of possible actions in a process, π is the timestamp execution of the action.

Definition 5 An Event Log ($\mathcal{L}$) is a set of traces $\mathcal{L}$=$\{t_0, \ldots, t_i\}$, where a trace is a sequence of events t=$[e_0, \ldots, e_j]$

An *Event* is the digital representation of an action at a moment in time. A set of events referenced to the same user or the same execution instance is called *Trace*.



(a) TPA (b) BPMN

Fig. 4 Correspondence between TPA and BPMN

4 Inductive PALIA (I-PALIA): the algorithm

Within this paper, we introduce an upgraded iteration of the PALIA algorithm, specifically devised to uncover activity logs through the application of Grammar Inference methods. Subsequently, the algorithm undertakes the task of detecting parallel configurations within the inferred model.

To facilitate this process, we have established certain foundational definitions:

Definition 6 (Directly followed) Let $\mathcal{T} = (N, \gamma)$ be a TPA, where N is the set of nodes and $\gamma \subseteq N \times N$ is the node transition relation. For $n_0, n_1 \in N$, we say that n_0 is *directly followed* by n_1 (denoted $n_0 \to n_1$) if and only if

$$(n_0, n_1) \in \gamma.$$

Similarly, for node traces γ_0, γ_1, we say that γ_0 is directly followed by γ_1 (denoted $\gamma_0 \to \gamma_1$) if and only if there exist $(n_0, n_1) \in \gamma_0$ and $(n_0', n_1') \in \gamma_1$ such that $n_1 \to n_0'$.

Definition 7 (Eventually followed) Let $\mathcal{T} = (N, \delta)$ be a TPA, where N is the set of nodes and $\delta \subseteq N \times N$ is the node transition relation. For $n_0, n_1 \in N$, we say that n_0 is *eventually followed* by n_1 (denoted $n_0 \Rightarrow n_1$) if and only if there exists a finite sequence of nodes $n_0, n_1', n_2', \ldots, n_{k-1}', n_1$ such that

$$(n_0, n_1') \in \delta,\ (n_1', n_2') \in \delta,\ \ldots,\ (n_{k-1}', n_1) \in \delta,$$

that is, there exists a path from n_0 to n_1 in the transition relation δ.

Similarly, for node traces γ_0, γ_1, we say that γ_0 is eventually followed by γ_1 (denoted $\gamma_0 \Rightarrow \gamma_1$) if and only if there exist $(n_0, n_1) \in \gamma_0$ and $(n_0', n_1') \in \gamma_1$ such that $n_1 \Rightarrow n_0'$. These definitions describe that two node transitions are directly followed ($\to$) when the second transition can be immediately accessed from the first one and eventually followed ($\Rightarrow$) when the second transition can be eventually accessed from the first one.

Definition 8 (Compatibility) Let $e = (a, \pi)$ be an event, where $a \in A$ is the activity label and π is the timestamp. Let $\mathrm{lab} : N \to A$ denote the function that maps each node to its activity label.

- Event–event: $e_0 \equiv e_1$ iff $a(e_0) = a(e_1)$.
- Event–node: $e \equiv n$ iff $a(e) = \mathrm{lab}(n)$.
- Node–node: $n_0 \equiv n_1$ iff $\mathrm{lab}(n_0) = \mathrm{lab}(n_1)$.
- Set–set: For finite sets $N_0, N_1 \subseteq N$, we write $N_0 \equiv N_1$ iff $|N_0| = |N_1|$ and there exists a bijection $f : N_0 \to N_1$ such that

$$\forall n \in N_0, \quad \mathrm{lab}(n) = \mathrm{lab}\big(f(n)\big).$$

Colloquially, Nodes and Events are compatible when they refer to the same action, and two sets of nodes are compatible if each one of the nodes of each set is compatible to a node of the other set.

Definition 9 (Transitions compatibility) Given two node transitions $\gamma_0 = \{D_0, R_0\}$, $\gamma_1 = \{D_1, R_1\}$, they are compatible ($\gamma_0 \equiv \gamma_1$) if $(Domain(\gamma_0) \equiv Domain(\gamma_1) \wedge Range(\gamma_0) \equiv Range(\gamma_1))$

Analogously, two node transitions are compatible ($\gamma_1 \equiv \gamma_2$) if their domains and ranges are compatible. We also define different modes of compatibility

Definition 10 (Transitions Strict compatibility) Given two node transitions $\gamma_0 = \{D_0, R_0\}$, $\gamma_1 = \{D_1, R_1\}$, they are strictly compatible ($\equiv^!$) if $Domain(\gamma_0) \equiv Domain(\gamma_1) \wedge Range(\gamma_0) = Range(\gamma_1)$

In other words, two node transitions are strictly compatible ($\gamma_1 \equiv^! \gamma_2$) if their domains are compatible, and their range nodes are exactly the same.

Definition 11 (Transitions Inline compatibility) Given two node transitions $\gamma_0 = \{D_0, R_0\}$, $\gamma_1 = \{D_1, R_1\}$, they are inline compatible ($\equiv^{\Rightarrow}$) if $\gamma_0 \equiv \gamma_1 \wedge (\gamma_0 \Rightarrow \gamma_1 \vee \gamma_1 \Rightarrow \gamma_0)$

That means, two node transitions are inlinely compatible ($\gamma_1 \equiv^{\Rightarrow} \gamma_2$) when these are compatible, and in addition, one is eventually followed by the other. In other words, both are located in the same process branch.

Definition 12 (Prefix Acceptor Tree (PAT)) A Prefix Acceptor Tree (PAT) is a tree-like TPA built from the learning log by taking all the prefixes in the sample as states and constructing the smallest TPA which is a tree, strongly consistent with the Log.

The Prefix Acceptor Tree creates a TPA that represents a tree from left to right with the events of the samples. The head of the tree is formed by the nodes representing the starting events of the traces, and the leaves represent the last events of the traces. Figure 5 shows an example of how this tree is created. It should be noticed that although TPA is able to represent parallel situations, the Prefix Acceptor Tree creates only non-parallel δ functions, so this algorithm is not able to represent parallelism.

Require: $\mathcal{L}, EquivalentMode(\equiv^?)$
Ensure: $\mathcal{T}$
1: $\mathcal{T} \leftarrow PrefixAceptorTree(\mathcal{L})$ ▷ Prefix Acceptor Tree Stage
2: **for all** $\gamma : (n^s, n^e) \in \Gamma$ **do** ▷ Consecutive Merge Stage
3: **if** $n^s \equiv n^e$ **then** Merge(n^s, n^e)
4: **end if**
5: **end for**
6: **for all** $n_0, n_1 \in F | n_0 \equiv n_1$ **do** ▷ Onward Merge Stage
7: $Merge(n_0, n_1)$
8: **end for**
9: **while** $\mathcal{T}$ has changes **do**
10: **for all** $\gamma_x \equiv^? \gamma_y | \gamma_x, \gamma_y \in \Gamma$ **do**
11: $Merge(Domain(\gamma_x), Domain(\gamma_y))$
12: $Merge(Range(\gamma_x), Range(\gamma_y))$
13: **end for**
14: **end while**
15: $\mathcal{T} \leftarrow ParallelMerge(\mathcal{T})$ ▷ Parallel Merge Stage. See Algorithm 3

Algorithm 2 Inductive PALIA algorithm.

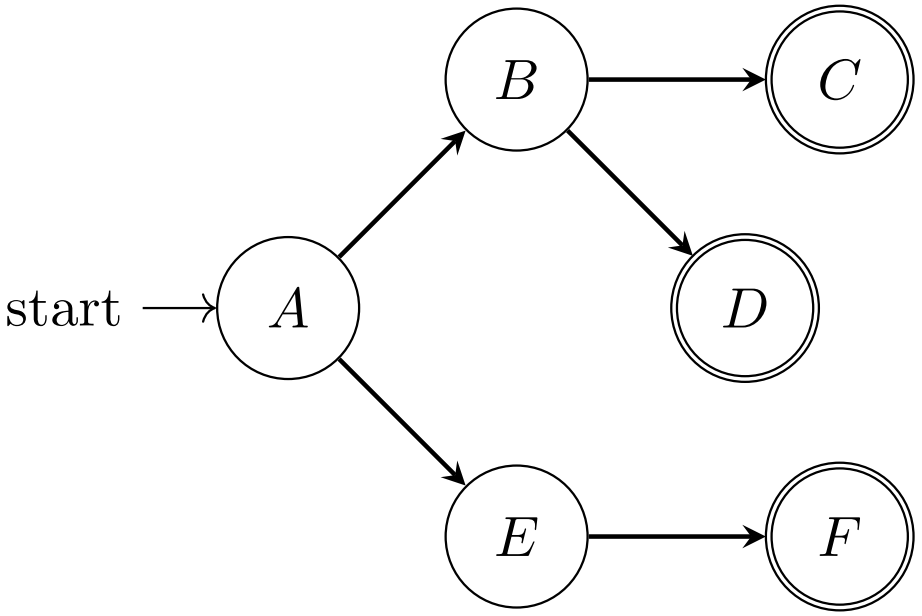


Fig. 5 Prefix Acceptor Tree of Log ={ABC,ABD,AEF}

Algorithm 2 shows the pseudocode of the presented algorithm. We can split the algorithm into four main stages. The first stage is to compute the *Prefix Acceptor Tree* that represents the log. Using the Prefix Acceptor Tree as a basis, Inductive PALIA will perform some generalizations over the algorithm using Grammar Inference techniques into two more stages. In the second stage, *Consecutive Merge*, the algorithm merges compatible consecutive nodes, generalizing all the transitions from one node to itself. In the third stage, *Onward Merge*, the algorithm merges all the final nodes that are compatible and, the algorithm Merge the domains and ranges of each couple of γ in Γ that are compatible according to one of the compatible modes defined, Strict ($\equiv^{!}$), Inline ($\equiv^{\Rightarrow}$), or Compatible ($\equiv$). The mode of compatibility selected affects the generalization of the algorithm. The *Strict Compatibility* is the more restrictive because it only generalizes the transitions sharing the ranges, producing more duplicated tasks. The *Inline Compatibility* generalizes the transitions located in the same domain of action (same branch), which means that they can be referenced to the same trace. Finally, the simple *Compatibility* maximizes the generalization, merging all the compatible transitions in the model independently of their position. This is the one that generates fewer duplicated tasks. The Onward Merge stage is performed until the TPA $\mathcal{T}$ has no new merges. This algorithm allows an ordered merging of the nodes that prevents their massive merging like in other basic algorithms such as Directly Follows Graphs (DFG) depending on the selected merging schema. The effects of the application of these generalizations schema will be analyzed in more detail in the following section with some examples.

Until this moment, the algorithm has discovered a process structure with sequences, splits, and loops, assuming no parallelism, but differentiating repeated non-consecutive

nodes. The next step is to identify the parallel sequences with the Algorithm 3 in the *Parallel Merge Stage*. In this algorithm, the parallelism has been defined as:

Definition 13 (Parallelism) Given $n_0, n_1 \in N$, n_0 and n_1 are parallel $(n_0 \| n_1)$ where:

$$n_0 \Rightarrow n_1 \wedge n_1 \Rightarrow n_0 \tag{1}$$

$$!n_0 \Leftrightarrow n_1 \wedge !n_1 \Leftrightarrow n_0 \tag{2}$$

Intuitively, two nodes are parallel $(n_0 \| n_1)$ if both are eventually followed by each other and these nodes are not acting as a loop, that means, there is no same path of transitions that contains $n_0 \Rightarrow n_1$ and $n_1 \Rightarrow n_0$, and vice-versa $(n_0 \Leftrightarrow n_1)$. With that definition, the Parallel Merge (Algorithm 3) defines the Parallel Regions for discovering the parallel situations:

Definition 14 (Parallel region) Given a TPA $\mathcal{T}$ and node n_{Split}, A *Parallel Region* is a set of nodes *R* where $n_{Split} \Rightarrow n_x \wedge \exists n_y : n_x \| n_y | \forall n_x, n_y \in R$.

A Parallel Region is a set of nodes that occurs after a split node n_{Split} and have parallelisms between themselves.

Definition 15 (Synchronization node) Given a TPA $\mathcal{T}$, a node n_{Split}, and a Parallel Region *R*, a Syncronization Node is a node n_{Syncro} where $n_i \Rightarrow n_{Syncro} | \forall n_i \in R$ and $\forall n_x | n_{Split} \Rightarrow n_x \wedge n_x \Rightarrow n_{Syncro} | n_x \in R$.

A Synchronization Node n_{Syncro} is a node that occurs immediately after the Parallel region. n_{Split} and n_{Syncro} delimit the Parallel Region and are the nodes that will define the start and the end of parallelism. According to that, the Algorithm 3 Discover and create the Parallel structures.

For example, in Fig. 8c, we can see that there is a node *A* located just before an isolated region of nodes such that everyone passing through *A* eventually passes through *D*. In this way, *A* would be the Split Node, *D* the Synchronization Node, and the intermediate nodes would constitute the parallel region.

Require: $\mathcal{T}$
Ensure: $\mathcal{T}$
1: **for all** $n_{Split} \in N$ **do**
2: $R \leftarrow GetParallelRegions(\mathcal{T}, n_{Split})$
3: **for all** $p \in R$ **do**
4: $n_{Syncro} \leftarrow GetSyncroNode(\mathcal{T}, n_{Split}, R)$
5: $seq \leftarrow$IdentifyParallelSequences($\mathcal{T}, R$) ▷ See Algorithm 4
6: $\mathcal{T} \leftarrow$CreateParallelTransitions($\mathcal{T}, n_{Split}, n_{Syncro}, R, seq$)
7: **end for**
8: **end for**

Algorithm 3 Parallel merge.

The *Parallel Merge* Algorithm discovers parallel regions after each node on the TPA. The algorithm $GetParallelRegions(\mathcal{T}, n_{Split})$ searches for all possible regions in the model by traversing all the nodes and checking if there exists a synchronization node that defines a parallel region. Once a Parallel Region is detected, the subsequent task involves

pinpointing the synchronization node that marks the boundaries of said Parallel Region. Subsequently, the algorithm proceeds to determine the parallel sequences within the identified Parallel Region. This specific process is detailed in Algorithm 4.

Require: $\mathcal{T}$, R
Ensure: $SeqMap$
1: **for all** $r \in R$ **do**
2: $\quad SeqMap(r) \leftarrow (r, S)$ where $S \subset R \wedge s_i \nparallel r | \forall s_i \in S$
3: **end for**

Algorithm 4 Identify parallel sequences.

The algorithm 4: *Identify Parallel Sequences* segregates the nodes within the Parallel Region into distinct groups, classifying those that do not exhibit parallelism with the rest. This methodology serves to discern the parallel sequences contained within the defined Parallel Region.

The algorithm identifies parallel nodes ($n_1 \| n_2$) that have a common previous transition ($n \rightarrow n_1$ and $n \rightarrow n_2$) to a node that is called n_Split. Once the sequence groups are identified, the algorithm identifies the n_{Sync} node, which is the first common node after the parallel sequence that marks the end of the sequence. With the n_{Split} and the n_{Sync}, we can identify the region of parallelism between them. Each node is assigned to one of the sequences, that is the one that is not parallel with the starting parallel node and is parallel with the rest. After this assignation, the parallel sequences are isolated removing all the leftover transitions.

For example, in Fig. 8c, within the identified parallel region, there are two possible starting nodes, B and C, and one secondary node, $B1$. Analyzing the sequences, we observe that $B1$ satisfies the parallelism condition with C (it can occur both before and after C), but not with B (it always occurs after B), so node $B1$ is assigned to the sequence of B. By separating the branches, we see that there are two branches: one with C, and another with B followed by $B1$.

Upon successful identification of the parallel sequences, the Parallel Merge Algorithm (Algorithm 3) allows the creation of the ascertained parallel structure. This involves establishing connections between each parallel sequence and the Split node at the beginning, as well as the Synchronization node at the conclusion of the respective sequences.

We can see an example of the Parallel Merge Algorithm with the Fig. 8.

5 Implementation

I-PALIA has been implemented in Java and as a Maven project as well as a ProM plugin.[2] Having the code available as a stand-alone Java project simplifies its embedding into new projects (the code can easily be imported into any Maven/Gradle/Ivy project[3]). The PaliaProM package, which imports the Maven dependency, allows us to easily benefit from the algorithm leveraging the infrastructure made available by ProM. The package contains two plugins, one called “Palia Miner” which takes an XES log object as input and produces a standard BPMN object as a result. The second plugin made available in the PaliaProM

[2] The Maven project is available at https://github.com/dtu-pa/palia. The ProM package, called PaliaProM, is available at https://github.com/dtu-pa/PaliaProM.

[3] See https://jitpack.io/#dtu-pa/palia.

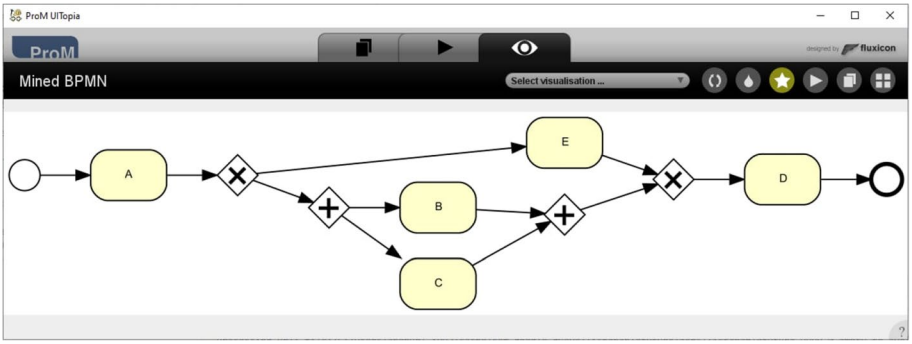


Fig. 6 The PaliaProM plugin showing the result of a mining session

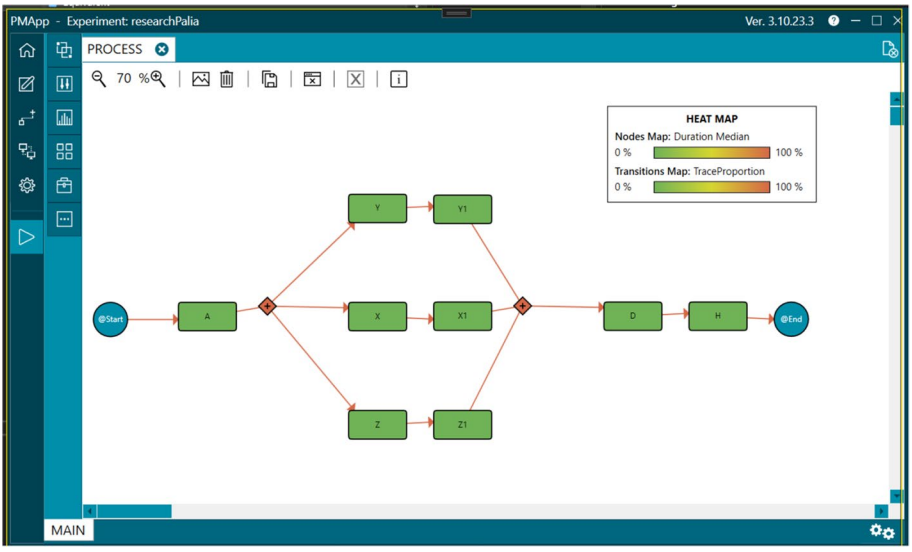


Fig. 7 PMApp application showing I-PALIA functionality

package is a visualizer for BPMN called "Graphviz BPMN visualisation" which exploits Graphviz to display any BPMN process model. A graphical representation of the latter is reported in Fig. 6.

Also, I-PALIA has been implemented in the PMApp tool (Ibanez-Sanchez et al., 2023) as a selectable Discovery algorithm. As PMApp accepts the TPA natively, the connectivity with the *Experiment viewer* was immediate. Figure 7 shows a TPA within PMApp after a Discovery.

Figure 8 shows a full example of the different stages of I-PALIA algorithm implemented in Java.[4] For this example we have used the log $\mathcal{L} = \{\, ADHDDHDHA,\ ACBB_1DDHA,\ ABB_1CDHDDHA \,\}$.

[4] https://github.com/dtu-pa/palia

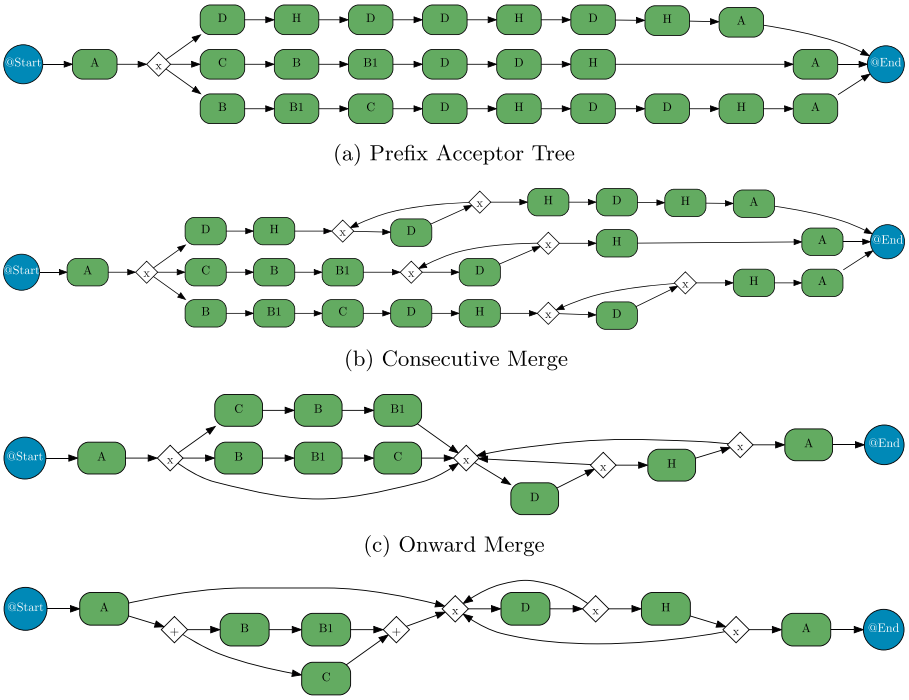


Fig. 8 I-PALIA Discovery Stages with log $\mathcal{L} = \{ADHDDHDHA, ACBB_1DDHA, ABB_1CDHDDHA\}$

Figure 8.a shows the result of applying the *Prefix Acceptor Tree* algorithm. As can be seen, the process shows a tree that represents the three traces of the log generalized by the prefix. In this case, the three traces share the same prefix A that becomes the first trace node, and then each trace is represented in a different branch.

The *Consecutive Merge Stage* is illustrated in Fig. 8.b. Here, it is possible to see the generalization of consecutive nodes. In this case, the three traces have the double repetition of the D action in the middle of the trace. I-PALIA generalise this, defining a self-loop in the three traces.

In Fig. 8.b we can detect some γ that can be compatible. For example, the end of the three traces is reached by three nodes with the same action A. This is a clear Strict Compatible pattern; in the first trace, the subsequence DH is an Inline Compatible Pattern, and the same pattern can be applied in the different traces as a Simple Compatible Pattern. Figure 8.c is the result of applying the Onward Merge stage with an Inline Compatible transition merge mode, until no more Inline compatibilities are available on the flow.

The last step is the *Parallel Merge Stage* that aims to identify parallelism in the flow. In Fig. 8.c we can see a parallel region identifiable by I-PALIA. The region between the first A (n_{Split}) and D (n_{sync}) represents a region suitable to be parallelized. I-PALIA detects B, and C as the head of parallel sequences and identifies BB_1 as a sequence according to the

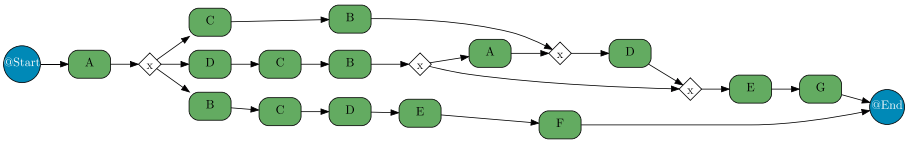
definition of parallelism. Figure 8.d presents the result of Parallel Merge and, consequently the final result of I-PALIA.

The selection of an adequate Transition Merging mode is key to success in having a good ratio between generalization and identification of duplicated tasks. Figure 9 shows an example applying the implemented version of I-PALIA over the log $\mathcal{L} = \{ABCDEF, ACBDEG, ADCBEG, ADCBADEG\}$ using the different three Transition Merging Models. As can be seen in this example the results can vary significantly. As expected, the Strict Compatible transition model (a) is the one that produces more duplicated tasks, that suppose less generalization. On the other side, the Simple Compatible Transition Mode is the one that performs a greater generalization reducing at the maximum the number of duplicated tasks.

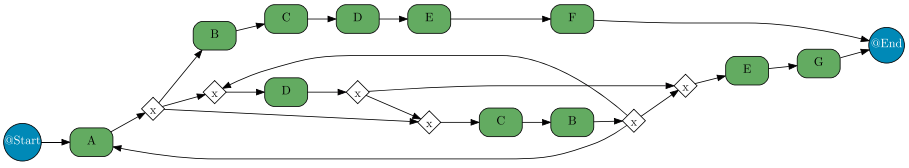
6 Performance evaluation

In order to evaluate the effectiveness of the algorithm, we decided to compare the models resulting from the mining of the process using the state of the art algorithms and tools for control-flow discovery. We designed a process (available in Fig. 10) specifically expressing the challenges of duplicated activities; in the case of the process, it is activity A.

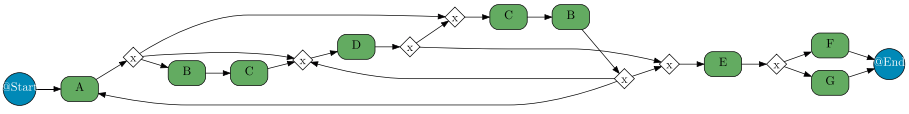
In addition to that, we incorporated behavior coming from the most common workflow patterns (Aalst et al., 2003): sequences, parallel split, synchronization, exclusive choice, and simple merge.



(a) Strict Compatible Transition Mode



(b) Inline Compatible Transition Mode



(c) Simple Compatible Transition Mode

Fig. 9 PALIA Discovery results with different Transition Merging modes with log $\mathcal{L} = \{ABCDEF, ACBDEG, ADCBEG, ADCBADEG\}$

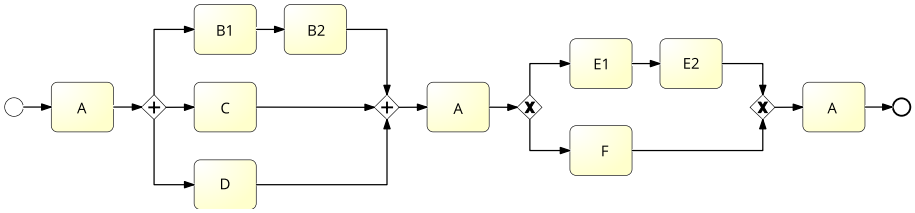
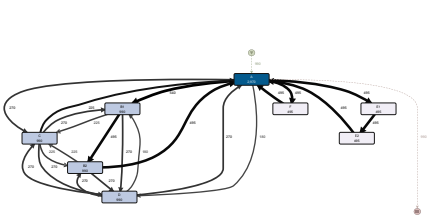
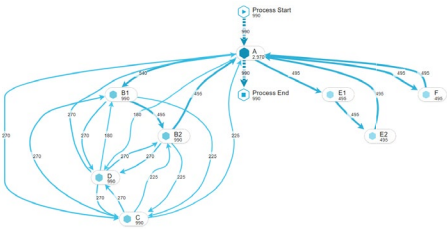


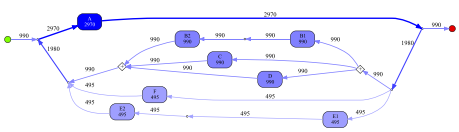
Fig. 10 The reference process we used for our simulation



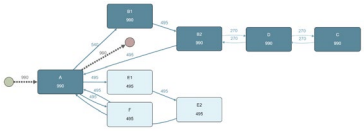
(a) Model discovered with Disco



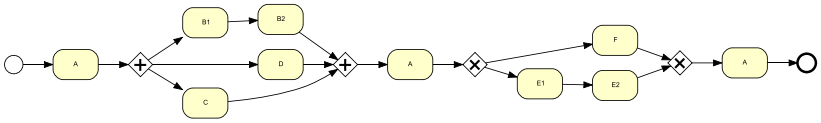
(b) Model discovered with Celonis



(c) Model discovered with Inductive Miner



(d) Model discovered with Apromore



(e) Model discovered with I-PALIA

Fig. 11 Results of the mining of the log using different techniques

We simulated the process using the tool Purple (Burattin et al., 2022)[5], configuring the tool for the rediscoverability purpose, which led to an event log with 990 traces and 8415 events.

We mined this log with I-PALIA as well as with Fluxicon Disco[6], Celonis[7], Apromore[8] and the Inductive Miner (Leemans et al., 2014a). The results are available in Fig. 11. While the model mined with I-PALIA matches perfectly the expected one, all other mining tools

[5] See http://pros.unicam.it:4300/.

[6] See https://fluxicon.com/disco/.

[7] See https://www.celonis.com/.

[8] See https://apromore.com/.

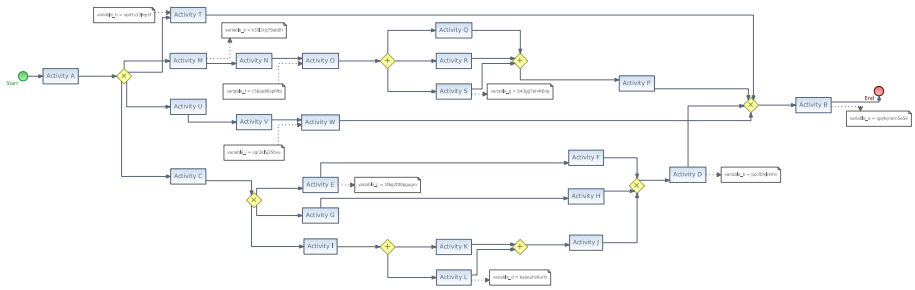


Fig. 12 Random model generated for the stress test

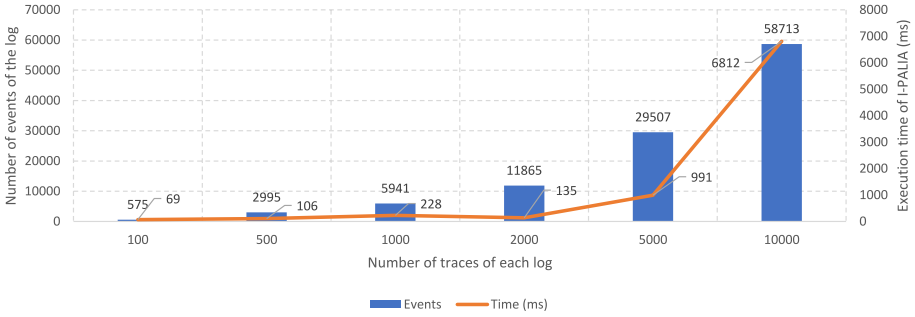


Fig. 13 Stress test on the algorithm against log with different sizes

and algorithms fail at extracting a model that resembles the original one. The main issue, as expected, is the duplication of activity A which is observed at the beginning, in the middle, and at the end of the process. This causes all models to show unstructured behavior, quite similar for each mining algorithm: the process can start with activity A and finish immediately, or can have repetitions of the combinations of the other activities (the part before and after the A in the middle).

In another battery of tests, we aimed to learn something about the computational performance of the I-PALIA implementation. For this purpose, we generated a random process model using PLG2 (Burattin, 2016) (cf. Fig. 12). With this model, we generated 6 event logs with 100, 500, 1000, 2000, 5000, and 10000 traces. These logs were mined with I-PALIA and we monitored the execution time. The tests were performed on a standard laptop, equipped with Java 1.15(TM) SE Runtime Environment on Windows 10 Enterprise 64bit, an Intel Core i7-7500U 2.70GHz CPU and 16GB of RAM. Results are reported in Fig. 13. As the plot shows, the time required to process is not negligible and could grow quite quickly. Considering the biggest log, we had 10000 traces and 58713 events, and our implementation of I-PALIA took almost 7 seconds for the actual mining.

For the final test, we decided to use I-PALIA to analyze a real-world event log. Specifically, we considered the SEPSIS event log (Mannhardt & Blinde, 2017).[9] This log contains events referring to sepsis cases from a Dutch hospital, where one case refers to the pathway

[9]The event log is publicly available at https://doi.org/10.4121/uuid:915d2bfb-7e84-49ad-a286-dc35f063a460.

through the hospital of one patient. The events have been recorded via the ERP system of the hospital, and the log contains 1050 cases with 15214 total events, where each event refers to one of the possible 16 different activities. For our tests, we decided to mine this log using Celonis, Fluxicon Disco, I-PALIA (with the Strict Compatible Transition Mode), and Inductive Miner. We set the parameters of the commercial tools and Inductive Miner to show all the possible behaviors (similarly to what I-PALIA does). The processing time for the log, in the case of I-PALIA, was about 5 seconds. All the results are reported in Fig. 14. As typical for the healthcare domain (Munoz-Gama et al., 2022), the resulting models show spaghetti-like behavior; however, it is interesting to note that in the case of I-PALIA, the algorithm manages, by duplicating activities, to provide a structured model in the beginning (cf. Fig. 14c), while towards the end of the process, the behavior becomes fundamentally challenging to tackle even for I-PALIA. In the Inductive Miner case, the situation is slightly different: the model looks very compact and, in principle, easier to understand, but a deeper analysis reveals that the model starts with a parallel split followed by optionality for each activity, which can also be repeated infinitely many times. From a quantitative point of view, comparing fitness (Berti & Aalst, 2019) and precision (Berti & Aalst, 2019) reveals the issue: as Table 1 shows, Inductive Miner achieves perfect fitness, but its precision is very low. On the other hand, I-PALIA manages to get a fairly good fitness and an almost-perfect precision. Combining these two aspects into the F-score (i.e., harmonic mean) shows that I-PALIA outperforms Inductive Miner in this case.

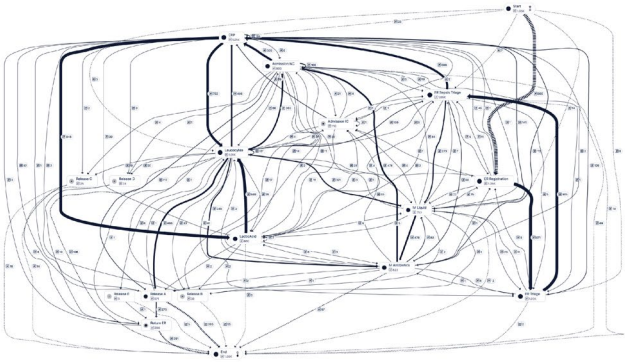
Finally, it is worth noting that while the models mined with Celonis, Disco, and Inductive Miner have 16 event nodes (as expected), the I-PALIA model has 62. This duplication of activities, while increasing the absolute model size, is what helps to deliver a more structured – at least in the first half – and more precise model.

7 Discussion

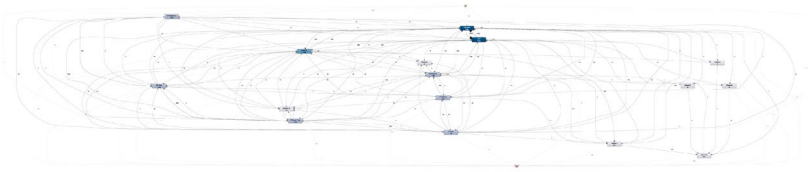
Despite significant progress in process discovery, certain challenges remain unresolved. In particular, the accurate identification and representation of duplicated activities continues to be a limitation for most existing algorithms. These open issues highlight the need for novel methods that can address such complexities while maintaining model precision and interpretability. In this paper, we propose an algorithm capable of identifying activity duplication, thereby improving the understanding of the model under certain circumstances.

There are algorithms available in literature that presents similarities to our proposal. State-based synthesis approaches such as the one presented in Aalst et al. (2006) utilize a two-step method: first, constructing a transition system from the event log, and subsequently applying region theory to synthesize the corresponding process model. While these approaches share similarities with our method in terms of employing an intermediate representation, our grammar inference-based approach offers explicit support for handling duplicated activities in BPMN models, which represents a distinctive contribution compared to region-based synthesis techniques.

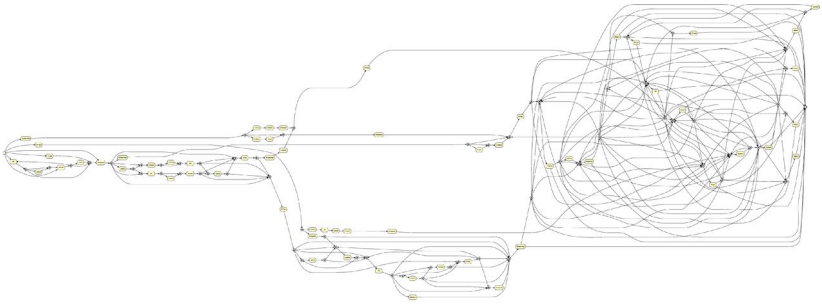
Regarding the discovery of nested patterns, it is important to clarify the current capabilities and limitations of our algorithm. Our approach analyzes all transitions in the process model and is able to identify a wide range of structures, including large loops and nested choices. Notably, loops and choices nested within parallel blocks can often be detected suc-



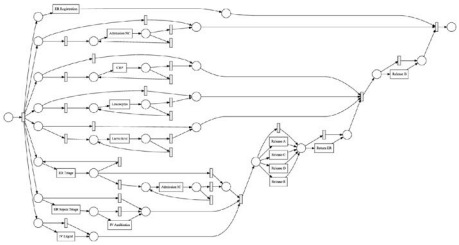
(a) Model discovered with Celonis



(b) Model discovered with Disco



(c) Model discovered with I-PALIA



(d) Model discovered with Inductive Miner

Fig. 14 The SEPSIS log mined using Celonis, Fluxicon Disco, I-PALIA, and Inductive Miner

Table 1 Fitness, precision, and F-score for I-PALIA and Inductive Miner on the SEPSIS log

	Fitness	Precision	F-score
I-PALIA	0.53	**0.99**	**0.69**
Inductive Miner	**1**	0.25	0.40

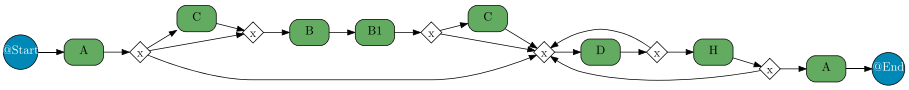


Fig. 15 Over-generalization in onward merge stage with simple compatible transition merge mode with the experiment of the Fig. 8

cessfully under certain circumstances. However, the detection of parallelism is performed specifically between the corresponding split and join nodes. As a consequence, the current implementation is not able to discover nested parallelisms, which may limit the expressiveness of the resulting model in scenarios involving complex parallel constructs. Addressing this limitation and improving the detection of nested parallel structures will be an important avenue for future research.

According to the selection of the best Transition Merge mode, in general, it depends totally on the task to be performed. However, based on our experience some best practices should be taken into account:

- The Simple Compatible Transition Mode is the one that has a greater maximization and, for that, is the one that usually better reduces the "spaghetti effect". On the other hand, the Strict Mode has the most limited generalization and, for that, creates bigger models. So, in general, in high variability problems is better to start with a Simple Compatible Mode to allow a better understanding of the model
- The Simple Compatible Mode can produce an over-generalization problem that can prevent the identification of parallel regions in I-PALIA Algorithm. See an example of this problem in Fig. 8. The Onward Merge Stage with Inline Transition Merge produces a flow (c) where the Parallel Merge algorithm can detect the parallel region. However, if instead of the Inline Transition Merge Mode, we would have applied the Simple Compatible Transition Merge Mode, the transitions $B \rightarrow B_1$ have also been merged, making the identification of the parallel region more difficult. Figure 15 shows the resultant workflow after applying Onward Merge with Simple Compatible Transition merge mode with this example. The result is less understandable than the one with Inline Mode. A solution to this problem is to perform a first run of the Onward Merge algorithm with an Inline Transition Mode and, after the Parallel Merge, perform a second run of Onward Merge with an Simple Compatible Transition Merge Mode.
- In terms of performance, the Strict Mode is the most efficient because the merge pivots on the incoming transitions of the nodes, and each merge pass can done simply by running through the nodes. On the other hand, the Inline Mode is the one that requires more computation because, in addition to running through the γ transitions, it is required to state that the transitions are in the same branch, and this is relatively hard to compute.

According to computational performance, While the algorithm shows very promising results, it still suffers from important issues. The most important ones are the current lack

of robustness to noise and its computational complexity. Regarding the lack of robustness to noise, this is certainly a problem that makes the algorithm not mature enough for many industrial settings, however, we believe this is an issue that can easily be addressed by considering frequencies of the direct following relations and applying some threshold on those before applying the rest of the computation. A more fundamental challenge regards the complexity of the algorithm. Right now, several steps and needed in order to reach the goal, and each of these contributes substantially to the complexity. For small logs the mining time is acceptable, but it grows quickly as the numbers of traces and events grow. Optimizations can be employed also in this case, both in terms of implementation (e.g., multi-threading) as well as more conceptual ones.

8 Conclusion and future work

In this paper, we presented I-PALIA, a process mining algorithm for control-flow discovery. The algorithm is capable of synthesising BPMN process models containing all basic workflow patterns (i.e., sequence, parallel split, synchronization, exclusive choice, and simple merge) as well as duplicated activities. The algorithm, which has a publicly available implementation as both ProM package as well as a Java Maven dependency has been tested and evaluated both qualitatively (against state-of-the-art tools) as well as quantitatively on logs of different sizes. The algorithm is also implemented in the research platform PMApp.

I-PALIA algorithm generates TPAs with a limited representativity, an interesting extension of the algorithm is to incorporate new stages, or incorporate some heuristics for relabeling that can identify more complex workflow patterns such as Milestones or Interleaved Parallel Routines. Also, the first stages of I-PALIA (except Parallel Merge) can be used as an entry point for other process mining discovery techniques to enhance his capability making them able to identify duplicated tasks. For example, the result of the Onward Merge can be used as an entry point for the Inductive Miner Algorithm (Leemans et al., 2014b) instead of the use of DFG, that are not able to identify duplicated tasks.

Other future work for I-PALIA certainly includes the extension of the algorithm to become noise tolerant, for example employing frequencies. Additionally improving the performance and the computational complexity of the approach certainly represents a fundamental step towards wider adoption.

Acknowledgements This research has been supported through projects MINE-GUIDE(PID2020-113723RB-C21) funded by MCIN/AEI/10.13039/501100011033 and LIFECHAMPS(Grant Agreement No 875329) under European Union's Horizon 2020 research and innovation program.

Author Contributions These authors contributed equally to this work

Funding Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature. This research has been partially funded by projects MINE-GUIDE(PID2020-113723RB-C21) funded by MCIN/AEI/10.13039/501100011033 and LIFECHAMPS(Grant Agreement No 875329) under European Union's Horizon 2020 research and innovation program. Funding for open access charge: CRUE-Universitat Politècnica de València

Data Availability No datasets were generated or analysed during the current study.

Declarations

Ethical Approval Not Applicable.

Competing interests The authors declare no competing interests.

References

Aalst, W. M. P. (2016). *Process mining* (2nd ed.). Springer. https://doi.org/10.1007/978-3-662-49851-4

Aalst, W. M., Rubin, V., Dongen, B. F., Kindler, E., & Günther, C. W. (2006). Process mining: A two-step approach using transition systems and regions. BPM Center Report BPM-06-30, BPMcenter. org. 6.

Aalst, W. M. P., Weijters, T. A. J. M. M., & Medeiros, A. K. A. (2006). Process mining with the heuristics miner-algorithm. BETA Working Paper Series, WP 166, Eindhoven University of Technology, Eindhoven.

Aalst, W. M. P., Hofstede, A. H. M., Kiepuszewski, B., & Barros, A. P. (2003). Workflow patterns. *Distributed and Parallel Databases, 14*(1), 5–51. https://doi.org/10.1023/A:1022883727209

Alkhammash, H., Polyvyanyy, A., & Moffat, A. (2024). Stochastic directly-follows process discovery using grammatical inference. In *International conference on advanced information systems engineering* (pp. 87–103). Springer

Augusto, A., Conforti, R., Dumas, M., Rosa, M., Maggi, F. M., Marrella, A., Mecella, M., & Soo, A. (2019). Automated discovery of process models from event logs: Review and benchmark. *IEEE Transactions on Knowledge and Data Engineering, 31*(4), 686–705. https://doi.org/10.1109/TKDE.2018.2841877

Berti, A., & Aalst, W. M. P. (2019). Reviving token-based replay: Increasing speed while improving diagnostics. In *Proceedings of ATAED@Petri Nets/ACSD 2019* (pp. 87–103). CEUR Workshop Proceedings.

Buijs, J. C. A. M., Dongen, B. F., & Aalst, W. M. P. (2014). Quality dimensions in process discovery: The importance of fitness, precision, generalization and simplicity. *International Journal of Cooperative Information Systems, 23*(01), 1440001. https://doi.org/10.1142/S0218843014400012

Burattin, A. (2016). PLG2 : Multiperspective process randomization with online and offline simulations. In *Online Proceedings of the BPM Demo Track 2016*. CEUR-WS.org.

Burattin, A., Re, B., Rossi, L., & Tiezzi, F. (2022). A purpose-guided log generation framework. In *Proceedings of BPM 2022*.

Calders, T., Günther, C. W., Pechenizkiy, M., & Rozinat, A. (2009). Using minimum description length for process mining. In *Proceedings of the 2009 ACM Symposium on Applied Computing - SAC '09* (pp. 1451–1455). ACM Press, New York, New York, USA. https://doi.org/10.1145/1529282.1529606. http://portal.acm.org/citation.cfm?doid=1529282.1529606

Daniel, F.,Barkaoui, K., & Dustdar, S. (2011). IEEE task force on process mining: Process mining manifesto. *Business Process Management Workshops* (pp. 169–194). Springer,

Duan, C., & Wei, Q. (2020). Process mining of duplicate tasks: A systematic literature review. In *2020 IEEE International Conference on Artificial Intelligence and Computer Applications (ICAICA)* (pp. 778–784). IEEE.

Fernandez-Llatas, C., & Burattin, A. (2023). I-PALIA: Discovering BPMN processes with duplicated activities for healthcare domains. In *Proceedings of the International workshop on process-oriented data science for healthcare (PODS4H23)*.

Fernández-Llatas, C., Meneu, T., Benedí, J. M., & Traver, V. (2010). Activity-based process mining for clinical pathways computer aided design. In *2010 Annual International Conference of the IEEE Engineering in Medicine and Biology* (pp. 6178–6181). https://doi.org/10.1109/IEMBS.2010.5627760

Fernandez-Llatas, C., Pileggi, S. F., Traver, V., & Benedi, J. M. (2011). Timed parallel automaton: A mathematical tool for defining highly expressive formal workflows. In *2011 Fifth asia modelling symposium* (pp. 56–61). https://doi.org/10.1109/AMS.2011.22

Goedertier, S., Martens, D., Vanthienen, J., & Baesens, B. (2009). Robust process discovery with artificial negative events. *The Journal of Machine Learning Research, 10*, 1305–1340.

Ibanez-Sanchez, G., Fernandez-Llatas, C., Valero-Ramon, Z., & Bayo-Monton, J. L. (2023). Pmapp: An interactive process mining toolkit for building healthcare dashboards. In *Explainable artificial intelligence and process mining applications for healthcare* (pp. 75–86). Springer.

Leemans, S. J. J., Fahland, D., & Aalst, W. M. P. (2014a). Discovering block-structured process models from event logs containing infrequent behaviour. In *Business process management workshops* (pp 66–78).

Leemans, S. J., Fahland, D., & Van Der Aalst, W. M. (2014b). Process and deviation exploration with inductive visual miner. In *12th International Conference on Business Process Management, BPM 2014* (pp. 46–50). CEUR-WS. org.

Lu, X., Fahland, D., Biggelaar, F. J., & Aalst, W. M. (2016). Handling duplicated tasks in process discovery by refining event labels. In *Business Process Management: 14th International Conference, BPM 2016, Rio de Janeiro, Brazil, September 18-22, 2016* (Proceedings 14, pp. 90–107). Springer.

Mannhardt, F., & Blinde, D. (2017). Analyzing the trajectories of patients with sepsis using process mining. In *RADAR+ EMISA 2017* (pp. 72–80). CEUR-ws.org.

Medeiros, A. K. A. (2006). Genetic process mining. PhD thesis, Technische Universiteit Eindhoven.

Munoz-Gama, J., Martin, N., Fernandez-Llatas, C., Johnson, O.A., Sepúlveda, M., Helm, E., Galvez-Yanjari, V., Rojas, E., Martinez-Millana, A., Aloini, D., Amantea, I.A., Andrews, R., Arias, M., Beerepoot, I., Benevento, E., Burattin, A., Capurro, D., Carmona, J., Comuzzi, M., . . . Zerbato, F. (2022). Process mining for healthcare: Characteristics and challenges. *Journal of Biomedical Informatics, 127*, 103994. https://doi.org/10.1016/j.jbi.2022.103994

OMG (2009). Business Process Model and Notation (BPMN) - Version 2.0, Beta 1.

Peterson, J. L. (1977). Petri nets. *ACM Computing Surveys (CSUR), 9*(3), 223–252. https://doi.org/10.1145/356698.356702

Rojas, E., Fernández-Llatas, C., Traver, V., Munoz-Gama, J., Sepúlveda, M., Herskovic, V., & Capurro, D. (2017). Palia-er: Bringing question-driven process mining closer to the emergency room. In *15th International Conference on Business Process Management (BPM 2017)*.

Russell, N., Hofstede, A. H. M., Aalst, W. M. P., & Mulyar, N. (2006). Workflow control-flow patterns: A revised view. BPM Center Report BPM-06-22, BPMcenter. org.

Stotts, P. D., & Pugh, W. (1994). Parallel finite automata for modeling concurrent software systems. *Journal of Systems and Software, 27*(1), 27–43.

Vázquez-Barreiros, B., Mucientes, M., & Lama, M. (2016). Enhancing discovered processes with duplicate tasks. *Information Sciences, 373*, 369–387. https://doi.org/10.1016/j.ins.2016.09.008