Incremental Discovery of Process Models Using Trace Fragments

Daniel Schuster[1,2], Niklas Föcking[1], Sebastiaan J. van Zelst[1,2], and Wil M. P. van der Aalst[1,2]

[1] Fraunhofer Institute for Applied Information Technology FIT, Germany
{daniel.schuster,niklas.foecking,sebastiaan.van.zelst}@fit.fraunhofer.de
[2] RWTH Aachen University, Aachen, Germany
wvdaalst@pads.rwth-aachen.de

Abstract. Process discovery learns process models from event data and is a crucial discipline within process mining. Most existing approaches are fully automated, i.e., event data is provided, and a process model is returned. Thus, process analysts cannot interact and intervene besides parameter settings. In contrast, Incremental Process Discovery (IPD) enables users to actively participate in the discovery phase by gradually selecting process behavior to be incorporated into a process model. Further, most discovery approaches assume process executions, also termed traces, recorded in event data to be complete—complete traces span the actual process from start to completion. Incomplete traces are usually removed in the event data preparation as most discovery algorithms cannot handle them respectively treat them simply as full traces. This paper presents a novel IPD approach that can incorporate process behavior recorded in trace fragments, thus supporting incomplete data. Our experiments show promising results indicating that using trace fragments within IPD leads to high-quality process models.

Keywords: Process mining · Process discovery · Alignments

1 Introduction

Process discovery, i.e., learning process models from event data, is a critical discipline within *process mining*. Discovered models are vital artifacts as they capture the actual execution of processes. Further, many subsequently applied process mining techniques require models as input, for example, generating temporal performance diagnostics and conformance checking statistics [9]. Moreover, these models are used for specifying process-aware information systems [15].

Most process discovery approaches are fully automated [3,22,23]. They take event data as input and return a process model. Apart from parameter settings, users cannot interact with these algorithms despite choosing the input event data and post-processing the discovered model. Moreover, most process discovery algorithms consider the process executions, i.e., *traces*, recorded in the event data to be *complete*—traces are assumed to span the process from start to completion. *Trace fragments* are usually filtered during event data preparation [6,7].

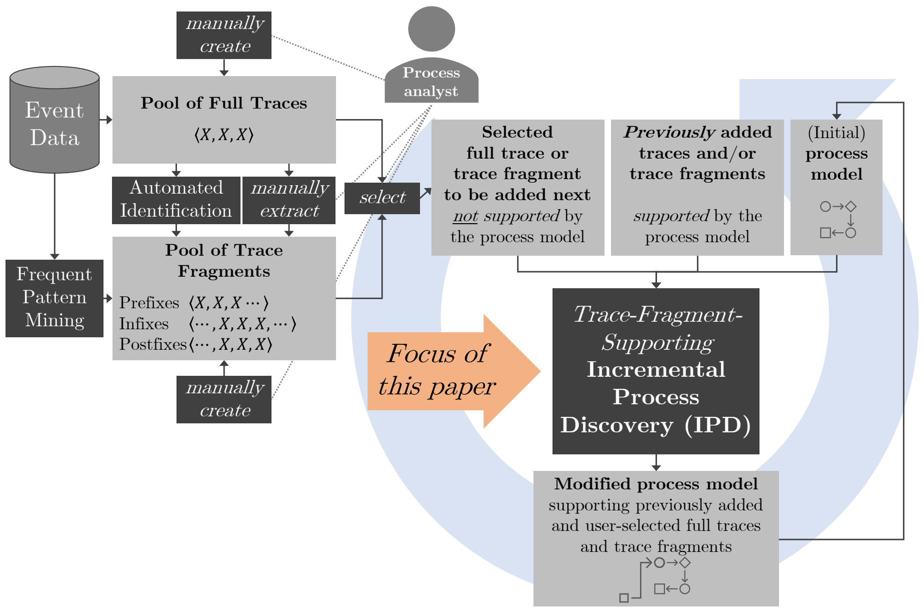

manually create
Process analyst
Event Data
Pool of Full Traces
⟨X,X,X⟩
Automated Identification
manually extract
select
Frequent Pattern Mining
Pool of Trace Fragments
Prefixes ⟨X,X,X ⋯⟩
Infixes ⟨⋯,X,X,X,⋯⟩
Postfixes ⟨⋯,X,X,X⟩
manually create
Selected full trace or trace fragment to be added next not supported by the process model
Previously added traces and/or trace fragments supported by the process model
(Initial) process model
Focus of this paper
Trace-Fragment-Supporting Incremental Process Discovery (IPD)
Modified process model supporting previously added and user-selected full traces and trace fragments

Fig. 1: Overview of Incremental Process Discovery (IPD) with trace fragments and potential origins of trace fragments. Process analysts gradually select process behavior, i.e., full traces and trace fragments, that is added to the process model

In contrast to conventional process discovery, taking event data as input and returning a process model, *domain-knowledge-utilizing process discovery* utilizes additional information besides event data, e.g., user feedback in an interactive discovery phase [10] or explicitly-specified knowledge like precedence constraint among activities [12]. We provide a review of such approaches in [19].

This paper focuses on *Incremental Process Discovery (IPD)* [17], which discovers process models from event data by gradually extending a model by new process behavior, cf. Fig. 1. The central research question addressed is: *How can trace fragments, i.e., trace prefixes/infixes/postfixes, be (incrementally) added to a process model?* We answer this question by proposing a novel IPD approach that allows gradually discovering models from trace fragments and full traces. Thus, the proposed approach utilizes incomplete process behavior, i.e., trace fragments. We evaluate the proposed approach on real-life event data. The results indicate that incorporating trace fragments is beneficial and yields high quality process models. The results further show that a distinction between trace fragments and full traces can lead to better models compared to approaches that do not support fragments respectively consider all traces as full. Finally, we implemented the approach in the open-source process mining tool *Cortado* [20].[3]

Consider Fig. 1; trace fragments may originate from different sources. First, event data itself may contain incomplete respectively partial traces that often occur when event data of a specific time range is extracted from information sys-

[3] https://cortado.fit.fraunhofer.de (from version 1.10.0)

tems. Since trace fragments are usually not labeled as such and are considered complete by state-of-the-art process discovery approaches, event data preparation techniques must be used, e.g., [6], to identify trace fragments that can then be added to the fragment pool, cf. Fig. 1. Second, users can manually extract relevant fragments from full traces. Reasons to proceed in this way are manifold. For instance, an analyst does not want certain variations from full traces that cover specific process stages in the discovered process model; instead, the analyst is only interested in specific fragments. Finally, users can manually create trace fragments during IPD if particular process behavior is not present in the data but should be reflected by the discovered model.

This paper addresses current challenges within business process management (BPM) and process mining. Central research challenges of the BPM discipline are identified in [4]. One challenge is the augmentation "of process mining with common sense and domain knowledge" [4, p. 3]. Domain knowledge about the process under study is often available besides event data; however, process mining techniques often do not utilize such domain knowledge. IPD itself and the proposed trace-fragment-supporting IPD approach allow such exploitation of domain knowledge because 1) users gradually select process behavior (full traces and fragments) to be incorporated into the process model under discovery and 2) can specify how traces from event data are interpreted, i.e., either as full or prefix/infix/postfix traces. By manually creating trace fragments (cf. Fig. 1), another means to incorporate domain knowledge exists. The authors [4] further argue that domain knowledge utilization is beneficial to overcome event data quality issues.

The remainder of this paper is structured as follows. Sect. 2 presents related work, while Sect. 3 introduces preliminaries. We present the proposed trace-fragment-supporting IPD approach in Sect. 4. An initial evaluation of the proposed approach is presented in Sect. 5. Finally, Sect. 6 concludes this paper.

2 Related Work

Plenty of conventional, automated process discovery algorithms exist; we refer to [3,22,23] for overviews. These mainly differ in the model formalism used, guarantees of the discovered model concerning model properties (e.g., soundness), concerning the event data provided (e.g., replay fitness), and computational complexity. From an input/output perspective, however, these algorithms work similarly, i.e., they take event data as input and automatically learn a model.

Domain-knowledge-utilizing process discovery approaches are significantly less common than conventional process discovery. We provide a systematic literature review in [19]. As considered in this work, IPD has been initially proposed in [17]. In [21], the authors also use the term incremental process discovery. However, they create multiple transition systems describing process behavior and then incrementally compose them into a single transition system from which a process model is eventually discovered. Thus, their definition of incremental discovery is unrelated to ours, as illustrated in Fig. 1.

Process model repair [2,11,14] is related to IPD, as elaborated in [19]. Model repair techniques extend process models by non-fitting process behavior. Although repair techniques are not intended to be used incrementally, they can be used similar as illustrated in Fig. 1. However, they create a process model as close as possible to the input process model since the focus is on repairing and not on discovering. This objective is an essential difference from IPD, where this objective does not necessarily exist and is even disadvantageous because the model in the discovery process is constantly being developed and changed.

To the best of our knowledge, no discovery approach, neither conventional nor domain-knowledge-utilizing, and no model repair approach addresses trace fragments explicitly. Thus, even if there is domain knowledge that allows traces to be identified as fragments and labeled as such, no existing approach supports it. However, note that discovery approaches utilizing region theory [5] can handle prefixes. On the contrary, trace fragments—most approaches consider all traces complete—are typically filtered from event logs to obtain better models [6,7]. Thus, supporting trace fragments within IPD is a novelty.

3 Preliminaries

Let X be a set. We denote the universe of multi-sets over X by $\mathcal{B}(X)$ and the set of all sequences over X as X^*, e.g., $[a^3, c] \in \mathcal{B}(\{a, b, c\})$ and $\langle a, b, b\rangle \in \{a, b, c\}^*$. Given two multi-sets $M, M' \in \mathcal{B}(X)$, we denote their union by $M \uplus M'$. We denote the length of a sequence σ by $|\sigma|$. For $1{\leq}i{\leq}|\sigma|$, $\sigma(i)$ represents the i-th element of σ. Given sequences σ and σ', we denote their concatenation by $\sigma{\cdot}\sigma'$, e.g., $\langle a\rangle{\cdot}\langle b, c\rangle = \langle a, b, c\rangle$. We extend the $\cdot$ operator to sets of sequences, i.e., let $S_1, S_2{\subseteq}X^*$ then $S_1{\cdot}S_2 = \{\sigma_1{\cdot}\sigma_2 \mid \sigma_1{\in}S_1{\wedge}\sigma_2{\in}S_2\}$. For sequences σ, σ', the set of all interleaved sequences is denoted by $\sigma{\diamond}\sigma'$, e.g., $\langle a, b\rangle{\diamond}\langle c\rangle = \{\langle a, b, c\rangle, \langle a, c, b\rangle, \langle c, a, b\rangle\}$. We extend the $\diamond$ operator to sets of sequences. Let $S_1, S_2{\subseteq}X^*$, $S_1{\diamond}S_2$ denotes the set of interleaved sequences, i.e., $S_1{\diamond}S_2 = \bigcup_{\sigma_1 \in S_1, \sigma_2 \in S_2} \sigma_1{\diamond}\sigma_2$.

For $\sigma{\in}X^*$ and $X'{\subseteq}X$, we define the projection function $\sigma_{\downarrow_{X'}}{:}X^*{\to}(X')^*$ with: $\langle\rangle_{\downarrow_{X'}}{=}\langle\rangle$, $\left(\langle x\rangle{\cdot}\sigma\right)_{\downarrow_{X'}}{=}\langle x\rangle{\cdot}\sigma_{\downarrow_{X'}}$ if $x{\in}X'$, and $(\langle x\rangle{\cdot}\sigma)_{\downarrow_{X'}}{=}\sigma_{\downarrow_{X'}}$ otherwise.

Let $t{=}(x_1, \dots, x_n) \in X_1{\times} \dots {\times}X_n$ be an n-tuple over n sets. We define projection functions that extract a specific element of t, i.e., $\pi_1(t){=}x_1, \dots, \pi_n(t){=}x_n$, e.g., $\pi_2\left((a, b, c)\right){=}b$. For a sequence $\sigma = \langle(x_1^1, \dots, x_n^1), \dots, (x_1^m, \dots, x_n^m)\rangle \in (X_1{\times} \dots {\times}X_n)^*$ containing n-tuples, we define projection functions $\pi_1^*(\sigma) = \langle x_1^1, \dots, x_1^m\rangle, \dots, \pi_n^*(\sigma) = \langle x_n^1, \dots, x_n^m\rangle$; e.g., $\pi_2^*\left(\langle(a, b), (c, d), (c, b)\rangle\right){=}\langle b, d, b\rangle$.

3.1 Event Data

Event logs are a collection of event data describing the execution of a process. Individual process executions, referred to as *traces*, are considered sequences of executed activities. For instance, $\sigma = \langle a_1, a_2, \dots, a_n\rangle$ is a trace consisting of n activities, and $L = \left[\langle c, b, d\rangle^3, \langle a, e, d\rangle^2\right]$ is an event log that consists of 5 traces.

Definition 1 (Trace & Event Log). *Let $\mathcal{A}$ be the universe of activities. A trace σ is a sequence of activities, i.e., $\sigma \in \mathcal{A}^*$. An event log L is a multi-set of traces, i.e., $L \subseteq \mathcal{B}(\mathcal{A}^*)$.*

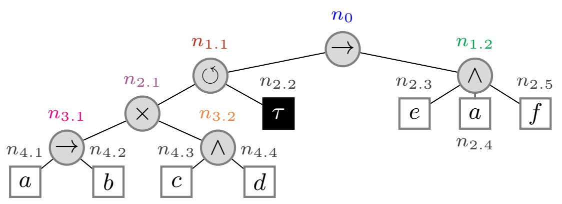


Fig. 2: Exemplary process tree $T_0 = \big(V_0, E_0, \lambda_0, n_0\big)$ with $V_0 = \{n_o, \dots, n_{4.4}\}$, $E_0 = \big\{(n_0, n_{1.1}), \dots, (n_{3.2}, n_{4.4})\big\}$, and $\lambda(n_0) = \rightarrow, \dots, \lambda(n_{4.4}) = d$

3.2 Process Models

We use process trees that represent a subclass of *sound workflow nets* and are an important model formalism used by many discovery approaches [8,13,17]. Process trees are rooted, labeled, ordered trees where leaf nodes represent activities and inner nodes control-flow operators that specify the execution of its subtrees. We distinguish four operators: sequence ($\rightarrow$), parallel ($\wedge$), loop ($\circlearrowleft$), and exclusive choice ($\times$). Fig. 2 depicts an example process tree T_0.

Definition 2 (Process Tree). *Let $\mathcal{A}$ be the universe of activities with $\tau \notin \mathcal{A}$ and $\bigoplus = \{\rightarrow, \times, \wedge, \circlearrowleft\}$ be the process tree operators. We define a process tree as a labeled, rooted tree $T=(V, E, \lambda, r)$ consisting of a totally ordered set of nodes V, edges $E \subseteq V \times V$, a labeling function $\lambda : V \rightarrow \mathcal{A} \cup \{\tau\} \cup \bigoplus$, and a root $r \in V$.*

- *$\big(\{n\}, \emptyset, \lambda, n\big)$ with $\lambda(n) \in \mathcal{A} \cup \{\tau\}$ is a process tree.*
- *Given $k>1$ trees $T_1=(V_1, E_1, \lambda_1, r_1), \dots, T_k=(V_k, E_k, \lambda_k, r_k)$, node $r \notin V_1 \cup \dots \cup V_k$, and $\forall 1 \leq i < j \leq k (V_i \cap V_j = \emptyset)$ then $T=(V, E, \lambda, r)$ is a process tree where:*
  - *$V = V_1 \cup \dots \cup V_k \cup \{r\}$,*
  - *$E = E_1 \cup \dots \cup E_k \cup \big\{(r, r_1), \dots, (r, r_k)\big\}$,*
  - *$\lambda(x) = \lambda_j(x)$ for all $j \in \{1, \dots, k\}, x \in V_j$, and*
  - *$\lambda(r) \in \bigoplus$ and $\lambda(r) = \circlearrowleft \;\Rightarrow\; k = 2$.*

We denote the universe of process trees by $\mathcal{T}$.

For an arbitrary $T=(V, E, \lambda, r) \in \mathcal{T}$, we define function $c^T : V \rightarrow V^*$ that returns the child nodes of a given node sorted accordingly; for instance, $c^{T_0}(n_{1.2}) = \langle n_{2.3}, n_{2.4}, n_{2.5} \rangle$. We refer to the *Lowest Common Ancestor* (LCA) of two nodes $n, n' \in V$ as $lca^T(n, n') \in V$; e.g., $lca^{T_0}(n_{4.4}, n_{2.2}) = n_{1.1}$. The function $\Delta^T(n) \rightarrow \mathcal{T}$ returns the *subtree* rooted at $n \in V$ from T. We write $T' \sqsubseteq T$ to denote that T' is a subtree of T.

We define the semantics of process trees via *running sequences* consisting of 2-tuples where the first entry is a node n

$$
\begin{aligned}
\rho = \Big\langle & (n_0, \mathit{open}),\\
& (n_{1.1}, \mathit{open}),\\
& \quad (n_{2.1}, \mathit{open}),\\
& \quad\quad (n_{3.2}, \mathit{open}),\\
& \quad\quad\quad (n_{4.4}, d), (n_{4.3}, c),\\
& \quad\quad (n_{3.2}, \mathit{close}),\\
& \quad (n_{2.1}, \mathit{close}),\\
& (n_{1.1}, \mathit{close}),\\
& (n_{1.2}, \mathit{open}),\\
& \quad (n_{2.3}, e), (n_{2.5}, f), (n_{2.4}, a),\\
& (n_{1.2}, \mathit{close}),\\
& (n_0, \mathit{close})\Big\rangle
\end{aligned}
$$

Fig. 3: Running sequence $\rho \in \mathcal{RS}(T_0)$

and the second entry is either the label if n is a leaf node or a label indicating the opening or closing of the subtree rooted at n. Fig. 3 shows one running sequence ρ of process tree T_0. Note that ρ corresponds to the trace $\big(\pi_2^*(\rho)\big)_{\downarrow\mathcal{A}} = \langle d, c, e, f, a\rangle$.

Definition 3 (Running Sequence). *Let $\mathcal{A}$ be the universe of activities with $\tau, open, close \notin \mathcal{A}$. Let $S = V\times(\mathcal{A}\cup\{\tau, open, close\})$ be the set of steps. For $T{=}(V,E,\lambda,r)\in\mathcal{T}$, we recursively define its running sequences $\mathcal{RS}(T)\subseteq S^*$.*

- *if $\lambda(r)\in\mathcal{A}\cup\{\tau\}$ (T is a leaf node): $\mathcal{RS}(T) = \big\{\big\langle(r,\lambda(r))\big\rangle\big\}$*
- *if $\lambda(r)={\rightarrow}$ with child nodes $c^T(r){=}\langle v_1,\dots,v_k\rangle$ for $k{\geq}1$:* $\mathcal{RS}(T) = \big\{\big\langle(r, open)\big\rangle\big\}{\cdot}\mathcal{RS}(\triangle^T(v_1)){\cdot}\dots{\cdot}\mathcal{RS}(\triangle^T(v_k)){\cdot}\big\{\big\langle(r, close)\big\rangle\big\}$
- *if $\lambda(r)=\times$ with child nodes $c^T(r){=}\langle v_1,\dots,v_k\rangle$ for $k{\geq}1$:* $\mathcal{RS}(T) = \big\{\big\langle(r, open)\big\rangle\big\}\cdot\mathcal{RS}(\triangle^T(v_1)){\cup}\dots{\cup}\mathcal{RS}(\triangle^T(v_k))\cdot\big\{\big\langle(r, close)\big\rangle\big\}$
- *if $\lambda(r){=}\wedge$ with child nodes $c^T(r){=}\langle v_1,\dots,v_k\rangle$ for $k{\geq}1$:* $\mathcal{RS}(T) = \big\{\big\langle(r, open)\big\rangle\big\}\cdot\mathcal{RS}(\triangle^T(v_1)){\diamond}\dots{\diamond}\mathcal{RS}(\triangle^T(v_k))\cdot\big\{\big\langle(r, close)\big\rangle\big\}$
- *if $\lambda(r)=\circlearrowleft$ with child nodes $c^T(r){=}\langle v_1,v_2\rangle$:* $\mathcal{RS}(T) = \Big\{\big\langle(r, open)\big\rangle{\cdot}\sigma_1{\cdot}\sigma_1'{\cdot}\sigma_2{\cdot}\sigma_2'{\cdot}\dots{\cdot}\sigma_m{\cdot}\big\langle(r, close)\big\rangle \mid m{\geq}1\wedge \forall 1{\leq}i{\leq}m\big(\sigma_i{\in}\mathcal{RS}(\triangle^T(v_1))\big)\wedge\forall 1{\leq}i{<}m\big(\sigma_i'{\in}\mathcal{RS}(\triangle^T(v_2))\big)\Big\}$

The *language* of a tree $T\in\mathcal{T}$ is a set of supported traces, i.e., $\mathcal{L}(T) = \big\{\pi_2^*(\sigma)_{\downarrow\mathcal{A}} \mid \sigma{\in}\mathcal{RS}(T)\big\}{\subseteq}\mathcal{A}^*$. Further, we define its prefix/infix/postfix language.

- $\mathcal{L}_{prefix}(T) = \{\sigma_1 \mid \sigma_1,\sigma_2\in\mathcal{A}^* \;\wedge\; \sigma_1\cdot\sigma_2\in\mathcal{L}(T)\}$
- $\mathcal{L}_{infix}(T) = \{\sigma_2 \mid \sigma_1,\sigma_2,\sigma_3\in\mathcal{A}^* \;\wedge\; \sigma_1\cdot\sigma_2\cdot\sigma_3\in\mathcal{L}(T)\}$
- $\mathcal{L}_{postfix}(T) = \{\sigma_2 \mid \sigma_1,\sigma_2\in\mathcal{A}^* \;\wedge\; \sigma_1\cdot\sigma_2\in\mathcal{L}(T)\}$

Finally, we formally introduce fitness-preserving process discovery.

Definition 4 (Fitness-Preserving Process Discovery). *Let $L\subseteq\mathcal{B}(\mathcal{A}^*)$ be an event log. We define a fitness-preserving, automated process discovery algorithm as a function $disc:\mathcal{B}(\mathcal{A}^*)\to\mathcal{T}$ such that $L\subseteq\mathcal{L}\big(disc(L)\big)$.*

3.3 Alignments

Alignments [1] are a state-of-the-art conformance-checking technique [9] to compare process models with traces. Full alignments, often referred to as alignments, and prefix alignments have been introduced in [1]. In [16], we define infix and postfix alignments and describe their computation. Fig. 4 shows an exemplary full, prefix, infix, and postfix alignment for different trace (fragments) and tree T_0. In general, the first row, i.e., the *trace part*, of any alignment corresponds to the trace (fragment) when ignoring the skip symbol $\gg$. Analogously, the second row, i.e., the *model part*, corresponds to a running sequence (fragment) of the tree when ignoring $\gg$. Each column represents an alignment move; we generally distinguish five types: synchronous moves, log moves, visible model moves, invisible model moves, and opening & closing model moves.

$\gg$	$\gg$	$\gg$	$\gg$	c	d	$\gg$	$\gg$	$\gg$	$\gg$	$\gg$	a	f	$\gg$	$\gg$
$(n_0, open)$	$(n_{1.1}, open)$	$(n_{2.1}, open)$	$(n_{3.2}, open)$	$(n_{4.3}, c)$	$(n_{4.4}, d)$	$(n_{3.2}, close)$	$(n_{2.1}, close)$	$(n_{1.1}, close)$	$(n_{1.2}, open)$	$(n_{2.3}, e)$	$(n_{2.4}, a)$	$(n_{2.5}, f)$	$(n_{1.2}, close)$	$(n_0, close)$

(a) Optimal *full alignment* γ_1 for the full trace $\langle c, d, a, f \rangle$ and T_0

$\gg$	$\gg$	$\gg$	$\gg$	a	b	$\gg$	$\gg$	$\gg$	$\gg$	$\gg$	d
$(n_0, open)$	$(n_{1.1}, open)$	$(n_{2.1}, open)$	$(n_{3.1}, open)$	$(n_{4.1}, a)$	$(n_{4.2}, b)$	$(n_{3.1}, close)$	$(n_{2.1}, close)$	$(n_{2.2}, \tau)$	$(n_{2.1}, open)$	$(n_{3.2}, open)$	$(n_{4.4}, d)$

(b) Optimal *prefix alignment* γ_2 for the trace prefix $\langle a, b, d \rangle$ and T_0

d	f	$\gg$	$\gg$	$\gg$	$\gg$	$\gg$	d
$(n_{4.4}, d)$	$\gg$	$(n_{3.2}, close)$	$(n_{2.1}, close)$	$(n_{2.2}, \tau)$	$(n_{2.1}, open)$	$(n_{3.2}, open)$	$(n_{4.4}, d)$

(c) Optimal *infix alignment* γ_3 for the trace infix $\langle d, f, d \rangle$ and T_0

e	e	f	$\gg$	$\gg$
$\gg$	$(n_{2.3}, e)$	$(n_{2.5}, f)$	$(n_{1.2}, close)$	$(n_0, close)$

(d) Optimal *postfix alignment* γ_4 for the trace postfix $\langle e, e, f \rangle$ and T_0

Fig. 4: Examples of optimal alignments for process tree T_0 (cf. Fig. 2)

Definition 5 (Full/Prefix/Infix/Postfix Alignment). *Let $\mathcal{A}$ be the universe of activity labels, $\gg \notin \mathcal{A} \cup \{\tau\}$, $\sigma \in \mathcal{A}^*$ be a trace (fragment), $T=(V, E, \lambda, r) \in \mathcal{T}$, and $S = V \times (\mathcal{A} \cup \{\tau, open, close\})$ be the set of running sequence steps. Sequence $\gamma \in \big((\mathcal{A} \cup \{\gg\}) \times (S \cup \{\gg\})\big)^*$ is a full/prefix/infix/postfix alignment if:*

1. $\sigma = \pi_1^*(\gamma)_{\downarrow_{\mathcal{A}}}$,
2. – ***Full alignment:*** $\pi_2^*(\gamma)_{\downarrow_S} \in \mathcal{RS}(T)$,
   – ***Prefix alignment:*** $\exists \rho \in S^* \big(\pi_2^*(\gamma)_{\downarrow_S} \cdot \rho \in \mathcal{RS}(T)\big)$,
   – ***Infix alignment:*** $\exists \rho_1, \rho_2 \in S^* \big(\rho_1 \cdot \pi_2^*(\gamma)_{\downarrow_S} \cdot \rho_2 \in \mathcal{RS}(T)\big)$,
   – ***Postfix alignment:*** $\exists \rho \in S^* \big(\rho \cdot \pi_2^*(\gamma)_{\downarrow_S} \in \mathcal{RS}(T)\big)$,
3. $(\gg, \gg) \notin \gamma$,
4. $\forall 1 \leq i \leq |\gamma| \; \big(\pi_1(\gamma(i)) \in \mathcal{A} \wedge \pi_2(\pi_2(\gamma(i))) \in \mathcal{A} \Rightarrow \pi_1(\gamma(i)) = \pi_2(\pi_2(\gamma(i)))\big)$, *and*
5. $\forall 1 \leq i \leq |\gamma| \; \big(\pi_2(\pi_2(\gamma(i))) \in \{open, close\} \Rightarrow \pi_1(\gamma(i)) = \gg \big)$.

We denote the universe of full/prefix/infix/postfix alignments for T and σ by $\Gamma_{full}(T, \sigma)$, $\Gamma_{prefix}(T, \sigma)$, $\Gamma_{infix}(T, \sigma)$, and $\Gamma_{postfix}(T, \sigma)$. We denote the universe of alignments as $\Gamma(T, \sigma) = \Gamma_{full}(T, \sigma) \cup \Gamma_{prefix}(T, \sigma) \cup \Gamma_{infix}(T, \sigma) \cup \Gamma_{postfix}(T, \sigma)$.

Consider Fig. 4d, $\gamma_4 = \big\langle (e, \gg), (e, (n_{2.3}, e)), (f, (n_{2.5}, f)), (\gg, (n_{1.2}, close)), (\gg, (n_0, close)) \big\rangle$ is a postfix alignment. Let $\gamma \in \Gamma(T, \sigma)$ and $\gamma(i)$ for $1 \leq i \leq |\gamma|$ be an alignment move. We introduce abbreviations for ease of reading.

– $traceLabel\big(\gamma(i)\big) = \pi_1(\gamma(i))$
– $modelNode\big(\gamma(i)\big) = \begin{cases} \pi_1\big(\pi_2(\gamma(i))\big) & \text{if } \pi_2\big(\gamma(i)\big) \in S \\ \gg & \text{otherwise} \end{cases}$
– $modelLabel\big(\gamma(i)\big) = \begin{cases} \pi_2\big(\pi_2(\gamma(i))\big) & \text{if } \pi_2\big(\gamma(i)\big) \in S \\ \gg & \text{otherwise} \end{cases}$

For example, consider postfix alignment γ_4 shown in Fig. 4d: $traceLabel\big(\gamma_4(2)\big) = e$, $modelNode\big(\gamma_4(2)\big) = n_{2.3}$, and $modelLabel\big(\gamma_4(2)\big) = e$.

For an alignment $\gamma \in \Gamma(T, \sigma)$, we say that the alignment move $\gamma(i)$ for $1 \leq i \leq |\gamma|$ *indicates a deviation* if it is a log move, i.e., $traceLabel(\gamma(i)) = \gg$, or a visible model move, i.e., $traceLabel(\gamma(i)) = \gg \; \wedge \; modelLabel(\gamma(i)) \in \mathcal{A}$.

Let $\square \in \{\mathit{full}, \mathit{prefix}, \mathit{infix}, \mathit{postfix}\}$, $T \in \mathcal{T}$, and $\sigma \in \mathcal{A}^*$. Since many alignments exist for a given trace (fragment) and a process tree, the concept of *optimality* exists. An alignment is optimal if the number of visible model moves and log moves is minimal. We define four functions $\mathit{align}_\square : \mathcal{T} \times \mathcal{A}^* \rightarrow \Gamma_\square(T, \sigma)$ that return a $\square$ alignment for $\sigma \in \mathcal{A}^*$ and $T \in \mathcal{T}$. We write $\mathit{align}_\square^{opt}$ to indicate that we compute an *optimal* $\square$ alignment.

4 Trace-Fragment-Supporting IPD

This section describes the proposed trace-fragment-supporting IPD approach (cf. Fig. 1) that builds on the IPD approach presented in [17], which only features full traces. The basic idea, however, remains. When a full trace/trace fragment is added that is not contained in the language of the current process tree, the proposed approach determines relevant subtrees in the given process tree causing the deviation and rediscovers these deviating subtrees such that previous added traces/trace fragments and additionally the given trace (fragment) are supported. The remainder of this section presents the algorithm in detail. Sect. 4.1 presents the core part of the algorithm and introduces a running example. Next, Sect. 4.2 describes how deviating subtrees are identified, and Sect. 4.3 introduces the corresponding sub-log calculation needed for rediscovery.

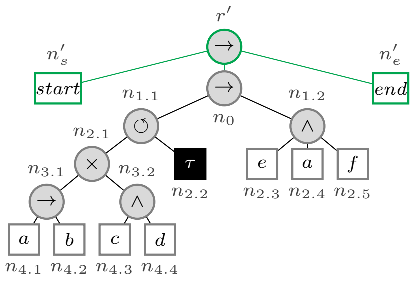
4.1 Overview

This section provides an overview of the proposed approach. Further, we introduce a running example demonstrating various critical steps of the approach, cf. Fig. 5. Below, we list the required four *inputs*.

1. trace (fragment) $\sigma_{next} \in \mathcal{A}^*$ to be added next to the current process tree
2. interpretation $\square \in \{\mathit{full}, \mathit{prefix}, \mathit{infix}, \mathit{postfix}\}$ of trace (fragment) σ_{next}
3. *previously added* traces and trace fragments, divided into full traces $L_{full} \subseteq \mathcal{A}^*$, prefixes $L_{prefix} \subseteq \mathcal{A}^*$, infixes $L_{infix} \subseteq \mathcal{A}^*$, and postfixes $L_{postfix} \subseteq \mathcal{A}^*$
4. process tree $T \in \mathcal{T}$, supporting previously added traces/trace fragments, i.e., $L_{full} \subseteq \mathcal{L}(T)$, $L_{prefix} \subseteq \mathcal{L}_{prefix}(T)$, $L_{infix} \subseteq \mathcal{L}_{infix}(T)$, and $L_{postfix} \subseteq \mathcal{L}_{postfix}(T)$

Note that the proposed approach requires an initial process tree as input to start the incremental process discovery in the very first iteration. For example, this initial process tree can consist of only a single (invisible) activity. Alternatively, users can manually model an initial tree or use a conventional process discovery algorithm to discover one.

The *output* of the trace-fragment-supporting IPD algorithm is a process tree T that, in addition to the previously added traces and trace fragments, contains σ_{next} in its language—depending on σ_{next}'s interpretation, $\sigma_{next} \in \mathcal{L}_\square(T)$. Subsequently, we introduce the overall algorithm, presented in Alg. 1, and exemplify critical steps with the running example shown in Fig. 5.



(a) Input process tree T, i.e., tree T_0 extended by a *start* and *end* activity

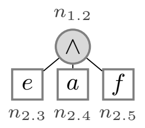
$\sigma_{next} = \langle a, a, f, end\rangle$
$\square = postfix$
$L_{full} = [\sigma_1 = \langle start, a, b, a, b, f, e, a, end\rangle]$
$L_{prefix} = [\sigma_2 = \langle start, d, c, e, a\rangle]$
$L_{infix} = [\sigma_3 = \langle c, f, a\rangle, \sigma_4 = \langle b, d, c, e, a\rangle]$
$L_{postfix} = [\sigma_5 = \langle f, a, end\rangle]$

(b) Input trace (fragment) σ_{next} to be added to T, its interpretation $\square$, previously added full traces, and trace fragments

1	2	3	4	5	6	7
a	a	f	$\gg$	$\gg$	end	$\gg$
$(n_{2.4}, a)$	$\gg$	$(n_{2.5}, f)$	$(n_{1.2}, close)$	$(n_0, close)$	(n'_e, end)	$(r', close$

(c) Optimal postfix alignment for σ_{next} and T—the 2. alignment move indicates a deviation (Alg. 1 line 2)

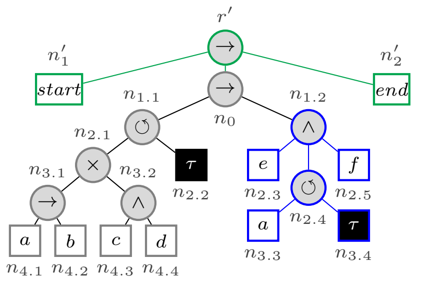
$lca^T(n_{2.4}, n_{2.5}) = n_{1.2}$
$T_{LCA} = \triangle^{T'}(n_{1.2})$



(d) Problematic subtree $T_{LCA} \sqsubseteq T$ (Alg. 1 line 3)

$L_{LCA} = [$	$\langle e, a, a, f\rangle$	*derived from* σ_{next}
	$\langle f, e, a\rangle$	*derived from* σ_1
	$\langle e, a, f\rangle^2$	*derived from* σ_2, σ_4
	$\langle f, a, e\rangle$	*derived from* σ_3
	$\langle e, f, a\rangle]$	*derived from* σ_5

(e) Sub-log L_{LCA} for T_{LCA} (Alg. 1 line 4) that represents trace fragments T_{LCA} must support to avoid the deviation indicated in the postfix alignment shown in Fig. 5c



(f) Replacing $T_{LCA} \sqsubseteq T$ by $disc(L_{LCA}) \in \mathcal{T}$ (Alg. 1 line 5)

Fig. 5: Running example of a full execution of the proposed IPD approach

Input Preparation Alg. 1 requires that the input artifacts, the tree T and the traces/trace fragments as described above, are extended by an artificial start and end activity. These artificial activities are needed to ensure the correct integration of trace infixes and postfixes into T.

1. Process tree T is extended by *start*, *end* $\notin \mathcal{A}$ activities, e.g., cf. Fig. 5a.
2. Trace (fragment) to be added next σ_{next} and previously added traces/trace fragments are correspondingly extended by *start* and *end* activities to match the extended process tree T, for example, consider Fig. 5b.

Extending Process Tree $\boldsymbol{T}$ This section describes Alg. 1. First, we calculate an optimal full/prefix/infix/postfix alignment γ according to σ_{next}'s interpre-

Algorithm 1: *TraceFragmentSupportingIPD*

```
input  : $T{=}(V,E,\lambda,r)\in\mathcal{T}$, // process tree to be extended
$\square\in\{full, prefix, infix, postfix\}$, $\sigma_{next}\in\mathcal{A}^*$, // $\square$ trace $\sigma_{next}$ to be added to $\mathcal{L}_\square(T)$
$L_{full}, L_{prefix}, L_{infix}, L_{postfix}\subseteq\mathcal{B}(\mathcal{A}^*)$, // previously added full traces/trace fragments
output: $T\in\mathcal{T}$ // $\sigma_{next}\in\mathcal{L}_\square(T)$, $L_{full}\subseteq\mathcal{L}(T)$, $L_{prefix}\subseteq\mathcal{L}_{prefix}(T)$,
    $L_{infix}{\subseteq}\mathcal{L}_{infix}(T)$, $L_{postfix}{\subseteq}\mathcal{L}_{postfix}(T)$
begin
1   $L_\square\leftarrow L_\square\uplus[\sigma_{next}]$;              // add $\sigma_{next}$ to the corresponding log $L_{next}$
2   while $\gamma\leftarrow align^{opt}_\square(T,\sigma_{next})$ indicates a deviation do        // $\sigma_{next}\notin\mathcal{L}_\square(T)$
3       $T_{LCA}\leftarrow DetermineSubtree(T,\gamma)$ ;                                  // Alg. 2
        if $T_{LCA}$ then
4           $L_{LCA}\leftarrow SubLog(T, T_{LCA}, L_{full}, L_{prefix}, L_{infix}, L_{postfix})$;   // Alg. 3
5           $T\leftarrow$ replace $T_{LCA}\sqsubseteq T$ by $disc(L_{LCA}){\in}\mathcal{T}$;
        else                        // no subtree causing the deviation could be determined
6           $T\leftarrow$ extend $T$ according to Fig. 6;
7   return $T$;
```

tation. In case γ does not indicate a deviation, we know that $\sigma_{next}\in\mathcal{L}_\square(T)$ and return. Otherwise, we call Alg. 2 in line 3 that determines the subtree $T_{LCA}\sqsubseteq T$ that causes the first cohesive block of deviations, as indicated in γ. Hereinafter, assume that T_{LCA} exists. Next, we calculate sub-log L_{LCA} for the determined subtree T_{LCA}, cf. line 4. The sub-log corresponds to all sub-traces that T_{LCA} must be able to replay, i.e., $L_{LCA}\not\subseteq\mathcal{L}(T_{LCA})$. Sub-log L_{LCA} is therefore calculated based on previous added full/prefix/infix/postfix traces and the trace (fragment) to be added next, i.e., σ_{next}. Next, we replace $T_{LCA}\sqsubseteq T$ by a new subtree $disc(L_{LCA})\in\mathcal{T}$ that fully supports the computed sub-log, i.e., $L_{LCA}\subseteq\mathcal{L}(disc(L_{LCA}))$ (line 5). Again, we compute alignment γ for the modified tree T and σ_{next} (line 2). If γ still indicates deviations, we repeat the procedure described above until all deviations are resolved. Note that the termination of Alg. 1 is guaranteed since in each iteration of the `while` block (line 2–6), the first contiguous block of deviations is resolved. Thus, eventually $\sigma_{next}\in\mathcal{L}_\square(T)$.

Consider the running example, cf. Fig. 5. The postfix alignment (cf. Fig. 5c) indicates a deviation—the second move is a log move, i.e., activity a cannot be replayed twice in the model. Next, we compute the subtree T_{LCA} that causes the deviation, cf. Fig. 5d. T_{LCA} supports the postfix $\langle a,f\rangle$ but not the postfix $\langle a,a,f\rangle$. Since we want to replace respectively rediscover T_{LCA}, we calculate a corresponding sub-log L_{LCA}, cf. Fig. 5e. The actual computation is explained in a subsequent section. However, note that the calculated sub-log L_{LCA} contains $\langle e,a,a,f\rangle$. Thus, when discovering a tree from L_{LCA} using a fitness-preserving discovery algorithm (cf. Def. 4), the discovered tree supports the execution of two subsequent a activities.

So far, we assumed that T_{LCA} causing the deviation(s) as indicated in γ could be determined. However, one case exists in which T_{LCA} cannot be determined, i.e., σ_{next} is an infix, and T does not contain any of its activities. Hence, γ includes only log moves. Thus, we do not have any reference point in the tree where the infix should happen. In this case, we extend tree T as depicted in Fig. 6, i.e., we discover a subtree $disc([\sigma_{next}])$, make it optional, and add it in parallel to T (cf. line 6). This extension guarantees that $\sigma_{next}\in\mathcal{L}_\square(T)$. This

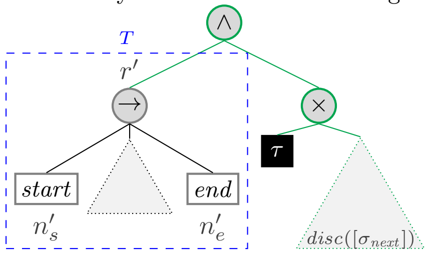


Fig. 6: Extending tree T by an optional parallel subtree supporting σ_{next}

Algorithm 2: *DetermineSubtree* (called in Alg. 1 line 3)

```
input  : $T \in \mathcal{T},\ \gamma \in \Gamma(T, \sigma_{next})$ // alignment for trace (fragment) $\sigma_{next}$ and $T$
output: $T_{LCA} \sqsubseteq T$ // subtree that is responsible for the first deviation (block)
begin
    forall $1 \leq i \leq |\gamma|$ do
        if $\gamma(i)$ indicates a deviation then
            $i_{before} \leftarrow$ closest move $\gamma(i_{before})$ before $\gamma(i)$ that is a synchronous move or an
              invisible model move (if possible, otherwise null);
            $i_{after} \leftarrow$ closest move $\gamma(i_{after})$ after $\gamma(i)$ that is a synchronous move or an
              invisible model move (if possible, otherwise null);
            if $i_{before} \wedge i_{after}$ then                    // both corresponding moves exist
                return $\Delta^T(lca^T(modelNode(\gamma(i_{before})), modelNode(\gamma(i_{after}))))$;
            else if $i_{before}$ then                 // only a corresponding move before exists
                return $\Delta^T(modelNode(i_{before}))$;
            else if $i_{after}$ then                   // only a corresponding move after exists
                return $\Delta^T(modelNode(i_{after}))$;
            else
                return null;
```

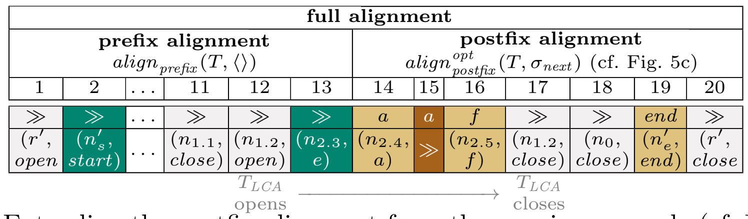
described procedure is however usually very The next sub-sections introduce the algorithms *DetermineSubtree* and *SubLog* called in Alg. 1 (line 3 and 4).

4.2 Subtree Detection

Alg. 2 describes the subtree detection *DetermineSubtree* of T_{LCA}. As input, Alg. 1 provides tree T and alignment $\gamma \in \Gamma_{\square}(T, \sigma_{next})$ indicating a deviation. The central idea is to find the first deviation (block) in γ and the closest alignment moves that: surround the found deviation (block), do not indicate a deviation, and correspond to an executed leaf node of T, cf. line 3 and 4. If such two moves can be found, we compute an LCA from the corresponding leaf nodes of these moves. We know that the subtree rooted at the computed LCA is causing the deviation (block), and hence, we return it (line 6). If we can only find one of the two moves, we return the subtree rooted at the corresponding node—this subtree consists of only a leaf node and indicates that around this leaf node, a deviation occurs regarding σ_{next}. In the particular case that no surrounding move can be found, we return nothing, cf. line 12. Note that this case can only happen if we have an infix alignment containing only log moves—all other alignments have at least a synchronous move on the initially added *start* or *end* activity (cf. Fig. 5a).

Algorithm 3: *SubLog* (called in Alg. 1 line 4)

```
    input  : $T{\in}\mathcal{T},\ T_{LCA}{\sqsubseteq}T,\ L_{full}, L_{prefix}, L_{infix}, L_{postfix}{\subseteq}\mathcal{B}(\mathcal{A}^*)$
    output: $L_{LCA} \subseteq \mathcal{B}(\mathcal{A}^*)$ // sub-log for $T_{LCA}$
    begin
1       $L_{LCA} \leftarrow []$;                                                  // initialize sub-log for $T_{LCA}$
2       forall $\sigma \in L_{full}$ do
3           $\gamma \leftarrow align^{opt}_{full}(T,\sigma)$;
4           $L_{LCA} \leftarrow L_{LCA} \uplus ExtractSubTraces(T_{LCA}, \gamma, \{1,\dots,|\gamma|\})$;     // Alg. 4
5       forall $\sigma \in L_{prefix}$ do
6           $\gamma \leftarrow align^{opt}_{prefix}(T,\sigma) \cdot align_{postfix}(T,\langle\rangle)$ such that $\gamma{\in}\Gamma_{full}(T,\sigma)$;
7           $I \leftarrow \{1,\dots,i\}$ such that $\langle\gamma(1),\dots,\gamma(i)\rangle = align^{opt}_{prefix}(T,\sigma)$;
8           $L_{LCA} \leftarrow L_{LCA} \uplus ExtractSubTraces(T_{LCA}, \gamma, I)$;                  // Alg. 4
9       forall $\sigma \in L_{infix}$ do
10          $\gamma \leftarrow align_{prefix}(T,\langle\rangle) \cdot align^{opt}_{infix}(T,\sigma) \cdot align_{postfix}(T,\langle\rangle)$ such that $\gamma{\in}\Gamma_{full}(T,\sigma)$;
11          $I \leftarrow \{i,\dots,i{+}n\}$ such that $\langle\gamma(i),\dots,\gamma(i{+}n)\rangle = align^{opt}_{infix}(T,\sigma)$;
12          $L_{LCA} \leftarrow L_{LCA} \uplus ExtractSubTraces(T_{LCA}, \gamma, I)$;                  // Alg. 4
13      forall $\sigma \in L_{postfix}$ do
14          $\gamma \leftarrow align_{prefix}(T,\langle\rangle) \cdot align^{opt}_{postfix}(T,\sigma)$ such that $\gamma{\in}\Gamma_{full}(T,\sigma)$;
15          $I \leftarrow \{i,\dots,|\gamma|\}$ such that $\langle\gamma(i),\dots,\gamma(|\gamma|)\rangle = align^{opt}_{postfix}(T,\sigma)$;
16          $L_{LCA} \leftarrow L_{LCA} \uplus ExtractSubTraces(T_{LCA}, \gamma, I)$;                  // Alg. 4
17      return $L_{LCA}$;
```



full alignment												
prefix alignment $align_{prefix}(T,\langle\rangle)$						postfix alignment $align^{opt}_{postfix}(T,\sigma_{next})$ (cf. Fig. 5c)						
1	2	...	11	12	13	14	15	16	17	18	19	20
$\gg$	$\gg$	...	$\gg$	$\gg$	$\gg$	a	a	f	$\gg$	$\gg$	end	$\gg$
$(r', open)$	$(n'_s, start)$	...	$(n_{1.1}, close)$	$(n_{1.2}, open)$	$(n_{2.3}, e)$	$(n_{2.4}, a)$	$\gg$	$(n_{2.5}, f)$	$(n_{1.2}, close)$	$(n_0, close)$	(n'_e, end)	$(r', close)$

T_{LCA} opens
T_{LCA} closes

Fig. 7: Extending the postfix alignment from the running example (cf. Fig. 5c)

Consider alignment γ from the running example (cf. Fig. 5c). Its first and only deviation is at the second move, surrounded by two synchronous moves representing the execution of node $n_{2.4}$ and $n_{2.5}$. Thus, we compute $LCA^T(n_{2.4}, n_{2.5}) = n_{1.2}$, and return subtree T_{LCA} rooted at $n_{1.2}$ (cf. Fig. 5d) because this subtree does not support executing two a activities, as indicated by γ.

4.3 Sub-log Calculation for Detected Subtree

Alg. 3 describes the sub-log calculation *SubLog* called in Alg. 1 line 4 for the determined subtree T_{LCA}. The output of the sub-log calculation is an event log L_{LCA} that the determined subtree T_{LCA} must support. To this end, all traces and trace fragments including σ_{next} are aligned with $T \sqsupseteq T_{LCA}$ to identify the corresponding sub-traces that T_{LCA} must support.

For example, consider postfix alignment γ (cf. Fig. 5c) and T_{LCA} with root node $n_{1.2}$ (cf. Fig. 5d). Adding $\sigma_{next}{=}\langle a, a, f\rangle$ to sub-log L_{LCA} would result in an unnecessary imprecise subtree because when replacing T_{LCA} by a rediscovered tree from L_{LCA}, activity e would be optional. However, no previously added trace (fragment) nor σ_{next} requires activity e being optional. Thus, we extend postfix alignment γ to a full one such that T_{LCA} is fully executed within the

Algorithm 4: *ExtractSubTraces* (called in Alg. 3)

```
input  : $T_{LCA}{=}(V_{LCA}, E_{LCA}, \lambda_{LCA}, r_{LCA}) \sqsubseteq T,\ \gamma{\in}\Gamma_{full}(T,\sigma),\ I \subseteq \{1,\dots,|\gamma|\}$
output: $L \subseteq \mathcal{B}(\mathcal{A}^*)$ // sub-log for $T_{LCA}$
begin
1   $L = [\,]$;                                                   // initialize sub-log for $T_{LCA}$
    forall $1{\le}i{\le}|\gamma|$ do                               // iterate over alignment moves
2       $\sigma' \leftarrow \langle\rangle$;
3       if $V_{LCA} = \{r_{LCA}\}$ then                            // $T_{LCA}$ is leaf node
4           while $modelNode(\gamma(i)) \neq r_{LCA}$ do
5               if $\gamma(i)$ is log move then
6                   $\sigma' \leftarrow \sigma' \cdot \langle traceLabel(\gamma(i))\rangle$;   // add log moves
7               $i \leftarrow i{+}1$;
            if $modelNode(\gamma(i)) = r_{LCA}$ then               // $r_{LCA}$ is executed
8               $\sigma' \leftarrow \sigma' \cdot \langle modelLabel(\gamma(i))\rangle$;   // $modelLabel(\gamma(i)) = \lambda_{LCA}(r_{LCA})$
9               if $\forall i{<}j{\le}|\gamma| \big(\gamma(j)$ is neither a sync. nor an invisible model move$\big)$ then
10                  $\sigma' \leftarrow \sigma' \cdot \big\langle traceLabel(\gamma(j)), \dots, traceLabel(\gamma(|\gamma|))\big\rangle_{\downarrow\mathcal{A}}$;
11              $L \leftarrow L \uplus [\sigma']$;
12      else                                                       // $T_{LCA}$ is a subtree with more than one node
13          if $modelNode(\gamma(i)) = r_{LCA}\ \wedge\ modelLabel(\gamma(i)) = open$ then
14              while $modelNode(\gamma(i)) \neq r_{LCA}\ \vee\ modelLabel(\gamma(i)) \neq close$ do
                    // consider all subsequent moves until $r_{LCA}$ is closed
15                  if $modelNode(\gamma(i)){\in}V_{LCA}\ \wedge \big[\gamma(i)$ is synchronous move $\vee$
                        $\big(\gamma(i)$ is visible model move $\wedge\, i{\notin}I\big)\big]$ then
16                      $\sigma' \leftarrow \sigma' \cdot \langle modelLabel(\gamma(i))\rangle$;
                    else if $traceLabel(\gamma(i)) \in \mathcal{A}$ then
17                      $\sigma' \leftarrow \sigma' \cdot \langle traceLabel(\gamma(i))\rangle$;
18                  $i \leftarrow i{+}1$;
19              $L \leftarrow L \uplus [\sigma']$;
20  return $L$;
```

model part. Fig. 7 exemplifies such an extension of γ. For each full execution of T_{LCA}, we generate a sub-trace. T_{LCA} is opened in move 12, and closed in move 17. All moves in between that represent the execution of a leaf node (i.e., move 13, 14, and 16) are contained in T_{LCA}. Thus, we add the sub-trace $\langle e, a, a, f\rangle$ to L_{LCA}. We proceed similarly for previously added traces and trace fragments.

Alg. 3 provides the sub-log calculation. For full traces, we calculate a full alignment (line 4) and extract the corresponding sub-traces. For trace fragments, we compute a corresponding prefix/infix/postfix alignment and expand this into a full alignment, as exemplified in Fig. 7. Extending to full alignments is required as T_{LCA} might span larger parts of the process and hence might be only partially executed within the prefix/infix/postfix alignment. For instance, consider Fig. 7. The depicted postfix alignment does not contain a full execution of T_{LCA}.

Alg. 4 defines the extraction of sub-trace(s) for T_{LCA} from full alignments. If T_{LCA} is a leaf node (line 3), we add all log moves until T_{LCA} is executed. If afterwards T_{LCA} is never executed again, potential log moves after the last execution of T_{LCA} are also added to the sub-trace σ'. Thus, per execution of T_{LCA} one sub-trace is added to L. Note that log moves only occur for the trace to be added, for all other previously added traces/trace fragments log moves do not occur in γ. If T_{LCA} is a leaf node (line 12), we search for the opening of

T_{LCA} (line 13). All activities from visible model/synchronous moves that belong to T_{LCA} (line 16) and log moves (line 17) are added to σ' until T_{LCA} is closed.

5 Evaluation

We present an initial evaluation of the proposed trace-fragment-supporting IPD approach. The central goal of the evaluation is to showcase that distinguishing trace fragments from full traces within IPD leads to comparable or even better process models than classic IPD [17], considering all traces as full ones.

5.1 Experimental Setup

We compare trace-fragment-supporting IPD (TFS-IPD) with IPD [17] and automated conventional process discovery algorithms: Inductive Miner (IM) [13], IM infrequent (IMf) [13], and evolutionary tree miner (ETM) [8]. All listed approaches discover process trees. We use publicly available real-life event logs.[4] Note that event logs generally consider all traces recorded as full traces. Thus, to obtain trace fragments, we proceed as follows.

1. Removing cases containing events in the first or last 20% of the period covered by the event log (objective: filtering incomplete traces)
2. Iterating over remaining traces. With probability $\frac{1}{2}$ we alter a full trace. If so, we apply with probability $\frac{1}{3}$ one of the following options (for $x = max\{1, 20\%$ avg. trace length$\}$).
   (a) we remove the first x activities (results in a trace prefix)
   (b) we remove the last x activities (results in a trace postfix)
   (c) we remove the first x and last x activities (results in a trace infix)

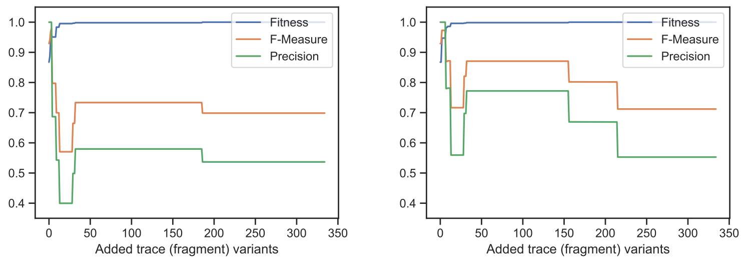
If the above procedure yields empty traces or empty trace fragments, we ignore them. We calculate fitness and precision using the log after the first step, as described above. For (TFS-)IPD, we discover an initial model from the 1% most frequent full trace variants using IM. The source code of our experiments, of (TFS-)IPD, and further results are available online.[5]

5.2 Results

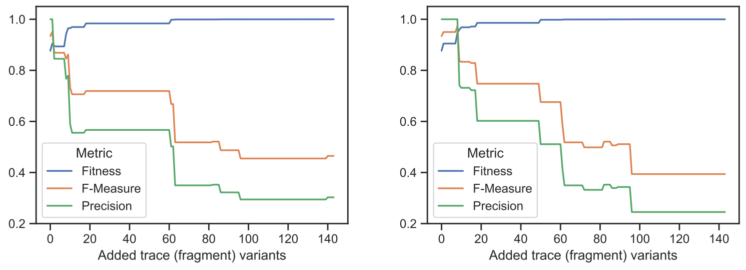
Fig. 8 compares IPD with TFS-IPD for three different event logs. Both approaches start from the identical initial model, and we add the same trace (fragment) variants in the same order (starting from the most frequent one). Across all logs, TFS-IPD significantly outperforms IPD in most cases. Especially in the beginning, we quickly obtain models with fitness around >90% where TFS-IPD outperforms IPD regarding precision. Note that the goal of IPD and TFS-IPD is not to necessarily incorporate all behavior because real-life event logs often contain noise and are, therefore, typically filtered.

[4] BPI Ch. 2020–Request for Payment (DOI: 10.4121/uuid:52fb97d4-4588-43c9-9d04-3604d4613b51)
Road Traffic Fine Management (DOI: 10.4121/uuid:270fd440-1057-4fb9-89a9-b699b47990f)
Receipt log (DOI: 10.4121/uuid:a07386a5-7be3-4367-9535-70bc9e77dbe6)

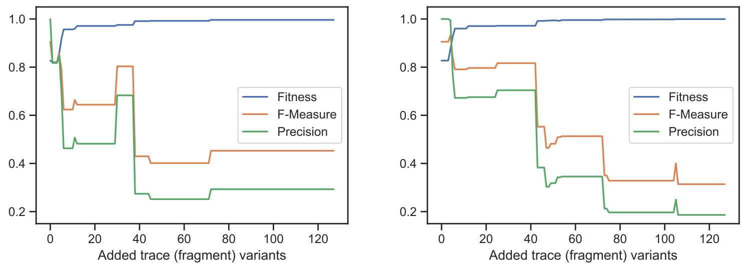
[5] https://github.com/fit-daniel-schuster/trace-fragment-supporting-incremental-process-discovery



(a) Road Traffic Fine Management (IPD vs. TFS-IPD)



(b) BPI CH. 2020–Request for Payment (IPD vs. TFS-IPD)



(c) Receipt of environmental permit applications (IPD vs. TFS-IPD)

Fig. 8: Comparing IPD (left) and TFS-IPD (right)

Table 1 lists the results for the different discovery approaches. Note that only TFS-IPD distinguishes between full traces and trace fragments; other approaches treat trace fragments as full traces. We observe that TFS-IPD often discovers process models of similar or even higher quality regarding the metrics shown. As expected, the more trace fragments are added by an approach, the higher the fitness, but the precision decreases. In short, TFS-IPD often learns more precise process models for comparable fitness values than other approaches.

6 Conclusion

We presented an IPD approach supporting trace fragments—prefix, infix, and postfix traces. Supporting trace fragments and thus incomplete data within process discovery is a novelty, as the general practice regarding trace fragments usually focuses on filtering or considering fragments as full traces. We have im-

Table 1: Model quality metrics rounded to two decimal points for the different approaches and different percentage values of added trace (fragment) variants

Approach	% of added trace (fragment) variants	Event log								
		Road Traffic Fine Management			BPI Ch. 2020–Request for Payment			Receipt of environmental permit applications		
		F-measure	fitness	precision	F-measure	fitness	precision	F-measure	fitness	precision
Trace-fragment-supporting IPD	20	0.87	1.00	0.77	0.75	0.99	0.60	0.82	0.97	0.70
	40	0.87	1.00	0.77	0.68	1.00	0.51	0.48	0.99	0.31
	60	0.80	1.00	0.67	0.51	1.00	0.34	0.33	1.00	0.20
	80	0.71	1.00	0.55	0.39	1.00	0.25	0.33	1.00	0.20
	100	0.71	1.00	0.55	0.39	1.00	0.25	0.31	1.00	0.19
IPD [17]	20	0.73	1.00	0.58	0.72	0.98	0.57	0.64	0.97	0.48
	40	0.73	1.00	0.58	0.72	0.98	0.57	0.40	0.99	0.25
	60	0.70	1.00	0.54	0.49	1.00	0.32	0.45	1.00	0.29
	80	0.70	1.00	0.54	0.45	1.00	0.29	0.45	1.00	0.29
	100	0.70	1.00	0.54	0.46	1.00	0.30	0.45	1.00	0.29
IM [13]	20	0.76	1.00	0.61	0.63	1.00	0.46	0.76	0.97	0.62
	40	0.72	1.00	0.57	0.68	1.00	0.24	0.42	0.84	0.28
	60	0.56	1.00	0.39	0.44	1.00	0.28	0.25	1.00	0.15
	80	0.65	1.00	0.48	0.39	1.00	0.24	0.28	1.00	0.17
	100	0.67	1.00	0.50	0.37	1.00	0.23	0.33	1.00	0.20
IMf (0.9) [13]	20	0.81	0.78	0.84	0.52	0.54	0.50	0.76	0.97	0.62
	40	0.81	0.78	0.84	0.26	0.64	0.17	0.42	0.84	0.28
	60	0.75	0.66	0.86	0.17	0.65	0.10	0.25	1.00	0.15
	80	0.71	0.66	0.77	0.43	0.86	0.29	0.28	1.00	0.17
	100	0.71	0.66	0.77	0.17	0.64	0.10	0.33	1.00	0.20
ETM (default settings, 60s timeout) [8]	20	0.51	0.99	0.34	0.68	0.92	0.54	0.75	0.90	0.64
	40	0.51	1.00	0.34	0.69	0.97	0.53	0.71	0.86	0.60
	60	0.82	0.77	0.89	0.63	0.96	0.47	0.60	0.87	0.46
	80	0.54	0.98	0.38	0.69	0.95	0.54	0.62	0.87	0.48
	100	0.52	0.98	0.35	0.64	0.96	0.48	0.67	0.87	0.54

plemented the proposed approach, including functionalities for handling trace fragments, in the open-source process mining tool *Cortado* [20]. Our experimental results indicate distinguishing trace fragments from full traces leads to high-quality models. While this paper focused on the foundational algorithmic aspects of supporting trace fragments in IPD, we plan to conduct a case study to investigate how process analysts can utilize trace fragments in real-world settings. Further, we plan to extend the sub-model freezing functionality for IPD [18] to support trace fragments as well.

References

1. Adriansyah, A.: Aligning observed and modeled behavior. Ph.D. thesis, Eindhoven University of Technology (2014)
2. Armas Cervantes, A., van Beest, N.R.T.P., La Rosa, M., Dumas, M., García-Bañuelos, L.: Interactive and incremental business process model repair. In: On the Move to Meaningful Internet Systems, LNCS, vol. 10573. Springer (2017)

3. Augusto, A., Conforti, R., Dumas, M., La Rosa, M., Maggi, F.M., Marrella, A., Mecella, M., Soo, A.: Automated discovery of process models from event logs: Review and benchmark. IEEE TKDE **31**(4) (2019)
4. Beerepoot, I., et al.: The biggest business process management problems to solve before we die. Computers in Industry **146** (2023)
5. Bergenthum, R., Desel, J., Lorenz, R., Mauser, S.: Process mining based on regions of languages. In: Business Process Management, LNCS, vol. 4714. Springer (2007)
6. Bernard, G., Andritsos, P.: Truncated trace classifier. removal of incomplete traces from event logs. In: Enterprise, Business-Process and Information Systems Modeling, LNBIP, vol. 387. Springer (2020)
7. Bezerra, F., Wainer, J., van der Aalst, W.M.P.: Anomaly detection using process mining. In: Enterprise, Business-Process and Information Systems Modeling, LNBIP, vol. 29. Springer (2009)
8. Buijs, J., van Dongen, B.F., van der Aalst, W.M.P.: A genetic algorithm for discovering process trees. In: Congress on Evolutionary Computation. IEEE (2012)
9. Carmona, J., van Dongen, B.F., Solti, A., Weidlich, M.: Conformance Checking. Springer (2018)
10. Dixit, P.M., Buijs, J.C.A.M., van der Aalst, W.M.P.: Prodigy : Human-in-the-loop process discovery. In: 12th International Conference on Research Challenges in Information Science (RCIS). IEEE (2018)
11. Fahland, D., van der Aalst, W.M.P.: Repairing process models to reflect reality. In: Business Process Management, LNCS, vol. 7481. Springer (2012)
12. Greco, G., Guzzo, A., Lupia, F., Pontieri, L.: Process discovery under precedence constraints. ACM Transactions on Knowledge Discovery from Data **9**(4) (2015)
13. Leemans, S.J.J.: Robust Process Mining with Guarantees. Springer (2022)
14. Polyvyanyy, A., van der Aalst, W.M.P., ter Hofstede, A.H.M., Wynn, M.T.: Impact-driven process model repair. ACM Transactions on Software Engineering and Methodology **25**(4) (2017)
15. Reichert, M., Weber, B.: Enabling Flexibility in Process-Aware Information Systems. Springer (2012)
16. Schuster, D., Föcking, N., van Zelst, S.J., van der Aalst, W.M.P.: Conformance checking for trace fragments using infix and postfix alignments. In: Cooperative Information Systems, LNCS, vol. 13591. Springer (2022)
17. Schuster, D., van Zelst, S.J., van der Aalst, W.M.P.: Incremental discovery of hierarchical process models. In: RCIS, LNBIP, vol. 385. Springer (2020)
18. Schuster, D., van Zelst, S.J., van der Aalst, W.M.P.: Freezing sub-models during incremental process discovery. In: Conceptual Modeling, LNCS, vol. 13011. Springer (2021)
19. Schuster, D., van Zelst, S.J., van der Aalst, W.M.P.: Utilizing domain knowledge in data-driven process discovery: A literature review. Computers in Industry **137** (2022)
20. Schuster, D., van Zelst, S.J., van der Aalst, W.M.P.: Cortado: A dedicated process mining tool for interactive process discovery. SoftwareX **22** (2023)
21. Solé, M., Carmona, J.: Incremental process discovery. In: Transactions on Petri Nets and Other Models of Concurrency V, LNCS, vol. 6900. Springer (2012)
22. van Dongen, B.F., Alves de Medeiros, A.K., Wen, L.: Process mining: Overview and outlook of Petri net discovery algorithms. In: Transactions on Petri Nets and Other Models of Concurrency II, LNCS, vol. 5460. Springer (2009)
23. de Weerdt, J., de Backer, M., Vanthienen, J., Baesens, B.: A multi-dimensional quality assessment of state-of-the-art process discovery algorithms using real-life event logs. Information Systems **37**(7) (2012)