Incremental Discovery of Hierarchical Process Models

Daniel Schuster[1(✉)], Sebastiaan J. van Zelst[1,2], and Wil M. P. van der Aalst[1,2]

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

Abstract. Many of today's information systems record the execution of (business) processes in great detail. Process mining utilizes such data and aims to extract valuable insights. Process discovery, a key research area in process mining, deals with the construction of process models based on recorded process behavior. Existing process discovery algorithms aim to provide a "push-button-technology", i.e., the algorithms discover a process model in a completely automated fashion. However, real data often contain noisy and/or infrequent complex behavioral patterns. As a result, the incorporation of all behavior leads to very imprecise or overly complex process models. At the same time, data pre-processing techniques have shown to be able to improve the precision of process models, i.e., without explicitly using domain knowledge. Yet, to obtain superior process discovery results, human input is still required. Therefore, we propose a discovery algorithm that allows a user to incrementally extend a process model by new behavior. The proposed algorithm is designed to localize and repair nonconforming process model parts by exploiting the hierarchical structure of the given process model. The evaluation shows that the process models obtained with our algorithm, which allows for incremental extension of a process model, have, in many cases, superior characteristics in comparison to process models obtained by using existing process discovery and model repair techniques.

Keywords: Process mining · Incremental process discovery · Process trees · Process model repair

1 Introduction

Process discovery is one of the three main fields in *process mining*, along with *conformance checking* and *process enhancement* [4]. In process discovery, the data generated during process executions and stored in information systems are utilized to generate a process model that describes the observed behavior. We refer to such data as *event data*. The obtained process models are used for a variety of purposes, e.g., to provide insights about the actual process performed and to analyze and improve performance and compliance problems.

F. Dalpiaz et al. (Eds.): RCIS 2020, LNBIP 385, pp. 417–433, 2020.
https://doi.org/10.1007/978-3-030-50316-1_25

Most process discovery techniques are fully automated, i.e., no interaction with the algorithm is possible during discovery. These techniques require event data as input and return a process model that describes the given observed behavior. Moreover, it is not directly possible, i.e., using process discovery techniques, to extend an existing process model with additional behavior, except by re-applying the algorithm to the entire extended event data. *Process model repair* techniques have been developed to add additional behavior to an existing model. However, they are not designed to be applied iteratively to a given event log to mimic an (incremental) process discovery algorithm.

In this paper, we propose an approach to *incrementally* discover process models. The algorithm allows the user to incrementally discover a process model by adding the behavior, trace by trace, to an existing process model. Thereby, the process model under construction gets incrementally extended. Hence, our approach combines the usually separate phases of event data filtering and discovery. In addition, the algorithm offers the possibility at any point in time to "auto-complete", i.e., observed behavior not yet processed is automatically added to the process model under construction. Our approach takes behavior that is not yet described by the process model and detects which parts of the process model must be altered. We focus on hierarchical, also called block-structured, process models and exploit their structure to determine the process model parts that must be changed. The evaluation of our proposed approach shows that the obtained process models have a comparable and in many cases superior quality compared to non-incremental process discovery algorithms, which have to be executed on the whole extended event data each time behavior is added. Furthermore, the conducted experiments show that our proposed approach outperforms an existing process model repair technique [12] in many cases.

The remainder of the paper is structured as follows. We present related work in Sect. 2. In Sect. 3, we present concepts, notations and definitions used throughout the paper. In Sect. 4, we present our novel approach to incrementally discover process models. Afterwards, we discuss the results of the conducted experiments in Sect. 5. Finally, we summarize the paper in Sect. 6.

2 Related Work

Various process discovery algorithms exist. An overview is beyond the scope of this paper, hence, we refer to [11]. We mainly focus on process model repair techniques, incremental and interactive process discovery.

The term *process model repair* was introduced in [12] and an extended algorithm to repair a process model was presented in [13]. In the paper, an event log L and a process model P is assumed, i.e., a Petri net, which does not accept all traces in L. The goal is to find a process model P' that accepts L. In comparison to our proposed approach, an essential goal for the authors is that the repaired model P' is structurally similar to the original model P since their focus is on model repair and not on process discovery. Since our proposed approach is an incremental algorithm for process discovery, similarity of the resulting model to the original model is not a requirement.

In [15], an *incremental process discovery* architecture was introduced that is based on merging new discovered process models into existing ones. In detail, an existing process model P is assumed and, for unseen behavior, a new process model is discovered and then merged into the existing model P. Furthermore, the approach is explicitly designed to work in an automated fashion. Two other approaches [14,19] calculate ordering relations of activities based on the given process model and on a yet unprocessed event log. The two obtained relations are then merged together and are used to retrieve a model. In [7] the authors describe a repair approach that incrementally highlights deviations in a process model with respect to a given event log. The user has to manually repair this deviations under the guidance of the algorithm.

Next to incremental process discovery algorithms, there is the field of *interactive process mining* [9]. In [10] an interactive process discovery algorithm is presented that assumes constant feedback from a user. Moreover, the user controls the algorithm by specifying how the process model should be altered. The algorithm supports the user by indicating favourable actions, e.g, where to place an activity in the process model (and also provides an "auto-complete option"). Furthermore, the algorithm ensures that the process model under construction retains certain properties, i.e., soundness.

3 Background

In this section, we introduce notations and definitions used in this paper.

Given an arbitrary set X, we denote the set of all sequences over X as X^*, e.g., $\langle a, b, b\rangle \in \{a, b, c\}^*$. We denote the empty sequence by $\langle\rangle$. For a given sequence σ, we denote its length as $|\sigma|$ and for $i \in \{1, \ldots, |\sigma|\}$, $\sigma(i)$ represents the i-th element of σ. Given two sequences σ and σ', we denote the concatenation of these two sequences by $\sigma \cdot \sigma'$. For instance, $\langle a\rangle \cdot \langle b, c\rangle = \langle a, b, c\rangle$. 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 given traces σ, σ', the set of all interleaved sequences is denoted by $\sigma \diamond \sigma'$. For example, $\langle a, b\rangle \diamond \langle c\rangle = \{\langle a, b, c\rangle, \langle a, c, b\rangle, \langle c, a, b\rangle\}$. We extend the $\diamond$ operator to sets of sequences. Let S_1, S_2 be two sets of sequences. $S_1 \diamond S_2$ denotes the set of interleaved sequences, i.e., $S_1 \diamond S_2 = \{\sigma_1 \diamond \sigma_2 | \sigma_1 \in S_1 \wedge \sigma_2 \in S_2\}$.

For a set X, a multi-set over X allows multiple appearances of the same element. Formally, a multi-set is a function $f{:}X \rightarrow \mathbb{N}_0$ that assigns a multiplicity to each element in X. For instance, given $X = \{a, b, c\}$, a multi-set over X is $[a^3, c]$, which contains three times an element a, no b and one c. We denote all possible multi-sets over X as $\mathcal{B}(X)$. Furthermore, given two multi-sets X and Y, $X \uplus Y$ denotes the union of two multi-sets, e.g., $[x^2, a] \uplus [x, y^2] = [x^3, a, y^2]$.

Next, we introduce projection functions. Given a set X, a sequence $\sigma \in X^*$ and $X' \subseteq X$. We recursively define $\sigma_{\downarrow_{X'}} \in X'^*$ with: $\langle\rangle_{\downarrow_{X'}} = \langle\rangle$, $(\langle x\rangle \cdot \sigma)_{\downarrow_{X'}} = \langle x\rangle \cdot \sigma_{\downarrow_{X'}}$ if $x \in X'$ and $(\langle x\rangle \cdot \sigma)_{\downarrow_{X'}} = \sigma_{\downarrow_{X'}}$ otherwise.

Let $t = (x_1, \ldots, x_n) \in X_1 \times \ldots \times X_n$ be an n-tuple over n sets. We define projection functions that extract a specific element of t, i.e., $\pi_1(t) = x_1, \ldots, \pi_n(t) = x_n$. For example, $\pi_2((a, b, c)) = b$.

Table 1. Example of an event log

Event-id	Case-id	Activity name	Timestamp	···
···	···	···	···	···
200	13	create order (c)	2020-01-02 15:29:24	···
201	27	receive payment (r)	2020-01-02 15:44:34	···
202	43	dispatch order (d)	2020-01-02 16:29:24	···
203	13	pack order (p)	2020-01-02 19:12:13	···
204	13	cancel order (a)	2020-01-03 11:32:21	···
···	···	···	···	···

Analogously, given a sequence of length m with n-tuples $\sigma = \langle(x_1^1, \ldots, x_n^1), \ldots, (x_1^m, \ldots, x_n^m)\rangle$, we define $\pi_1^*(\sigma) = \langle x_1^1, \ldots, x_1^m\rangle, \ldots, \pi_n^*(\sigma) = \langle x_n^1, \ldots, x_n^m\rangle$. For instance, $\pi_2^*(\langle(a, b), (a, c), (b, a)\rangle) = \langle b, c, a\rangle$.

3.1 Event Data and Event Logs

The execution of (business) processes generates event data in the corresponding information systems. Such data describe the activities performed, which process instance they belong to and they contain various metadata about the activities performed. Activities performed in the context of a specific process instance are referred to as a *trace*, i.e., a sequence of activities.

Consider Table 1 in which we present an example of an event log. For instance, if we consider all events related to the case-id 13, we observe the trace ⟨create order (c), pack order (p), cancel order (a)⟩. For simplicity, we abbreviate activities with letters. A *variant* describes a unique sequence of activities which can occur several times in an event log. Since, in the context of this paper, we are only interested in the traces that occurred, we define an event log as a multiset of traces. Note that, event data as depicted in Table 1 can be translated easily into a multiset of traces.

Definition 1 (Event Log). *Let $\mathcal{A}$ denote the universe of activities. An* event log *is a multiset of sequences over $\mathcal{A}$, i.e., $L \in \mathcal{B}(\mathcal{A}^*)$.*

3.2 Process Models

A process model describes the (intended) behavior of a process. Many process modeling formalisms exist, ranging from informal textual descriptions to mathematical models with exact execution semantics. In the field of process mining, *workflow nets* [1] are often used to represent process models since concurrent behavior can be modelled in a compact manner. In this paper we focus on *process trees* that represent hierarchical structured, sound workflow nets, i.e., block-structured workflow nets [16]. We formally define process trees in Definition 2.

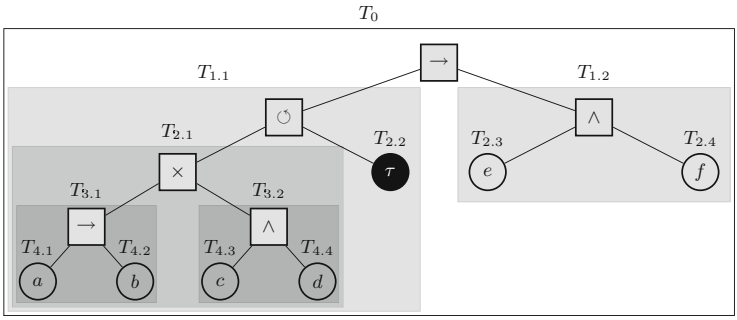


Fig. 1. Example of a process tree T_0

Definition 2 (Process Tree). *Let $\mathcal{A}$ be the universe of activities and let $\tau \notin \mathcal{A}$. Let $\oplus = \{\rightarrow, \times, \wedge, \circlearrowleft\}$ be the set of process tree operators.*

- *given an arbitrary $a \in \mathcal{A} \cup \{\tau\}$, a is a process tree*
- *given $n \geq 1$ process trees $T_1, T_2, \ldots, T_n$ and an operator $\bullet \in \{\rightarrow, \times, \wedge\}$, $T = \bullet(T_1, T_2, \ldots, T_n)$ is a process tree*
- *given two process trees T_1, T_2, $T = \circlearrowleft(T_1, T_2)$ is a process tree*

We denote the set of all process trees over $\mathcal{A}$ as $\mathcal{T}_\mathcal{A}$. Furthermore, we denote for a process tree T the set of its leaf nodes by L_T. Note that, by definition, leaf nodes always contain activities or the silent activity τ, and inner nodes and the root node always contain process tree operators. Consider Fig. 1 which shows an example of a process tree T_0. Note that the tree can also be presented textually: $\rightarrow(\circlearrowleft(\times(\rightarrow(a, b), \wedge(c, d)), \tau), \wedge(e, f))$.

For given process trees T and T', we call T' a *subtree* of T if T' is contained in T. For instance, $T_{3.2}$ is a subtree of $T_{1.1}$ (Fig. 1). Given two subtrees T_x, T_y of a tree T, we define the *lowest common ancestor (LCA)* as the tree T_{LCA} such that the distance between T's root node and T_{LCA}'s root node is maximal and T_x, T_y are contained in T_{LCA}. For example, given the two subtrees $T_{4.2}$ and $T_{2.2}$ of T_0 (Fig. 1), $T_{1.1}$ is the LCA of $T_{4.2}$ and $T_{2.2}$.

In the following, we first informally describe the semantics of a process tree and afterwards, present formal definitions. The sequence operator $\rightarrow$ indicates that the subtrees have to be sequentially executed. For example, the root of process tree T_0 is a sequence operator. Hence, the left subtree $T_{1.1}$ has to be executed before the right one $T_{1.2}$. The loop operator $\circlearrowleft$, which has by definition two subtrees, contains a loop body which is the first subtree and a redo part, the second subtree. The loop body has to be executed at least once. Afterwards, the redo part can be optionally executed. In case the redo part is executed, the loop body must be executed afterwards again. The choice operator $\times$, i.e., exclusive or, indicates that exactly one subtree must be executed. The parallel operator $\wedge$ indicates a parallel (interleaved) execution of the subtrees. For instance, for the tree $T_{1.2}$ the activities e and f can be executed in any order.

a	b	$\gg$	$\gg$	c	f	$\gg$
a	b	τ	d	c	f	e
$(T_{4.1})$	$(T_{4.2})$	$(T_{2.2})$	$(T_{4.4})$	$(T_{4.3})$	$(T_{2.4})$	$(T_{2.3})$

a	$\gg$	b	c	f	$\gg$
$\gg$	a	b	$\gg$	f	e
	$(T_{4.1})$	$(T_{4.2})$		$(T_{2.4})$	$(T_{2.3})$

Fig. 2. Two possible alignments for process tree T_0 (Fig. 1) and $\langle a, b, c, f \rangle$

We denote the language of a process tree $T \in \mathcal{T}_\mathcal{A}$, i.e., the set of accepted traces over $\mathcal{A}$, as $\mathcal{L}(T)$. For instance, $\langle a,b,f,e \rangle, \langle d,c,a,b,e,f \rangle \in \mathcal{L}(T_0)$ and $\langle d,c,a,e,f \rangle, \langle a,c,e,f \rangle \notin \mathcal{L}(T_0)$. Next, we define the semantics of process trees based on [4].

Definition 3 (Semantics of Process Trees). *For a process tree $T \in \mathcal{T}_\mathcal{A}$, we recursively define its language $\mathcal{L}(T)$.*

- *if* $T = a \in \mathcal{A}$, $\mathcal{L}(T) = \{\langle a \rangle\}$
- *if* $T = \tau$, $\mathcal{L}(T) = \{\langle\rangle\}$
- *if* $T = \rightarrow(T_1, \ldots, T_n)$, $\mathcal{L}(T) = \mathcal{L}(T_1) \cdot \ldots \cdot \mathcal{L}(T_n)$
- *if* $T = \wedge(T_1, \ldots, T_n)$, $\mathcal{L}(T) = \mathcal{L}(T_1) \diamond \ldots \diamond \mathcal{L}(T_n)$
- *if* $T = \times(T_1, \ldots, T_n)$, $\mathcal{L}(T) = \mathcal{L}(T_1) \cup \ldots \cup \mathcal{L}(T_n)$
- *if* $T = \circlearrowleft(T_1, T_2)$, $\mathcal{L}(T) = \{\sigma_1 \cdot \sigma'_1 \cdot \sigma_2 \cdot \sigma'_2 \ldots \cdot \sigma_m \mid m \geq 1 \wedge \forall 1 \leq i \leq m(\sigma_i \in \mathcal{L}(T_1)) \wedge \forall 1 \leq i \leq m(\sigma'_i \in \mathcal{L}(T_2))\}$

3.3 Alignments

Alignments have been developed to map observed behavior onto modeled behavior [5]. They are used to determine if a given trace conforms to a given process model. In the case of deviations, alignments indicate the detected deviations in the process model and in the trace.

In Fig. 2, two possible alignments for the process tree T_0 (Fig. 1) and the trace $\langle a, b, c, f \rangle$ are given. The first row, the trace part, always corresponds to the given trace (ignoring the skip symbol $\gg$). The second row, the model part, always corresponds to a trace that is accepted by the given process model (ignoring $\gg$).

An alignment move corresponds to a single column in Fig. 2. We distinguish four different alignment moves. Synchronous moves, highlighted in light gray, indicate that the observed behavior in the trace can be replayed in the process model. For example, the first two moves of the left alignment in Fig. 2 represent synchronous moves, i.e., the observed activities a and b could be replayed in the process model. Log moves, highlighted in black, indicate additional observed behavior that cannot be replayed in the process model and therefore represent a deviation. Model moves, highlighted in dark gray, indicate that behavior is missing in the given trace according to the process model. Model moves can be further differentiated into visible and invisible model moves. Given the first alignment from Fig. 2, the first model move represents an invisible model move because the executed activity is the silent activity τ. Note that invisible model moves do not represent deviations. The second model move represents a visible

a	b	$\gg$	c	d	$\gg$	a	b	e	f
a	b	τ	c	d	τ	a	b	e	f
$(T_{4.1})$	$(T_{4.2})$	$(T_{2.2})$	$(T_{4.3})$	$(T_{4.4})$	$(T_{2.2})$	$(T_{4.1})$	$(T_{4.2})$	$(T_{2.3})$	$(T_{2.4})$

T_0	T_0	T_0	T_0	T_0	T_0	T_0	T_0	T_0	T_0
$T_{1.1}$	$T_{1.1}$	$T_{1.1}$	$T_{1.1}$	$T_{1.1}$	$T_{1.1}$	$T_{1.1}$	$T_{1.1}$		
$T_{2.1}$	$T_{2.1}$		$T_{2.1}$	$T_{2.1}$		$T_{2.1}$	$T_{2.1}$		
$T_{3.1}$	$T_{3.1}$					$T_{3.1}$	$T_{3.1}$		
			$T_{3.2}$	$T_{3.2}$					
								$T_{1.2}$	$T_{1.2}$

(a) Alignment and listing of subtrees containing the executed process tree leave nodes

T_0	$[\langle a, b, c, d, a, b, e, f\rangle]$
$T_{1.1}$	$[\langle a, b, c, d, a, b\rangle]$
$T_{2.1}$	$[\langle a, b\rangle^2, \langle c, d\rangle]$
$T_{3.1}$	$[\langle a, b\rangle^2]$
$T_{3.2}$	$[\langle c, d\rangle]$
$T_{1.2}$	$[\langle e, f\rangle]$

(b) Sub event logs

Fig. 3. Calculation of the sub event log for T_0 (Fig. 1) and $L = [\langle a, b, c, d, a, b, e, f\rangle]$

model move since the executed activity $d \neq \tau$. Visible model moves represent deviations because a modeled activity was not observed.

Definition 4 (Alignment). *Let $\mathcal{A}$ denote the universe of activities, let $\sigma \in \mathcal{A}^*$ be a trace and let $T \in \mathcal{T}_\mathcal{A}$ be a process tree with the set of leaf nodes L_T. A sequence $\gamma \in ((\mathcal{A} \cup \{\gg\}) \times (L_T \cup \{\gg\}))^*$ is an alignment iff:*

1. $\sigma = \pi_1^*(\gamma)_{\downarrow_\mathcal{A}}$
2. $\pi_2^*(\gamma)_{\downarrow_{L_T}} \in \mathcal{L}(T)$
3. $(\gg, \gg) \notin \gamma$
4. $(a_1, a_2) \notin \gamma$ *for* $a_1 \in \mathcal{A}, a_2 \in L_T$ *s.t.* $a_1 \neq a_2$

For given T and σ, the set of all possible alignments is denoted by $\Gamma(\sigma, T)$. Since many alignments exist for a given trace and a process model, there is the concept of optimal alignments. In general, an optimal alignment minimizes the number of mismatches between the process model and the trace. To determine optimal alignments, costs are assigned to alignment moves. A cost minimal alignment for a given trace and a process model is considered to be an optimal alignment. In this paper, we assume the *standard cost function* that assigns cost 0 to synchronous and invisible model moves. Furthermore, it assigns cost 1 to visible model and log moves. Note that there can be several optimal alignments for a given model and trace. The calculation of optimal alignments was shown to be reducible to the shortest path problem [5]. Note that there can be several optimal alignments.

We denote the set of optimal alignments for σ and T by $\bar{\Gamma}(\sigma, T)$. Observe that, under the standard cost function, an alignment $\gamma \in \bar{\Gamma}(\sigma, T)$ indicates a deviation between the trace σ and the process tree T if the costs are higher than 0.

3.4 Sub Event Logs for Process Trees

In this section, we define the concept of a sub event log. Assume a process tree T and a perfectly fitting event log L, i.e., $\{\sigma \in L\} \subseteq \mathcal{L}(T)$. We define for each

Algorithm 1. Calculation of sub event logs for process trees

Input: $L \in \mathcal{B}(\mathcal{A}^*), T \in \mathcal{T}_\mathcal{A}$ (Assumption: $\{\sigma \in L\} \subseteq \mathcal{L}(T)$))
Output: sub event log for each subtree of T, i.e., $s : \mathcal{T}_\mathcal{A} \to \mathcal{B}(\mathcal{A}^*)$
begin
1 **forall the** *subtrees* T' *of* T **do**
2 $s(T') \leftarrow [\,]$ // initialize sub event logs
3 **forall the** $\sigma \in L$ **do**
4 let $\gamma \in \bar{\Gamma}(\sigma, T)$ // calculate optimal alignment for σ and T
5 **forall the** *subtrees* T' *of* T **do**
6 $t(T') \leftarrow \langle\rangle$ // initialize trace for each subtree
7 **for** $i \in \{1, \ldots, |\gamma|\}$ **do**
8 $m \leftarrow \gamma(i)$ // extract i-th alignment move
9 $T_l \leftarrow \pi_2(m)$ // extract executed process leaf node
10 **forall the** *subtrees* T' *of* T **do**
11 **if** T_l *is subtree of* $T' \vee T_l = T'$ **then**
12 $t(T') \leftarrow t(T') \cdot \langle \pi_2(m) \rangle$ // add executed activity to T''s trace
13 **else if** $t(T') \neq \langle\rangle$ **then**
14 $s(T') \leftarrow s(T') \uplus [t(T')]$ // add trace to T''s sub event log
15 $t(T') \leftarrow \langle\rangle$ // reset trace
16 **return** s

subtree in T a sub event log that reflects which parts of the given traces from L are handled by the subtree.

Assume the event log $L = [\langle a, b, c, d, a, b, e, f \rangle]$ and the process tree T_0 (Fig. 1). The event log L is perfectly fitting because the only trace $\sigma = \langle a, b, c, d, a, b, e, f \rangle$ in L is accepted by T_0, i.e., $\sigma \in \mathcal{L}(T_0)$. To calculate sub event logs, we first calculate alignments for each trace in the given event log. The alignment of σ and T_0 is depicted in the upper part of Fig. 3a. Since σ is accepted by T_0, we only observe invisible model moves and synchronous moves. Below the depicted alignment, all subtrees are listed that contain the executed leaf nodes. For example, the first executed leaf node a ($T_{4.1}$) is a subtree of $T_0, T_{1.1}, T_{2.1}$ and $T_{3.1}$.

Obviously, all executed leaf nodes are subtrees of T_0. Hence, we add the complete trace to T_0's sub event log (Fig. 3b). Note that the sub event log of the whole process tree, i.e., T_0, is always equal to the given event log. The subtree $T_{1.1}$ contains all executed leaf nodes from the 1st leaf a ($T_{4.1}$) to the last execution of b ($T_{4.2}$). This sequence of executed leaf nodes corresponds to the trace $\langle a, b, c, d, a, b \rangle$ that is added to $T_{1.1}$'s sub event log. The subtree $T_{2.1}$ contains the first two executed leaf nodes, i.e., a ($T_{4.1}$) and b ($T_{4.2}$). The 3rd executed leaf node τ ($T_{2.2}$) is not contained in $T_{2.1}$. Therefore, we add the trace that corresponds to the first two executed leaf nodes, i.e., $\langle a, b \rangle$, to $T_{2.1}$'s sub event log. The 4th and 5th executed leaf nodes are again a subtree of $T_{2.1}$, but not the 6th leaf node. Hence, we add the trace $\langle c, d \rangle$, which corresponds to the 4th

and 5^{th} executed leaf node, to $T_{2.1}$'s sub event log. The 7^{th} and 8^{th} executed leaf nodes are again subtrees of $T_{2.1}$, and therefore, we add the trace $\langle a, b \rangle$ to $T_{2.1}$'s sub event log. By processing the alignment for each subtree in the presented way, we obtain sub event logs for each subtree in T_0 as shown in Fig. 3b.

In Algorithm 1 we present a formal description of the sub event log calculation. We successively calculate an alignment for each trace in L (line 4). First, we initialize an empty trace for each subtree T' of T that will be eventually added to T''s sub log (line 6). Next, we iterate over the alignment, i.e., the executed process tree leaf nodes since the alignment contains only synchronous and invisible model moves. For every subtree that contains the current executed leaf node, we add the corresponding activity to its trace (line 14). If the current executed leaf node is not contained in a subtree T', we add the corresponding trace, if it is not empty, to T''s sub log (line 14) and reset the trace (line 15).

4 Incremental Discovery of Process Trees

In this section, we present our approach to incrementally discover process trees. In general, we assume an initially given process tree T, which is incrementally modified trace by trace. If a new trace σ_i is not accepted by the current process tree T, we calculate an optimal alignment and localize the nonconforming parts in T. We then modify the identified process tree part(s) to make the obtained process tree T' accept σ_i. Afterwards, we continue with T' and process the next trace σ_{i+1} analogously. In case a trace is already accepted by the current process tree, we move on to the next trace without modifying the current process tree.

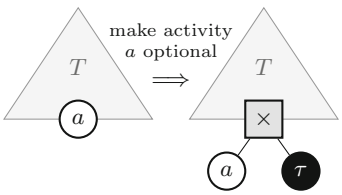
The remainder of this section is structured as follows. First, we introduce an approach to repair a single deviation. We then present a more advanced approach that additionally handles blocks of deviations and uses the previously mentioned approach as a fallback option.

4.1 Repairing Single Deviations

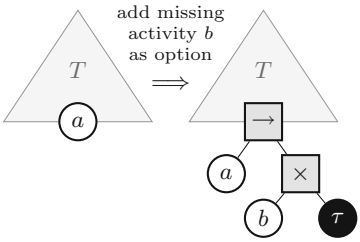
In this section, we present an approach to repair a single alignment move which corresponds to a deviation in a process tree. We assume that a process tree T, a trace σ and an alignment $\gamma \in \bar{\Gamma}(T, \sigma)$ are given. Moreover, we assume that potential deviations in the given alignment γ are repaired from left to right. Next, we present process tree modifications to repair various deviations.

Assume that the given alignment contains a visible model move, i.e., $\gamma = \frac{\cdots \;|\; \gg \;\cdots}{\cdots \;|\; a \;\cdots}$. Since a model move indicates that a modeled activity was not observed, we make the corresponding leaf node a optional. Therefore, we replace the leaf node a by the choice construct $\times(a, \tau)$ (Fig. 4a). This ensures that the activity is optional in the process model and no longer causes a model move.

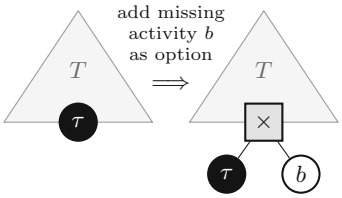
If the alignment contains a log move, we have to differentiate two cases, i.e., the standard case and the root case. For the standard case the alignment is of the form $\gamma = \frac{\cdots \;|\; a \;\; b \;\cdots}{\cdots \;|\; a \;\; \gg \;\cdots}$ or $\gamma = \frac{\cdots \;|\; \gg \;\; b \;\cdots}{\cdots \;|\; \tau \;\; \gg \;\cdots}$, i.e., directly before the deviation,



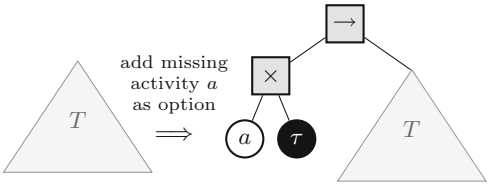
(a) Repairing a visible model move



(b) Repairing a log move with preceding synchronous move



(c) Repairing a log move with preceding invisible model move



(d) Repairing a log move (root case)

Fig. 4. Repairing a single deviation - process tree repair modifications

there is either a synchronous or an invisible model move. In this case, we extend the process tree such that we ensure that after the activity a or τ it is possible to optionally execute the missing activity in the process model. Therefore, we replace the leaf node a by $\rightarrow (a, \times(b, \tau))$ (Fig. 4b). Accordingly, we change the process tree for a preceding invisible model movement, i.e., we replace τ by $\times(\tau, b)$ (Fig. 4c). In the other case, the root case, the log move is at the beginning of the alignment, i.e., $\gamma = \begin{smallmatrix} a & \cdots \\ \gg & \cdots \end{smallmatrix}$. In this case, we add the possibility to optionally execute the missing activity a before the current tree. Let T be the given process tree, we alter T to $\rightarrow (\times(a, \tau), T)$ (Fig. 4d), i.e., we extend the given process tree at the root node. Since we assume that deviations in an alignment are repaired from left to right, one of the two cases always applies to log moves.

The presented approach allows us to fix multiple deviations in an alignment by separately repairing all deviations from left to right. Furthermore, the approach is deterministic because in each iteration we repair a deviation of the given alignment. Moreover, we always add behavior to the process tree and never remove behavior, i.e., we always extend the language of accepted traces. In the next section, we present a further approach that additionally handles blocks of deviations and uses the presented approach as a fallback option.

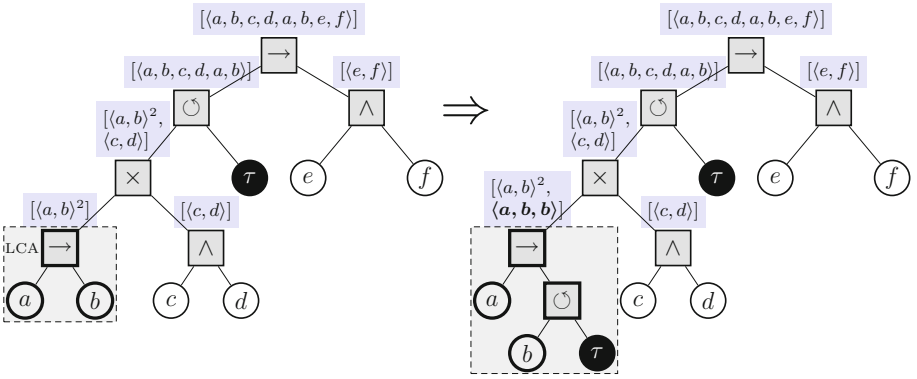


Fig. 5. Conceptual idea of the proposed LCA approach

4.2 Repairing Blocks of Deviations

In this section, we present our more advanced approach that additionally handles blocks of deviations. First, we present the conceptual idea and an example. Afterwards, we introduce the algorithm.

Conceptual Idea. The proposed LCA approach assumes an initial process tree T, a perfectly fitting event log L, i.e., $\{\sigma \in L\} \subseteq \mathcal{L}(T)$, and a trace σ. The event log L represents the traces processed so far, i.e., traces that must be accepted by the process tree T. Furthermore, the LCA approach assumes a process tree discovery algorithm $disc : \mathcal{B}(\mathcal{A}^*) \rightarrow \mathcal{T}_\mathcal{A}$ that, given any event log, returns a perfectly fitting process tree.[1] The proposed LCA approach returns a process tree T' such that the given trace σ and the log L are accepted.

Assume the process tree T_0 (Fig. 1), the event log $L = [\langle a, b, c, d, a, b, e, f\rangle]$ and the trace $\sigma = \langle a, b, b, e, f\rangle$, which is not accepted by T_0. When we apply the LCA approach, we first calculate an optimal alignment.

$\boldsymbol{a}$	$\boldsymbol{b}$	$\boldsymbol{b}$	e	f
a	$\gg$	b	e	f
$(T_{4.1})$		$(T_{4.2})$	$(T_{2.3})$	$(T_{2.4})$

We always repair the first occurring (block of successively occurring) deviation(s). In the given example, we observe a log move on b and before and after the deviation a synchronous move. Next, we calculate the LCA of a $(T_{4.1})$ and b $(T_{4.2})$ that encompass the deviation. The LCA is $T_{3.1}$ and its sub event log is $[\langle a, b\rangle^2]$ as depicted in the left process tree in Fig. 5. The calculated LCA corresponds to a subtree that causes the deviation and must therefore be changed. Hence, we add the trace $\langle \boldsymbol{a, b, b}\rangle$ to $T_{3.1}$'s sub event log and apply the given $disc$ algorithm on the extended sub event log. For instance, we could get $\rightarrow (a, \circlearrowleft (b, \tau))$ depending

[1] For example, the Inductive Miner algorithm [16] fulfills the listed requirements.

on the concrete instantiation of *disc*. Finally, we replace $T_{3.1}$ by the discovered process tree, i.e., $T_0' = \rightarrow (\circlearrowleft (\times(\underline{\rightarrow(a, \circlearrowleft(b, \tau))}, \wedge(c, d)), \tau), \wedge(e, f))$.

Next, we again compute an alignment of the updated process tree T_0' and γ. In case of further deviations, we repair them in the above-described manner. Otherwise, we return the modified process tree and the extended event log $L' = L \uplus [\langle a, b, b, e, f \rangle]$. Hereinafter, we formally describe the algorithm in detail.

Algorithmic Description. First, an optimal alignment is calculated for the given trace σ and the process tree T, i.e., $\gamma \in \bar{\Gamma}(T, \sigma)$. In case there exist no deviations, we return T. In case of deviations, we repair the first (block of) deviation(s). Assume the alignment is of the form as depicted below.

$$\gamma = \begin{array}{|c|c|c|c|c|c|c|c|c|} \hline \cdots & x_i' & \cdots & x_i & \cdots \textit{ deviation(s) } \cdots & x_j & \cdots & x_j' & \cdots \\ \hline \cdots & T_i' & \cdots & T_i & \cdots \textit{ deviation(s) } \cdots & T_j & \cdots & T_j' & \cdots \\ \hline \end{array}$$

We have a (block of) deviation(s), i.e., visible model moves and/or log moves, and directly before and after the (block of) deviation(s) there is no deviation, i.e., either a synchronous move or an invisible model move in each position. Let T_i be the process tree leaf node executed before the deviation(s) and T_j be the one after the deviation(s). We then calculate the LCA of T_i and T_j, hereinafter referred to as T_{LCA}. Note that T_i and T_j are subtrees of T_{LCA}.

Next, we check which of the executed process tree leaf nodes preceding T_i are also a subtree of T_{LCA}. Assume T_i' is a subtree of T_{LCA} and all process tree leafs from T_i' until T_i are a subtree of T_{LCA} too. Besides, either the process tree leaf node executed before T_i' is not a subtree of T_{LCA}, or T_i' is the first executed leaf node in the alignment. Since we repair the first occurring (block of) deviation(s), we know that before T_i only synchronous or invisible model moves occur.

Analogously, we check which executed process tree leaf nodes after T_j are a subtree of T_{LCA}. Note that there is a difference because log moves and visible model moves potentially occur after T_j because we always repair the first (block of) deviation(s). However, except that we ignore log moves, we proceed as described above. Let T_j' be the last leaf node s.t. all executed leaf nodes from T_j to T_j' are a subtree of T_{LCA}. In addition, either the next executed leaf node after T_j' is not a subtree of T_{LCA} or there exist no more executed tree leaves after T_j'.

Given T_i' and T_j', we add the trace $\langle x_i', \ldots, x_j' \rangle \downarrow_{\mathcal{A}}$ (ignoring $\gg$) to T_{LCA}'s sub event log $L_{T_{LCA}}$, i.e., $L'_{T_{LCA}} = L_{T_{LCA}} \uplus [\langle x_i', \ldots, x_j' \rangle \downarrow_{\mathcal{A}}]$. Next, we apply the given *disc* algorithm on $L'_{T_{LCA}}$ and replace T_{LCA} by the newly discovered process tree $disc(L'_{T_{LCA}})$. Since the process tree $disc(L'_{T_{LCA}})$ accepts the trace $\langle x_i', \ldots, x_j' \rangle \downarrow_{\mathcal{A}}$, we repaired the first (block of) deviation(s). Afterwards, we again calculate an optimal alignment on the updated process tree and σ. If there are still deviations, we again repair the first (block of) deviation(s).

In the case that before or after the (block of) deviation(s) no process tree leaf node was executed and hence, we cannot compute a LCA, we apply the repair approach from the previous section, which repairs a single deviation, on the first log or visible model move. Afterwards, we apply the above described algorithm

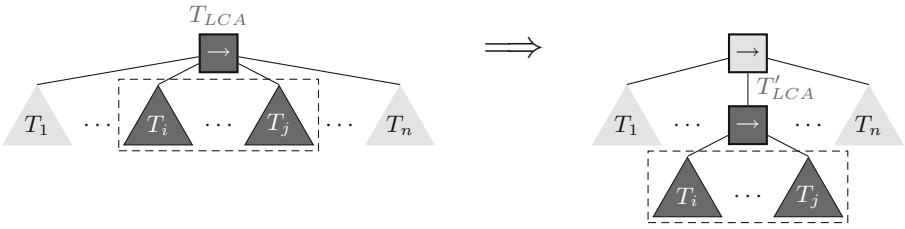


Fig. 6. Pulling down a sequence LCA in the process tree hierarchy

again on the updated process tree. Thereby, we ensure that the proposed LCA approach is deterministic since in every iteration a (block of) deviation(s) is repaired either by rediscovering the determined LCA or by applying the single deviation repair approach. Next, we refine the calculation of a LCA to minimize the affected subtrees getting altered.

Lowering an LCA in the Tree Hierarchy. For a process tree with a low height, it is likely that the proposed LCA approach determines the root as LCA and therefore, re-discovers the entire process tree. This behavior is not desirable since in this case the quality of the returned process tree solely depends on the given *disc* algorithm and most often, deviations can be repaired at a lower subtree.

To keep the affected subtrees that get rediscovered small, we introduce expansion rules to lower the detected LCA in the tree hierarchy. For this purpose, we use language preserving reduction rules in the reverse direction [17].

Assume that the determined LCA, denoted by T_{LCA}, contains the sequence operator and has n child nodes. Furthermore, assume that the deviation was localized between two children of T_{LCA}, i.e., between T_i and T_j. Hence, the process tree is of the form $T_{LCA} = \rightarrow (T_1, \ldots, T_i, \ldots, T_j, \ldots T_n)$. Then we know that all child nodes of T_{LCA} which are not between T_i and T_j are not responsible for the deviation. Hence, we can cut $T_i, \ldots, T_j$ and replace the nodes by a new sequence operator with the cut subtrees as children, i.e., $\rightarrow (T_1, \ldots, \rightarrow (T_i, \ldots, T_j), \ldots T_n)$ (Fig. 6). If we re-compute the LCA, we will get $T'_{LCA} = \rightarrow (T_i, \ldots, T_j)$.

In case the LCA contains the parallel or choice operator, we lower the detected LCA in a similar manner. Assume the deviation was localized between two children of T_{LCA}, i.e., between T_i and T_j, and that T_{LCA} has n child nodes. Hence, the process tree is of the form $T_{LCA} = \bullet(T_1, \ldots, T_i, \ldots, T_j, \ldots T_n)$ for $\bullet \in \{\times, \wedge\}$. In this case, we extract the two child nodes T_i and T_j and pull them one level down in the process tree: $T_{LCA} = \bullet(T_1, \ldots, \bullet(T_i, T_j), \ldots T_n)$.

5 Evaluation

We evaluated the proposed LCA approach on the basis of a publicly available, real event log. In the following section, we present the experimental setup. Subsequently, we present and discuss the results of the conducted experiments.

5.1 Experimental Setup

In the experiments, we compare the LCA approach against the Inductive Miner (IM) [16], which discovers a process tree that accepts the given event log, and the model repair approach presented in [13]. Note that the repair algorithm does not guarantee to return a hierarchical process model. We implemented the LCA approach extending *PM4Py* [8], a process mining library for Python. Since both the LCA approach and the IM algorithm guarantee the above mentioned properties for the returned process tree, we use the IM algorithm as a comparison algorithm. Furthermore, we use the IM algorithm inside our LCA approach as an instantiation of the *disc*-algorithm, which is used for rediscovering subtrees.

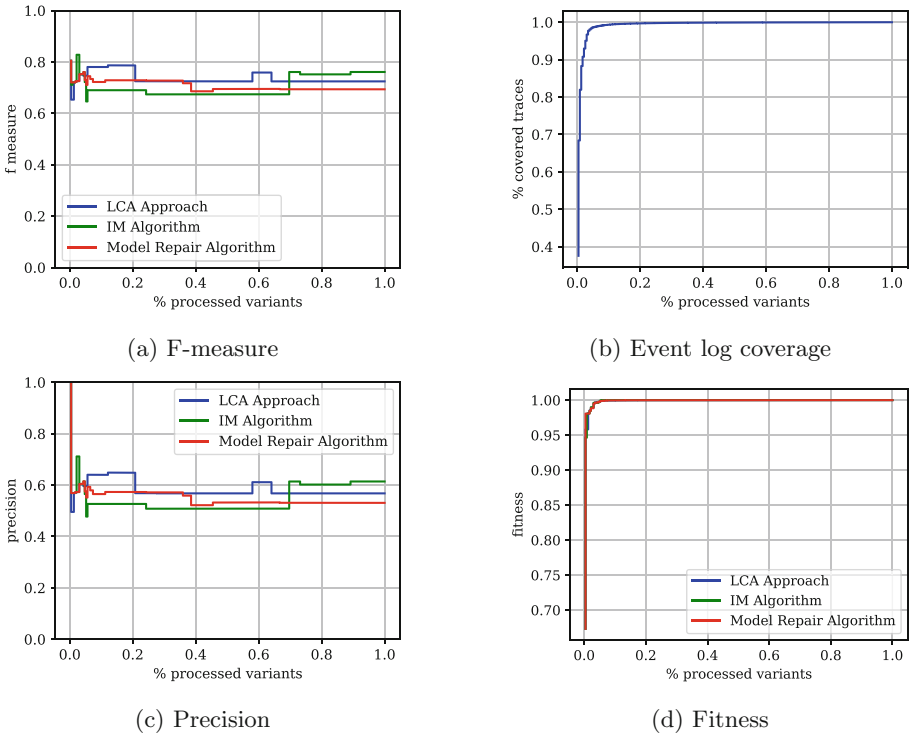
As input, we use a publicly available event log that contains data about a road fine management process [18]. We use the complete event log, e.g., we do not filter outliers. We sorted the event log based on variant frequencies in descending order, i.e., the most occurring variant first. We chose this sorting since in real applications it is common to consider first the most frequent behavior and filter out infrequent behavior. Note that the order of traces influences the resulting process model in our approach and in the model repair approach.

To compare the obtained process models, we use the f-measure regarding the whole event log. The f-measure takes the harmonic mean of the precision and the fitness of a process model with respect to a given event log. Fitness reflects how good a process model can replay a given event log. In contrast, precision reflects how much additional behavior next to the given event log is accepted by the process model. The aim is that both the fitness and the precision and thus, the f-measure are close to 1. We use alignment-based approaches for fitness [2] and precision calculation [6].

The procedure of the conducted experiments is described below. First, we discover a process tree on the first variant with the IM algorithm since the LCA approach and the model repair algorithm require an initial process model. Note that the LCA approach can be used with any initial model. Afterwards, we add variant by variant to the initially given process model with the LCA approach. Analogously, we repair the initially given process model trace by trace with the model repair algorithm. In addition, we iteratively apply the IM algorithm on the 1^{st} variant, the $1^{st} + 2^{nd}$ variant, etc.

5.2 Results

In Fig. 7 we present the obtained results. We observe that the f-measures (Fig. 7a) of the process models obtained by the LCA approach are higher compared to models obtained by the IM and the model repair algorithm for the majority of processed variants. Note that the IM algorithm returns process trees with a higher f-measure in the end. However, for real process discovery applications it is unusual to incorporate the entire behavior in an event log. Reasons for not trying to incorporate all observed behavior are data quality issues, outliers and incomplete behavior. Furthermore, observe that after processing the first 15% of all variants, we already cover >99% of all recorded traces (Fig. 7b) and obtain



(a) F-measure

(b) Event log coverage

(c) Precision

(d) Fitness

Fig. 7. Results on f-measure, precision, fitness and event log coverage

a process model with the LCA approach that outperforms the other techniques (Fig. 7a). The jump in the f-measure for the IM algorithm at 70% processed variants results from the fact that the IM gets more behavior as input, i.e., a larger event log, and therefore, detects a more suited pattern which leads to a more precise process tree in this case.

In Fig. 7c the precision values are depicted. These influence the f-measure most because we guarantee perfect fitness w.r.t. the added trace variants. Also here we can see that for most of the processed variants the LCA approach delivers more precise models. However, if we add more than 70% of all variants, the IM algorithm suddenly delivers more precise models. For fitness (Fig. 7d) the differences between LCA and IM are minor.

The higher f-measure of the LCA approach in many cases compared to the IM algorithm can be explained by the differences in the *representational bias* [3]. The LCA approach may return models that have duplicate labels, i.e., the same activity can occur in multiple leaf nodes. In comparison, the models returned by the IM algorithm do not allow for duplicate labels. Note that also the model repair algorithm allows duplicate labels.

6 Conclusion

In this paper, we presented a novel algorithm to incrementally discover a process tree. The approach utilizes the hierarchical structure of a process tree to localize the deviating subtree and rediscovers it. The conducted experiments show that the obtained process models have in many cases better quality in comparison to models produced by a process discovery and model repair algorithm with same guarantees about the resulting model, i.e., replay fitness. Actually, it is surprising that our incremental discovery approach works so well. We do not use domain knowledge and see many ways to improve the technique. The potential to outperform existing approaches even further is therefore high.

While most process discovery algorithms are fully automated, i.e., they assume an event log and return a process model, the LCA approach is able to incrementally add behavior to an existing model. Therefore, it can be used to evolve a process model trace by trace. This makes it easy for the user to see the impact on the process model when a trace is added. Thus, by the incremental selection of traces by a user, the usually separated phases of event data filtering and process discovery are connected. Hence, our approach enables the user to interactively discover a process model by selecting iteratively which behavior should be covered by the process model.

As future work, we plan to investigate both the impact of the initially given model and the ordering of traces incrementally given to the LCA approach on the resulting process model. Furthermore, we plan to explore different strategies for determining the deviating subtree next to the LCA calculation. We also plan to further develop this algorithm into an advanced interactive process discovery algorithm that provides further user interaction possibilities next to the incremental selection of traces/observed behavior.

References

1. van der Aalst, W.M.P.: The application of petri nets to workflow management. J. Circ. Syst. Comput. **8**(1), 21–66 (1998). https://doi.org/10.1142/S0218126698000043
2. van der Aalst, W.M.P., Adriansyah, A., van Dongen, B.F.: Replaying history on process models for conformance checking and performance analysis. Wiley Interdiscip. Rev. Data Min. Knowl. Discov. **2**(2), 182–192 (2012). https://doi.org/10.1002/widm.1045
3. van der Aalst, W.M.P.: On the representational bias in process mining. In: Reddy, S., Tata, S. (eds.) Proceedings of the 20th IEEE International Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises, WETICE 2011, Paris, France, 27–29 June 2011, pp. 2–7. IEEE Computer Society (2011). https://doi.org/10.1109/WETICE.2011.64
4. van der Aalst, W.M.P.: Process Mining: Data Science in Action, 2nd edn. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49851-4
5. Adriansyah, A.: Aligning Observed and Modeled Behavior. Ph.D. thesis, Eindhoven University of Technology, Department of Mathematics and Computer Science, July 2014. https://doi.org/10.6100/IR770080

6. Adriansyah, A., Munoz-Gama, J., Carmona, J., van Dongen, B.F., van der Aalst, W.M.P.: Measuring precision of modeled behavior. Inf. Syst. E-Bus. Manag. **13**(1), 37–67 (2015). https://doi.org/10.1007/s10257-014-0234-7
7. 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: Panetto, H., Debruyne, C., Gaaloul, W., Papazoglou, M., Paschke, A., Ardagna, C.A., Meersman, R. (eds.) OTM 2017. LNCS, vol. 10573, pp. 53–74. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-69462-7_5
8. Berti, A., Zelstvan Zelst, S.J., Aalstvan der Aalst, W.M.P.: Process mining for python (PM4Py): bridging the gap between process-and data science. In: Proceedings of the ICPM Demo Track 2019, Co-Located with 1st International Conference on Process Mining (ICPM 2019), Aachen, Germany, 24–26 June 2019, pp. 13–16 (2019). http://ceur-ws.org/Vol-2374/
9. Dixit, P.: Interactive process mining. Ph.D. thesis, Department of Mathematics and Computer Science, June 2019
10. Dixit, P.M., Verbeek, H.M.W., Buijs, J.C.A.M., van der Aalst, W.M.P.: Interactive data-driven process model construction. In: Trujillo, J.C., Davis, K.C., Du, X., Li, Z., Ling, T.W., Li, G., Lee, M.L. (eds.) ER 2018. LNCS, vol. 11157, pp. 251–265. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-00847-5_19
11. van Dongen, B.F., Alves de Medeiros, A.K., Wen, L.: Process mining: overview and outlook of petri net discovery algorithms. In: Jensen, K., van der Aalst, W.M.P. (eds.) Transactions on Petri Nets and Other Models of Concurrency II. LNCS, vol. 5460, pp. 225–242. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00899-3_13
12. Fahland, D., van der Aalst, W.M.P.: Repairing process models to reflect reality. In: Barros, A., Gal, A., Kindler, E. (eds.) BPM 2012. LNCS, vol. 7481, pp. 229–245. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32885-5_19
13. Fahland, D., van der Aalst, W.M.P.: Model repair: aligning process models to reality. Inf. Syst. **47**, 220–243 (2015). https://doi.org/10.1016/j.is.2013.12.007
14. Kalsing, A., do Nascimento, G.S., Iochpe, C., Thom, L.H.: An incremental process mining approach to extract knowledge from legacy systems. In: Proceedings of the 14th IEEE International Enterprise Distributed Object Computing Conference, EDOC 2010, Vitória, Brazil, 25–29 October 2010, pp. 79–88. IEEE Computer Society (2010). https://doi.org/10.1109/EDOC.2010.13
15. Kindler, E., Rubin, V., Schäfer, W.: Incremental workflow mining based on document versioning information. In: Li, M., Boehm, B., Osterweil, L.J. (eds.) SPW 2005. LNCS, vol. 3840, pp. 287–301. Springer, Heidelberg (2006). https://doi.org/10.1007/11608035_25
16. Leemans, S.J.J., Fahland, D., van der Aalst, W.M.P.: Discovering block-structured process models from event logs - a constructive approach. In: Colom, J.-M., Desel, J. (eds.) PETRI NETS 2013. LNCS, vol. 7927, pp. 311–329. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38697-8_17
17. Leemans, S.J.J., Fahland, D., van der Aalst, W.M.P.: Scalable process discovery and conformance checking. Softw. Syst. Model. **17**(2), 599–631 (2018). https://doi.org/10.1007/s10270-016-0545-x
18. Leonide Leoni, M., Mannhardt, F.: Road traffic fine management process - event log. 4TU.Centre for Research Data (2015), https://doi.org/10.4121/uuid:270fd440-1057-4fb9-89a9-b699b47990f5. Accessed 12 Oct 2019
19. Sun, W., Li, T., Peng, W., Sun, T.: Incremental workflow mining with optional patterns and its application to production printing process. Int. J. Intell. Control Syst. **12**, 45–55 (2007)