Improving Precision in Process Trees Using Subprocess Tree Logs*

Christian Rennert[0000−0003−4614−6171] (✉) and
Wil M. P. van der Aalst[0000−0002−0955−6940]

Chair of Process and Data Science (PADS), RWTH Aachen University, Germany
{rennert,wvdaalst}@pads.rwth-aachen.de

Abstract. Process mining is a family of techniques that provide tools for gaining insights from processes in, for example, business, industrial, healthcare and administrative settings. Process discovery, as a field of process mining, aims to give a process model that describes a process given by an event log. A process model describes an underlying process well if it contains all behavior relevant (fitness) and if it does not model behavior that is not contained in the event log (precision). The Inductive Miner (IM) family provides algorithms to find process models on complex event logs efficiently and in an easy-to-understand process model representation using process trees. Due to its characteristics, the IM family is one of the state-of-the-art discovery algorithms and is implemented in software of market-leading process mining vendors. Nevertheless, process trees and in particular those discovered by the IM can have imprecise parts. In this work, we combine existing work and present an approach that replaces such parts with more precise parts while preserving fitness. In addition, we demonstrate the frameworks applicability and utilization by improving process trees discovered by the IM, using the IM itself. Further, guarantees on the preservation of fitness and precision are given. Our experiments clearly show that our techniques can be applied to real-life event logs and that they lead to an improvement in precision.

Keywords: Process Tree · Event Data · Process Discovery · Process Enhancement.

1 Introduction and Related Work

Information systems are widely used to collect organizational data related to processes performed. This data are so-called *event data*, where an event corresponds to an activity that is executed for a running instance of a process. By considering a temporal relation between events, the events of each running instance can be translated to a case and multiple cases can be combined into an *event log*. In general, event data may contain further attributes that are not considered in this work. Using event logs, one can use *process mining* to derive insights, value, and actions. Process mining can be used, to evaluate and improve processes, e.g., in terms of sustainability, fairness, productivity or resource consumption.

* We thank the Alexander von Humboldt (AvH) Stiftung for supporting our research.

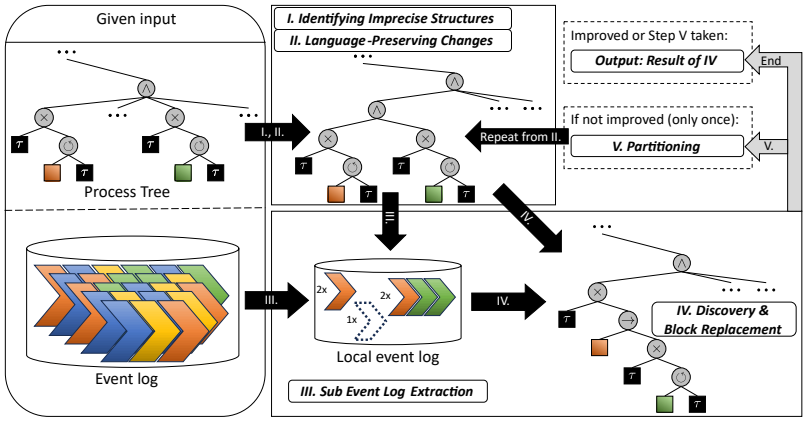

Fig. 1: Overview of process tree projection & replacement (PtR) framework.

Process discovery is a subfield of process mining that aims to discover a *process model* given an event log that can be taken as a starting point for further analysis. In general, a process model describes the possible, not necessarily sequential, flow of activities in a process. Process models are evaluated against four driving quality criteria. A process model is considered a good representation if it represents all running instances of the process seen in the event log (*fitness*), if it does not allow for running sequences unrelated to what is present in the event log (*precision*), if it is easy to understand for a reader (*simplicity*), and if it allows for running sequences not supported to what is seen in the event log but what is likely to happen as well, i.e., if it is not overfitting (*generalization*). For real-life event logs, in most cases it is not possible to satisfy all four quality criteria with one model.

In this work, we focus on *process trees* as a class of process models. Due to their block-structure, process trees are easy to understand. The Inductive Miner (IM) [5], as the state-of-the-art algorithm for discovering process trees, does not guarantee to return the most precise process tree given an event log. A process tree that is not the most precise can be improved using the property that blocks can be replaced with more precise blocks without having to consider the entire tree, improving overall precision while maintaining fitness. Existing theory is taken as a starting point for the approach presented and evaluated in this work. The introduced process tree projection & replacement (PtR) framework follows the procedure shown in Figure 1: Given as input an (intermediate) process tree and an event log, first, imprecise structures are identified; second, if necessary, language-preserving changes are made to separate the imprecise structure from other structures; third, the sub event log for the separated imprecise structure is extracted; and fourth, it is used as input to discover a process tree that replaces the previous imprecise structure. As an additional fifth step, a partitioning based on the activities contained in the separated process tree can be applied at most once, if no improvement has occurred after the fourth step. This would result in the second through fourth steps having to be performed again.

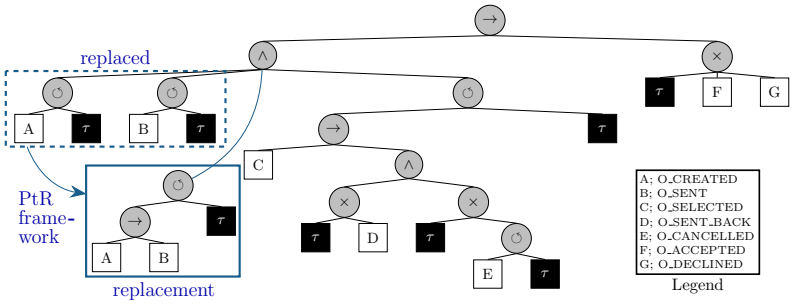

Fig. 2: Comparative view before and after the application of the PtR framework to a process tree discovered by the IM for the offer process filtered BPI12 event log.

An example for the application of the PtR framework is given in Figure 2 visualizing a process tree discovered by the IM on the BPI12 challenge log[1] that is filtered for the events related to the offer process. Here, the replaced part of the process tree and the replacement are indicated. The application of our framework improves the precision as it restricts the process tree such that each creation of an offer is eventually followed by the offer being sent. Further, fitness is preserved by the framework.

There are different process discovery algorithms existent. Region-based discovery algorithms such as the ILP-Miner [10] or the Prime Miner [3] guarantee a strong relation between the input event log and the returned process model and therefore return process models with high precision in their class of process models but are limited by their representational bias (e.g. no use of silent transitions). Furthermore, region-based approaches tend to not handle noise and tend to have a high runtime. Here, the Inductive Miner (IM) can be put into relation as it achieves a good generalization and is computationally fast. Similar to the Split Miner [2], the IM discovers process models solely using the directly-follows relation between events in the event log and therefore reducing its complexity, which may result in less precise models than models returned by region-based techniques. Additionally, the Inductive Miner does not guarantee to discover the most precise process tree for a given event log due to its greedy approach.

The algorithms and approaches proposed in this work are related to the field of incremental process discovery [4] and to repair techniques in the field of process enhancement [6]. In particular, the work by Schuster et al. [9] is the most related to the ideas presented in this work. In their work, the authors aim to incrementally discover and adapt process trees by including traces successively into the behavior modeled. In this work, we use their proposed techniques for tree modification and sublog identification. However, we extend their work by giving a framework that tackles precision improvement in general and as we give a formal reasoning on the guarantees of the techniques. Conceptually, the framework presented is related to a similar work on imprecise structure replacement in Petri nets [8]. As process trees can be translated into block-structured

[1] All event logs used in this work are taken from https://data.4tu.nl/.

Petri nets, the results of this work can be taken as starting point for refining the imprecise structure replacement on Petri nets to also consider block structures.

The remainder of this work is structured as follows. Section 2 presents mathematical notations and concepts related to process mining used in this work. Section 3 introduces the framework presented in this work in detail. Section 4 describes and proves the guarantees of the steps taken in our framework. In Section 5, the framework is evaluated using process trees discovered on real-life events logs with the Inductive Miner without noise filtering. Lastly, in Section 6, this paper is concluded and outlook for future work is given.

2 Preliminaries

Basic Notations. A multiset, e.g., $[a, b, b, a, b, c] = [a^2, b^3, c]$ can contain an element multiple times. We refer to the set of multisets over a set X as $\mathbb{M}(X)$. While sets and multisets are unordered, sequences are ordered and can contain an element multiple times, e.g., $\langle a, b, a, a, c\rangle \in \{a, b, c\}^*$. We denote the projection of a sequence $\sigma \in X^*$ on a subset of activities $Y \subseteq X$ by $\sigma{\restriction_Y}$. For example, $\langle a, b, c, b, a\rangle{\restriction_{\{a,c\}}} = \langle a, c, a\rangle$. For a tuple $t = (x_1, x_2, \ldots, x_n)$ with $n \in \mathbb{N}$ and $i \in [1, n]$ we denote by π_i the selector function selecting the i-th element in a tuple, i.e., $\pi_i(t) = x_i$. For example, $\pi_3((a, b, c, d)) = c$. We uplift all functions applicable for single elements of a sequence to be applicable to the full sequence, i.e., for a function $f\colon X \to Y$ and a sequence $\sigma = \langle x_1, x_2, \ldots, x_n\rangle$ we write $f(\sigma) = \langle f(x_1), f(x_2), \ldots, f(x_n)\rangle$. Similarly, given a function applicable to two elements of two sets $X \subseteq \mathcal{X}, Y \subseteq \mathcal{Y}$, we uplift the function to all combinations of elements, i.e., for a function $\circ\colon \mathcal{X} \to \mathcal{Y}$ we write $X \circ Y = \bigcup_{x \in X, y \in Y} \{x \circ y\}$. For any two traces $\sigma \in X^*, \sigma' \in Y^*$ and for any two elements $x \in X, y \in Y$, we denote the shuffle operation $\diamond\colon (X^*, Y^*) \to (X \cup Y)^*$ recursively with: $\sigma \diamond \langle\rangle = \sigma$, $\langle\rangle \diamond \sigma' = \sigma'$ and $(\sigma \cdot x) \diamond (\sigma' \cdot y) = (((\sigma \cdot x) \diamond \sigma') \cdot \langle y\rangle) \cup ((\sigma \diamond (\sigma' \cdot y)) \cdot \langle x\rangle)$, i.e., $\langle a, b\rangle \diamond \langle c, d\rangle = \{\langle a, b, c, d\rangle, \langle a, c, b, d\rangle, \langle a, c, d, b\rangle, \langle c, a, b, d\rangle, \langle c, a, d, b\rangle, \langle c, d, a, b\rangle\}$.

Event Logs and Process Trees. The universe of activities (e.g. actions or operations) is denoted by $\mathscr{A}$. A trace $\sigma \in \mathscr{A}^*$ is a finite sequence of activities. The universe of traces is denoted by $\mathscr{T}$. A log $L \in \mathbb{M}(\mathscr{A}^*)$ is a multiset of traces. We represent the behavior described by an event log using process trees. A process tree is an hierarchical process model. Every leaf of a process tree is labeled with an activity, while every other node is labeled with an operator $\oplus = \{\rightarrow, \wedge, \times, \circlearrowleft\}$ such as the sequential operator $\rightarrow$, the parallel operator $\wedge$, the exclusive choice operator $\times$ and the loop operator $\circlearrowleft$. Every operator node can have an arbitrary, non-empty set of children except for the loop operator $\circlearrowleft$ that has two children exactly. For the $\rightarrow$ and $\circlearrowleft$ operator, the order of children is relevant and taken into account by totally ordering the edges in the process tree. A so-called silent activity is labeled with the designated silent activity label $\tau \notin \mathscr{A}$. Further we describe process trees as follows.

Definition 1 (Process Tree). *A process tree is a directed acyclic graph defined as a quintuplet $T = (V, E, A, l, r)$, where V is a finite set of nodes, $E \subseteq V \times V$*

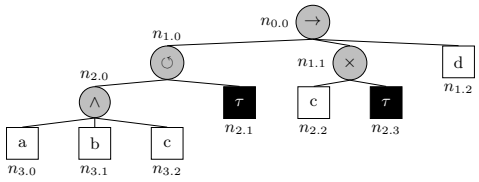

Fig. 3: Process Tree $T_p = (V_p, E_p, A_p, l_p, r_p) = (\{n_{0.0}, n_{1.0}, \ldots, n_{3.2}\}, \{(n_{0.0}, n_{1.0}), (n_{0.0}, n_{1.1}), \ldots, (n_{2.0}, n_{3.2})\}, \{a, b, c, d\}, \{(n_{0.0}, \rightarrow), (n_{1.0}, \times), \ldots, (n_{3.2}, c)\}, n_{0.0})$.

is a totally ordered set of edges between nodes, $A \subseteq \mathscr{A}$ with $\tau \notin A$ is a set of non-silent activity labels, $l : V \rightarrow A \cup \{\tau\} \cup \oplus$ for $\oplus = \{\rightarrow, \wedge, \times, \circlearrowleft\}$, and r is the root node of the process tree. Further, the following holds:

- *$T = (\{n\}, \emptyset, \{a\}, \{(n, a)\}, n)$ with $a \in \mathscr{A} \cup \{\tau\}$ is a process tree*
- *given $k \geq 1$ distinct process trees $T_1, T_2, \ldots, T_k$ with $T_l = (V_l, E_l, A_l, l_l, r_l)$ for $l \in [1, k]$, i.e., $\forall_{i,j \in [1,k]} : V_i \cup V_j \neq \emptyset \Rightarrow i = j$ then $T = (V, E, A, l, r)$ is a process tree for which:*
  - *$V = \bigcup_{i \in [1,k]} V_i \cup \{r\}$*
  - *$E = \bigcup_{i \in [1,k]} (E_i \cup \{(r, r_i)\})$*
  - *$A = \bigcup_{i \in [1,k]} A_i$*
  - *$l = \bigcup_{i \in [1,k]} l_i \cup \{(r, op) \mid op \in \oplus\}$*

  where $(op = \circlearrowleft \Rightarrow k = 2)$ and $\forall_{i \in [1,k]} : r \notin V_i$ holds.

A process tree given can be represented using a graphical notation, as shown in Figure 3. Further, a process tree can also be represented textually, e.g., $T_p \mathrel{\widehat{=}} \rightarrow(\circlearrowleft(\wedge(a, b, c), \tau), \times(c, \tau), d)$ for the process tree in Figure 3. For a process tree $T = (V, E, A, l, r)$ and a node $n \in V$ given, we denote the subtree rooted in n as $T_n = (V_n, E_n, A_n, l_n, r_n)$, i.e., $r_n = n$ holds. Further, we define a child function $ch^T(n)$ returning the children of n as a sequence according to the order of edges E by $ch^T(n) = \langle n_1, n_2, \ldots, n_k \rangle$, s.t., $\forall_{(n,n_o) \in E} : n_o \in \{n_1, n_2, \ldots, n_k\}$. For example, given the node $n_{2.0} \in V_p$ of the process tree T_p shown in Figure 3, we conclude the subtree $T_{n_{2.0}}$ rooted in $n_{2.0}$ to be $T_{n_{2.0}} \mathrel{\widehat{=}} \wedge(a, b, c)$ and its children $ch^{T_p}(n_{2.0})$ to be $ch^{T_p}(n_{2.0}) = \{n_{3.0}, n_{3.1}, n_{3.2}\}$.

Each process tree has a non-empty set of valid sequences of node visits, so-called running sequences. A running sequence is defined recursively as follows.

Definition 2 (Process Tree Running Sequences). *Let $T = (V, E, A, l, r)$ be a process tree. We define its running sequences $\Omega_T \subseteq V \times (A \cup \{\tau, op, cl\})^*$ with $op(en), cl(ose) \notin A$ recursively as:*

- *$\Omega_T = \{\langle (r, l(r)) \rangle\}$ iff r is a leaf node, i.e., $E = \emptyset$ holds,*
- *$\Omega_T = \{\langle (r, op) \cdot \Omega_{T_{n_1}} \cdot \Omega_{T_{n_2}} \cdot \ldots \cdot \Omega_{T_{n_k}} \cdot (r, cl) \rangle\}$ for $k \geq 1$, $ch^T(r) = \langle n_1, n_2, \ldots, n_k \rangle$ iff r is labeled as sequential node, i.e., $l(r) = \rightarrow$ holds,*
- *$\Omega_T = \{\langle (r, op) \cdot \bigcup_{i \in [1,k]} \Omega_{T_{n_i}} \cdot (r, cl) \rangle\}$ for $k \geq 1$, $ch^T(r) = \langle n_1, n_2, \ldots, n_k \rangle$ iff r is labeled as exclusive choice node, i.e., $l(r) = \times$ holds,*
- *$\Omega_T = \{\langle (r, op) \cdot (\Omega_{T_{n_1}} \diamond \Omega_{T_{n_2}} \diamond \ldots \diamond \Omega_{T_{n_k}}) \cdot (r, cl) \rangle\}$ for $k \geq 1$, $ch^T(r) = \langle n_1, n_2, \ldots, n_k \rangle$ iff r is labeled as parallel node, i.e., $l(r) = \wedge$ holds,*

– $\Omega_T = \{\langle (r, op) \cdot \sigma_1 \cdot \sigma'_1 \cdot \sigma_2 \cdot \sigma'_2 \ldots \cdot \sigma'_{m-1} \cdot \sigma_m \cdot (r, cl)\rangle \mid m \in \mathbb{N} \wedge \sigma_m \in \Omega_{T_{n_1}} \wedge \forall_{i \in [1,m-1]} \colon \sigma_i \in \Omega_{T_{n_1}} \wedge \sigma'_i \in \Omega_{T_{n_2}}\}$ *for* $ch^T(r) = \langle n_1, n_2\rangle$ *iff* r *is labeled as loop node, i.e.,* $l(r) = \circlearrowleft$ *holds,*

Given all possible running sequences existing in a tree, we can conclude the language of the process tree $\mathcal{L}(T)$ by considering the occurrences of the leaves with non-silent activities in the running sequence.

Definition 3 (Process Tree Language). *Let* $T = (V, E, A, l, r)$ *be a process tree. We define the corresponding language by* $\mathcal{L}(T) = \{(\pi_2(\omega))\restriction_A \mid \omega \in \Omega_T\}$.

For example, the running sequences of the subprocess tree $T_{n_{1.1}} \mathrel{\widehat{=}} \times(c, \tau)$ of T_p shown in Figure 3 are $\Omega_{T_{n_{1.1}}} = \{\langle (n_{1.1}, op), (n_{2.2}, c), (n_{1.1}, cl)\rangle, \langle (n_{1.1}, op), (n_{2.2}, \tau), (n_{1.1}, cl)\rangle\}$ and the process tree language is $\mathcal{L}(T_{n_{1.1}}) = \{\langle c\rangle, \langle\rangle\}$. We say that a running sequence ω is corresponding to a trace σ if the occurrences of the leaves labeled with an activity from set of non-silent activities A_v match with the trace σ, i.e., if $(\pi_2(\omega))\restriction_{A_v} = \sigma$ holds. For a running sequence given, we can identify all subtraces modeled by a subprocess tree. The union of all such subtraces corresponds to a sub event log of the subprocess tree.

Definition 4 (Sub Event Log of a Subprocess Tree (cf. [9])). *Let* $T = (V, E, A, l, r)$ *be a process tree and* $T_s = (V_s, E_s, A_s, l_s, r_s)$ *be a subtree of* T*, i.e.,* $r_s \in V$ *holds. Further, let* ω *be a running sequence on* T*. Then we define the sub event log* $\omega\restriction_{T_s} \in \mathbb{M}(\mathcal{L}(T_s))$ *of the subprocess tree* T_s *considering the running sequence* $\omega = \langle a_1, a_2, \ldots, a_n\rangle$ *with* $n \in \mathbb{N}$ *as:*

$$\omega\restriction_{T_s} = [\pi_2(\langle a_i, a_{i+1}, \ldots, a_j\rangle\restriction_{(V_s \times A_s)}) \mid i \in [1, n-1] \wedge j \in [i+1, n] \wedge \forall_{k \in [i,j]} \colon (a_k = (T_s, op) \Leftrightarrow k = i \wedge a_k = (T_s, cl) \Leftrightarrow k = j)].$$

In this work, we only consider nodes for replacement that have a common father node that is labeled with the parallel operator, i.e., we only consider concurrent (sub-)process trees to be replaced. Further, as we want to keep precise (sub-)process trees which are concurrent as well, we may not always want to find the sub event log of all concurrent (sub-)process trees, but rather the sub event log of a partition of them. Therefore, we present a technique to separate a partition of (sub-)process trees from their siblings. This is achieved using expansion rules that lower the partition while leaving their siblings unchanged. This preserves the language of the process tree, as we show in Section 4.

Definition 5 (Lowering a Set of Children of a $\wedge$-Rooted Tree (cf. [9])). *Given* $T = (V, E, A, l, r)$ *a process tree and* T_n *a subprocess tree with root* $n \in V$*,* $ch^T(n) = \langle cn_1, cn_2, \ldots, cn_j\rangle$ *for* $j \in \mathbb{N}$ *and* $l(n) = \wedge$*. Then we can add a node* n_{low}*, where* $V_c = ch^{T'}(n_{low}) \subseteq ch^T(n)$ *holds, while preserving the language of the tree* T *resulting in a process tree* $T' = (V', E', A, l', r)$ *where:*

$$V' = V \cup \{n_{low}\}$$
$$E' = (E \setminus \{(n, cn) \mid cn \in V_c\}) \cup \{(n, n_{low})\} \cup \{(n_{low}, cn) \mid cn \in V_c\}$$
$$l' = l \cup \{(n_{low}, \wedge)\}.$$

Given the process tree $T_p \mathrel{\widehat{=}} \rightarrow(\circlearrowleft(\wedge(a,b,c),\tau),\times(c,\tau),d)$ shown in Figure 3, we obtain $T_p \mathrel{\widehat{=}} \rightarrow(\circlearrowleft(\wedge(b,\wedge(a,c)),\tau),\times(c,\tau),d)$ by lowering the set $\{n_{3.0}, n_{3.2}\}$. Last, we give a comparative view on fitness and precision between process trees.

Definition 6 (Fitness and Precision Comparison). *Let L be an event log and T, T' be two process trees. T' is at least as fitting as T considering the event log L, if every trace $\sigma \in L$ that is in the language of T is also in the language of T', i.e., $\sigma \in \mathcal{L}(T) \Rightarrow \sigma \in \mathcal{L}(T')$. T' is at least as precise as T, if every trace $\sigma \in \mathscr{T}$ that is in the language of T' is also in the language of T, i.e., $\sigma \in \mathcal{L}(T') \Rightarrow \sigma \in \mathcal{L}(T)$. T' is more precise than T, if there is an additional trace $\sigma' \in \mathscr{T} \setminus L$ that is in the language of T but not of T', i.e., $\sigma' \in \mathcal{L}(T) \wedge \sigma' \notin \mathcal{L}(T')$.*

3 Process Tree Projection & Replacement Framework

The Inductive Miner analyzes a log and splits the log greedily into two sublogs by projecting the current log onto two distinct sets of activities following defined rules and fall-throughs. For each log analysis, a new node is inserted into the result process tree. Here, the algorithm can be prone to find imprecise subtrees as it does not use look-ahead. An example of such an imprecise subtree is a tree in which at least two activities can be replayed in parallel and arbitrarily often, i.e., in the process tree $\wedge(\times(\tau,\circlearrowleft(a,\tau)),b,\times(\circlearrowleft(c,\tau)))$ this is the case for the subprocess trees containing activities a and c. As this kind of subprocess trees restricts the behavior only marginally, we consider such structures as most promising candidates for replacement and precision improvement.

Our presented approach is not limited to process trees discovered by the Inductive Miner as other discovery algorithms may also discover process trees with imprecise structures. However, given that there are no other known discovery algorithm for process trees, and for reasons of comprehensibility, we restrict ourselves to the Inductive Miner without noise filtering. In the following, we give a short overview of the *Process Tree Projection & Replacement Framework (PtR framework)* presented and used in this work.

PtR framework. We follow the structure shown in Figure 1 consisting of the following steps recursively applied to an event log L and a process tree T:

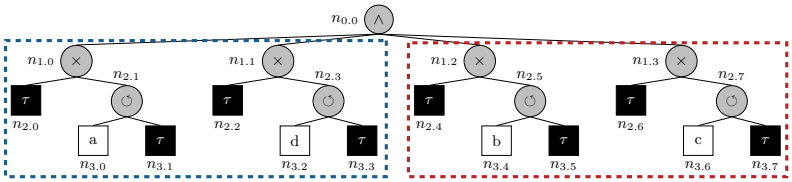

Fig. 4: Process tree T_1 discovered by the IM on the event log L_1. T_{blue} and T_{red} show two partitions and their corresponding lowered subtrees resulting from Step I and II.

Step I - Identification: Identifying imprecise structures as candidates, e.g., multiple flower structures, choice constructs modeling optionality or loop constructs which occur in parallel can be considered.
Step II - Modification: Applying expansion rules to separate the imprecise structures as subprocess trees according to Definition 5, e.g., we lower a set of parallel nodes that are considered as imprecise candidates by adding another parallel node that is a parent of the candidate nodes, while their former parent is the parent of the new node.
Step III - Sub Event Log Extraction: Extracting and uniting all sub event log of the separated subprocess tree, which are obtained according to Definition 4. In case that a running sequence, relevant for the sub event log extraction, does not exist, we obtain a running sequence by using alignments [1].
Step IV - Discovery and Block Replacement: Discovering a replacing process tree where each trace in the sub event log is in its language and potentially replacing the separated structure if the replacement is more precise.
Step V - Partitioning (Fall-Through): If precision is not improved, a fall-through is applied that bi-partitions the children of the separated structure and considers the partitions as new candidates for improval. For each such candidate, the Steps II-IV of the framework are applied.

As an example, we consider T_1 shown in Figure 4 which is discovered by the Inductive Miner on the log $L_1 = [\langle a, d, a, d\rangle, \langle b\rangle, \langle b, c, c, b, c, c\rangle, \langle b, a, d, c, a, c, d\rangle]$. Here, the potential discovery of imprecise structures becomes more apparent as the language $\mathcal{L}(T_1) = \{a, b, c, d\}^*$ of the example process tree T_1 is imprecise considering the input event log L_1. Here, the PtR framework is applicable to the process tree T_1 and the log L_1. This results in the process tree T_1' shown in Figure 5. In detail, the following (intermediate) results are produced:
Step I: The nodes $n_{1.0}, n_{1.1}, n_{1.2}$ and $n_{1.3}$ are in parallel and flower structures and are therefore identified as a candidate for replacement together.
Step II-V: Applying Steps II-III results in the input event log L_1 for which we rediscover the identical process tree in Step IV. Conclusively, we are in Step V as fall-through and thus bi-partition the set of nodes. For the sake of comprehensibility, we follow the optimal partitioning to be applied directly in this

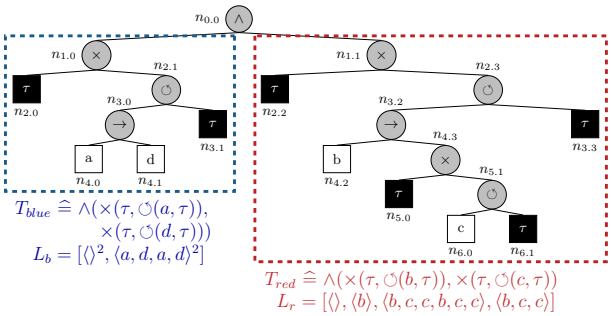

Fig. 5: Process tree T_1' resulting from applying the PtR framework to event log L_1 and the processs tree T_1. The logs L_b and L_r correspond to the logs identified by Step III.

example. Thus, the partitioning results in a blue set of nodes $n_b = \{n_{1.0}, n_{1.1}\}$ and a red set of nodes $n_r = \{n_{1.2}, n_{1.3}\}$, both being hightlighted in Figure 4 and considered as new candidates for replacement.
Step II: We modify T_1 by applying the expansion rules to partition n_b and partition n_r, resulting in a modified process tree $T_i \mathrel{\widehat{=}} \wedge(T_{blue}, T_{red})$ with $T_{blue} \mathrel{\widehat{=}} \wedge(\times(\tau, \circlearrowleft(a, \tau)), \times(\tau, \circlearrowleft(d, \tau))$ and $T_{red} \mathrel{\widehat{=}} \wedge(\times(\tau, \circlearrowleft(b, \tau)), \times(\tau, \circlearrowleft(c, \tau))$.
Step III: Given T_i, we can extract and unite all sub event logs by projecting on both subtrees T_{blue} and T_{red} resulting in $L_{blue} = L_1 \restriction_{T_{blue}} = [\langle\rangle^2, \langle a, d, a, d\rangle^2]$ and $L_{red} = L_1 \restriction_{T_{red}} = [\langle\rangle, \langle b\rangle, \langle b, c, c, b, c, c\rangle, \langle b, c, c\rangle]$.
Step IV: Given the extracted logs, we rediscover process trees using the Inductive Miner on the logs L_{blue} and L_{red} resulting in $T'_{blue} \mathrel{\widehat{=}} \times(\tau, \circlearrowleft(\rightarrow(a, d), \tau))$ and $T'_{red} \mathrel{\widehat{=}} \times(\tau, \circlearrowleft(\rightarrow(b, \times(\tau, \circlearrowleft(c, \tau))), \tau))$ which replace T_{blue} and T_{red} in the process tree T_i resulting in $T'_1 \mathrel{\widehat{=}} \wedge(T'_{blue}, T'_{red})$ shown in Figure 5.

Here, fitness is preserved as the log L_1 remains replayable, while precision for T'_1 is improved since, for example, the trace $\langle c\rangle$ has a running sequence for the process tree T_1 but not for T'_1, i.e., $\langle c\rangle \in \mathcal{L}(T_1) \wedge \langle c\rangle \notin \mathcal{L}(T'_1)$ holds.

4 Guarantees of the Framework

In this section, we discuss the guarantees of the methods proposed. For this purpose, we show that all steps taken do neither reduce fitness nor precision. We start with Step II where we distinguish two cases. In the first case, we do not have to lower a set of nodes such that the input tree remains unchanged and thus the fitness and precision are preserved. In the second case, we lower a (partitioned) set of children $N_c = \{cn'_1, cn'_2, \ldots, cn'_j\} \subseteq ch^T(r) = \{cn_1, cn_2, \ldots, cn_n\}$ with $j, n \in \mathbb{N}$ of a (sub-)process tree $T = (V, E, A, l, r)$. Initially, from Definition 2 and 3, we can conclude that the process tree language of the tree T is $\mathcal{L}(T) = \{\mathcal{L}(cn_1) \diamond \mathcal{L}(cn_2) \diamond \ldots \diamond \mathcal{L}(cn_n)\}$. Lowering the set of children N_c results in a node n_{low} being added such that we obtain a process tree $T' = (V', E', A, l', r)$ according to Definition 5. By construction, $\mathcal{L}(T_{n_{low}}) = \{cn'_1 \diamond cn'_2 \diamond \ldots \diamond cn'_j\}$ and $\mathcal{L}(T') = \{\mathcal{L}(n_{low}) \diamond \mathcal{L}(cn_1) \diamond \mathcal{L}(cn_2) \diamond \ldots \diamond \mathcal{L}(cn_n) \mid \forall_{i \in [1,n]} \colon cn_i \notin N_c\}$ hold, resulting in $\mathcal{L}(T) = \mathcal{L}(T')$ due to the shuffle operator being associative. Conclusively, lowering a set of children does not change the language of a process tree and in particular does not reduce fitness nor precision.

Next, we want to show that applying Steps III and IV does neither reduce fitness nor precision. Towards fitness, we show that each part of all traces replayed in the replaced process tree remains replayable in the replacing process tree. We follow the following line of argument. For each trace a running sequence (possibly derived from an alignment) can be computed. Given a set of running sequences, we can directly determine their language on the replaced process tree. This corresponds to the traces that result from the occurrences of the leaves of the replaced process tree between the replaced process tree's opening and closing in the running sequence. Taking into account that such a language for only running sequence given, is identical to the sub event log computation, we conclude the language of the replaced process tree to be equal to the extraction and union

Table 1: Results of our framework for real-life logs, for which our algorithm and ETC_{All}-precision are computable in alotted time and space.

Log	ETC_{All}-precision			
	Initial	PtR	Difference	Percentage
BPI12 - O events	0.457	0.612	0.155	33.9%
BPI12 - W events	0.431	0.496	0.065	15.1%
BPI13 - closed problems	0.510	0.612	0.102	20.0%
BPI20 - Domestic Declarations	0.216	0.288	0.072	33.3%
BPI20 - Request for Payment	0.236	0.273	0.037	15.7%
Sepsis Cases	0.227	0.234	0.007	3.1%

of all sub event logs as applied in Step III. The resulting log is the basis for the discovery of the replacing process tree and therefore the lower bound of its modeled language (as required in Step IV). Thus, the parts of the running sequences of the event log replayed on the replaced subprocess tree are replayable on the replacing subprocess tree. In conclusion, all traces have an alignment at least as good as before - in particular, fitting traces remain replayable.

Towards precision, we can trivially conclude an overall precision preservation as we do a precision comparison between the replacing and the replaced process tree in Step IV assuring that we do not reduce precision. Such a comparison would be redundant if the discovery algorithm used guarantees that it discovers a process tree whose language is contained in the log obtained by Step III. This is a result of the log obtained by Step III being a subset of the language of the replaced process tree. This property holds as the log obtained by Step III is computed using the running sequences obtained from the input log. Conclusively, the replacing process tree would not allow for more behavior than the replaced process tree resulting in a preservation of precision.

Taking all those arguments into consideration and that further Step I and Step V do not alter the input process tree, there is no step in our framework that reduces either fitness or precision. Thus, both are preserved using our framework.

5 Evaluation

In this section, we evaluate the PtR framework proposed in Section 3 with real-life event logs. We implemented a plugin called "Process Tree Projection & Replacement Framework (PtR framework)" in ProM[2] that applies the framework automatically to an input event log and its corresponding process tree.

For our evaluation, we want to quantitatively evaluate precision by using *ETC_{All}-precision* [7]. We consider pairs of real-life event logs, namely the BPI challenge logs from 2011-2020, the Sepsis event log and the RTFM event logs, and their corresponding discovered process trees discovered by the Inductive Miner without noise filtering. Noise filtering is not used, since, this way, imprecise

[2] Available at https://promtools.org/.

structures are obtained on less complex event logs. Further, ETC_{All}-*precision* computation becomes more probable to be feasible. As imprecise structures we restrict ourselves to flower and loop constructs. In Table 1, we report the precision before and after the application of the PtR framework for only those pairs, where both the running sequence computation of the input event log (Step III in the framework) and the ETC-precision computation of the evaluation finish in allotted time and space. Further, we report the difference of the precision values and its relative improvement given by the percentage of the increasement.

The other pairs of event logs and their process trees contain three pairs (BPI12 - application, BPI13 - open, RTFM) for which no imprecise structure is identifiable, one pair (BPI13 - open) for which no improvement was achieved, one pair (BPI13 - incidents) for which we improve the input process tree but ETC_{All} precision computation is not feasible within allotted time, and for the process tree of all the other pairs imprecise structures are identified, but running sequence computation is not feasible.

Our results show, that for most real-life event logs an imprecise structure is discovered by the Inductive Miner. For all event logs listed in Table 1 but one, we improve precision significantly by at least 15% compared to the initial precision value. This qualitative improvement clearly shows the applicability and relevance of our approach and motivates further research regarding improvement of running times, which is not the focus of this work.

Given our experiments we suggest to improve the IM by optimizing the way it applies certain fall-throughs. Consider the discovered and replaced structure in Figure 2. Here, the IM first evaluates two sublogs, one containing only the activity "O_CREATED" and the other containing only the activity "O_SENT". For both sublogs none of the four standard cuts is found and thus fall-throughs (mainly the activity-concurrent fall-through) are applied greedily. Note that the order of fall-through cuts matters and can result in a non-optimal set of subtrees. We propose to re-evaluate sets of such imprecise subtrees based on which we construct one sublog that allows to discover one more precise subprocess tree.

6 Conclusion and Future Work

In this work, we introduce a framework to improve precision by replacing imprecise structures in process trees. It gives guarantees on the preservation of fitness and precision, and therefore never decreases the quality of its input process tree. Further, if we use the IM within our framework, we inherit IM's desirable features such as soundness and fast computation of process models. Our work is supported by experiments on process trees discovered by the well-known Inductive Miner without noise filtering for which we use the IM itself to replace imprecise structures. Our experiments show that imprecise structures occur for most real-life logs available. For six real-life event logs and their corresponding process trees, we improve precision. This indicates the relevance of our work.

However, since running sequences must be computed for all traces before applying the IM, there are inputs for which our implementation did not run

within the time and space allotted. Therefore, either efficiency can be improved or could result in being a starting point for adaptations within the IM itself. For the latter, the relevant sub event logs would already exist within the IM and could be used as input for further refinement.

Otherwise, the framework presented in this paper can be evaluated for the IM with noise filtering. Here, the results can be taken as starting point for research on the ranking of desired alignments when computing the sub event log, as the quality of the discovered replacing process tree is strongly related to the sub event log given. Further, heuristics for the identification of promising imprecise structures and for checking how and whether to partition potential imprecise structures are missing.

References

1. Adriansyah, A.: Aligning observed and modeled behavior. Ph.D. thesis, Mathematics and Computer Science (2014). https://doi.org/10.6100/IR770080
2. 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). https://doi.org/10.1007/s10115-018-1214-x
3. Bergenthum, R.: Prime miner - process discovery using prime event structures. In: ICPM 2019, Aachen, Germany. pp. 41–48. IEEE (2019). https://doi.org/10.1109/ICPM.2019.00017
4. Kindler, E., Rubin, V.A., Schäfer, W.: Incremental workflow mining based on document versioning information. In: SPW 2005. Revised Selected Papers. Lecture Notes in Computer Science, vol. 3840, pp. 287–301. Springer (2005). https://doi.org/10.1007/11608035_25
5. Leemans, S.J.J., Fahland, D., van der Aalst, W.M.P.: Discovering block-structured process models from event logs - A constructive approach. In: PETRI NETS 2013. Proceedings. Lecture Notes in Computer Science, vol. 7927, pp. 311–329. Springer (2013). https://doi.org/10.1007/978-3-642-38697-8_17
6. de Leoni, M.: Foundations of process enhancement. In: Process Mining Handbook, Lecture Notes in Business Information Processing, vol. 448, pp. 243–273. Springer (2022). https://doi.org/10.1007/978-3-031-08848-3_8
7. Munoz-Gama, J., Carmona, J.: A fresh look at precision in process conformance. In: BPM 2010. Proceedings. Lecture Notes in Computer Science, vol. 6336, pp. 211–226. Springer (2010). https://doi.org/10.1007/978-3-642-15618-2_16
8. Rennert, C., Mannel, L.L., van der Aalst, W.M.P.: Improving the est-miner models by replacing imprecise structures using place projection. In: ATAED'23 at Petri Nets 2023. CEUR Workshop Proceedings, vol. 3424. CEUR-WS.org (2023), https://ceur-ws.org/Vol-3424/paper3.pdf
9. Schuster, D., van Zelst, S.J., van der Aalst, W.M.P.: Incremental discovery of hierarchical process models. In: RCIS 2020, Proceedings. Lecture Notes in Business Information Processing, vol. 385, pp. 417–433. Springer (2020). https://doi.org/10.1007/978-3-030-50316-1_25
10. van der Werf, J.M.E.M., van Dongen, B.F., Hurkens, C.A.J., Serebrenik, A.: Process discovery using integer linear programming. In: PETRI NETS 2008. Proceedings. Lecture Notes in Computer Science, vol. 5062, pp. 368–387. Springer (2008). https://doi.org/10.1007/978-3-540-68746-7_24