BULLETIN OF THE POLISH ACADEMY OF SCIENCES
TECHNICAL SCIENCES, Vol. 71(1), 2023, Article number: e143551
DOI: 10.24425/bpasts.2022.143551

CONTROL AND INFORMATICS

Continuous update of business process trees using Continuous Inductive Miner

Tomasz P. PAWLAK* and **Bartosz GÓRKA**

Institute of Computing Science, Poznan University of Technology, Poland

Abstract. Business processes are omnipresent in nowadays economy: companies operate repetitively to achieve their goals, e.g., deliver goods, complete orders. The business process model is the key to understanding, managing, controlling, and verifying the operations of a company. Modeling of business processes may be a legal requirement in some market segments, e.g., financial in the European Union, and a prerequisite for certification, e.g., of the ISO-9001 standard. However, business processes naturally evolve, and continuous model adaptation is essential for rapid spot and reaction to changes in the process. The main contribution of this work is the Continuous Inductive Miner (CIM) algorithm that discovers and continuously adapts the process tree, an established representation of the process model, using the batches of event logs of the business process. CIM joins the exclusive guarantees of its two batch predecessors, the Inductive Miner (IM) and the Inductive Miner – directly-follows-based (IMd): perfectly fit and sound models, and single-pass event log processing, respectively. CIM offers much shorter computation times in the update scenario than IM and IMd. CIM employs statistical information to work around the need to remember event logs as IM does while ensuring the perfect fit, contrary to IMd.

Key words: business process; process mining; business intelligence; event log; concept drift.

1. INTRODUCTION

1.1. Background

In modern business, every activity leaves a digital mark: customers' orders, supply deliveries, wire transfers, etc. We call these marks the *events* and the process that raised them the *business process* [1]. Computers routinely collect *event logs*. The eXtensible Event Stream (XES) [2] is an industry standard for event logs with established software support. A well-formed XES log consists of traces corresponding to business cases, e.g., paper submissions. A trace is a list of events raised by the activities, e.g., invite reviewers, collect reviews, decide, and send decision. Section 2.1 details the event log.

Business processes evolve, e.g., employees reorder or skip some activities for work efficiency, technology adapts to customers' demands, or the legal environment changes. Such changes in the process are called *concept drift*. The event log, as a record of reality, reflects the concept drift. This opens the way to building *descriptive models* that describe the same process at different stages of evolution. The descriptive model is a kind of aggregation of the evidence in the event log to the form suitable for human inspection. It facilitates understanding of the process and may reveal several deficiencies, e.g., bottlenecks, waste of resources, or rare abuses that without the descriptive model would remain undetected. The descriptive models capture real human behavior and interactions with machines. This property enables a variety of simulation, what-if, and predictive analyses of process operations.

*e-mail: tpawlak@cs.put.poznan.pl

Manuscript submitted 2022-04-15, revised 2022-09-09, initially accepted for publication 2022-10-09, published in February 2023.

In contrast, the *normative model* represents the designed flow of the process. Maintaining the normative model up to date as the process evolves incurs extra costs. The non-maintained normative model for a long-living process may not align with reality or be not optimal. By juxtaposing the normative and descriptive models of the same process, one can spot differences, accept changes in the process, or apply improvements that prevent unwanted behavior.

In this study, we employ the representation of a *process tree* [3] of the process model. The process tree is a hierarchy of control flow operators finished with activities in the leaves. The control flow operators specify the choice and order of subtrees to execute. Section 2.2 introduces the process tree in detail.

We employ three gain-type criteria for process models: fitness, precision, and generalization. *Fitness* measures the part of the event log represented by the model. *Precision* measures the part of the business cases allowed by the model observed in the event log. *Generalization* assesses how well the model represents the 'idea' of the process rather than remembers parts of the event log. When looking for (rare) deficiencies in the process, obtaining the perfect fit model is crucial even at the expense of the other criteria. Section 2.3 discusses the quality criteria.

We distinguish two variants of the *process tree discovery problem*: *batch* and *update*. The batch problem may be a part of a computer-assisted operating audit, where the auditor analyses the past process operations. Solving this problem helps to identify the deficiencies and abuses in the process, possibly distant in the past, but does not allow us to proactively signal them. In contrast, the update problem is to update the existing process tree using the differences in the event log as soon as they occur, revealing now and here the concept drift in the process. In

© 2023 The Author(s). This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/)

this study, we accept the differences of two forms: new traces entering the event log and old traces leaving the event log. Updating the process tree with new traces enables the detection of new process behavior. Forgetting old traces is crucial to eliminate from the process tree the behavior that no longer holds. The update problem aims to update the process tree as fast as possible using the *differential event log* only. This requirement makes the batch discovery techniques inapplicable. Section 2.3 details both variants of the problem.

Process Mining (PM) [4] focuses on, but is not limited to, algorithms for the discovery of process models from event logs. The Inductive Miner (IM) algorithm family [5–8] is designed specifically for process trees. These algorithms have several desirable properties, e.g., the base IM algorithm [5] produces the perfect fit process tree to the event log at the expense of remembering the parts of the event log. Another variant [8] reads every event in the event log exactly once without guaranteeing a perfect fit. Section 3 reviews the existing algorithms. Section 4 details the IM algorithm.

1.2. Goals and contributions of this study

This leads to the main hypothesis of this study: *An update algorithm for a process tree using differential event log produces new process tree with fitness, precision, generalization, and discovery time no worse than for a process tree produced from scratch using IM fed with the equivalent batch event log.*

The main contribution of this study is the *Continuous Inductive Miner* (CIM) algorithm described in Section 5. It hybridizes and extends algorithms [5, 8] in several directions:

- Given an existing process tree and a differential event log, CIM identifies the parts of this tree affected by the differential event log and updates only these parts.
- CIM reduces memory consumption by using basic statistics rather than remembering the (parts of) event log with the tree nodes, as in [5].
- CIM reads every event once like [8], but unlike [8], it guarantees a perfect fit.

In Section 6, we decompose the primary research hypothesis into parts and verify them experimentally. Section 7 discusses the achievements of this study. Section 8 concludes this work.

2. PRELIMINARIES

Sections 2.1 and 2.2 define the formal objects used in this study: the event log and the process tree, respectively. Section 2.3 poses the process tree discovery problem using these objects.

2.1. Event log

We borrow the definition of *event log* from the XES standard [2] and limit it to the features relevant to this study:

Definition 1. The event e is a set of uniquely labeled attributes. $\#_{name}(e)$ is the name of the activity that raised e, and $\#_{time}(e)$ is the timestamp of e. Two events e_1 and e_2 equal if and only if $\#_{name}(e_1) = \#_{name}(e_2)$.

The trace $t = [e_1, e_2, \dots, e_n]$ is a sequence of events, and $|t|$ is the total number of events in t. Two traces t_1 and t_2 equal if and only if $|t_1| = |t_2|$ and the events at the same indices in t_1 and t_2 equal.

The event log $L = \left\{t_1^{k_1}, t_2^{k_2}, \dots, t_m^{k_m}\right\}$ is a multiset of traces, where superscripts $k_1, k_2, \dots, k_m$ refer to the numbers of occurrences of the traces $t_1, t_2, \dots, t_m$, respectively; $k_i = 1$ can be omitted.

We assume that an event log L refers to exactly one process, a trace $t \in L$ refers to exactly one business case of this process, and events $e \in t$ are ordered ascending by $\#_{time}(e)$.

For brevity, we abuse the notation and write down the events using their activity names. For example, the event log $L = \left\{[a,b,c,d,f]^2, [c,b,a,e,d,g]^3\right\}$ consists of trace $t_1 = [a,b,c,d,f]$ that occurred twice and trace $t_2 = [c,b,a,e,d,g]$ that occurred three times, where $a, b, \dots, g$ are the activities.

2.2. Process tree

We use the recursive definition of *process tree* from [3, 5, 8]:

Definition 2. Let A be a set of activities, and let $\odot$ be at least 2-ary control flow operator. Then, the process tree T is either $T = a$, where $a \in A$, or $T = \odot(T_1, T_2, \dots, T_n)$.

The process tree is either a degenerated tree of a single node with a single activity or a composition of process trees with control flow operators in the intermediate nodes. The control flow operators impose the partial order relation between the direct subtrees and hence between the activities in the leaves. Activities may be labeled or silent. A *silent activity* τ is not recorded in the event log. Silent activities are useful in technical structures, e.g., the exclusive choice between a silent activity and a labeled activity causes the latter to be optional. See Table 1 for other uses of silent activities. Process trees are block-structured: every subtree is a valid process tree independent of the others. This opens the way for the composition of high-level business processes using low-level ones.

Let us define the control flow operators:

Definition 3. Let the set of activities A be an alphabet, and let $\mathscr{L}(T)$ denote the language of T over A.[1] Then, $\mathscr{L}(\tau) = \{[]\}$, $\mathscr{L}(a) = \{[a]\}$ for all $a \in A \setminus \{\tau\}$, and $\mathscr{L}(\odot(T_1, T_2, \dots, T_n))$ depends on the control flow operator $\odot \in \{\times, \rightarrow, \wedge, \circlearrowleft\}$:

- For the *exclusive choice* operator $\times$, it is the union of the languages of the subtrees: $\mathscr{L}\big(\times(T_1,T_2,\dots,T_n)\big) = \bigcup_{i=1}^{n} \mathscr{L}(T_i)$.
- For the *sequence* operator $\rightarrow$, it is the Cartesian product of the languages of the subtrees: $\mathscr{L}\big(\rightarrow(T_1,T_2,\dots,T_n)\big) = \mathscr{L}(T_1) \times \mathscr{L}(T_2) \times \dots \times \mathscr{L}(T_n)$.
- For the *parallel* operator $\wedge$, it is the merge[2] of the languages of the subtrees: $\mathscr{L}\big(\wedge(T_1,T_2,\dots,T_n)\big) = \left\{\sigma : |\sigma| = \sum_{i=1}^{n} |\sigma_i|, \sigma_i \in \mathscr{L}(T_i), a_j \prec a_k \in \sigma_i \implies a_j \prec a_k \in \sigma\right\}$.

[1] A formal language $\mathscr{L}$ over alphabet A is the set of words over A. A word over A is a finite sequence of the symbols from A.

[2] The merge of words a and b is the word c with the interleaved symbols from both a and b such that the relative order of the symbols from each of a and b separately is preserved in c [9]. The merge of two languages is the set of all merges of all words from these languages.

- For the *loop* operator $\circlearrowleft$, it is the Cartesian product of the union over $m \in [0;\infty)$ of the m-ary Cartesian powers[3] of the language of the first subtree $\mathscr{L}(T_1)$ and the language of any remaining subtree and $\mathscr{L}(T_1)$: $\mathscr{L}\big(\circlearrowleft(T_1, T_2, \ldots, T_n)\big) = \bigcup_{m=0}^{\infty} \big(\bigcup_{i=2}^{n} \mathscr{L}(T_1) \times \mathscr{L}(T_i)\big)^m \times \mathscr{L}(T_1)$.

$\times$ is a decision point to choose the subtree to execute activities from; $\rightarrow$ imposes the left-to-right order relation between the activities from different subtrees; $\wedge$ imposes the series-parallel partial order relation [10] between the activities from different subtrees; and $\circlearrowleft$ imposes the cyclic order relation [11] between the activities from different subtrees, where the activities from the leftmost subtree execute first, then zero or more times the activities from any other subtree immediately followed by the activities from the leftmost subtree execute. Note that $\big|\mathscr{L}(\circlearrowleft (T_1, T_2, \ldots, T_n))\big| = \infty$ except if $\forall_{i=1}^{n} \mathscr{L}(T_i) = \{[]\}$.

Figure 1 shows an exemplary process tree for a paper review process. It consists of nine activities $A = \{a, b, c, d, e, f, g, h, \tau\}$. The $\rightarrow$ operator on the root node splits the process into three ordered phases. The $\wedge$ operator in the leftmost subtree indicates that the first phase consists of inviting reviewers in parallel, i.e., the activities a, b, and c run in any order but before any other activity. The $\circlearrowleft$ operator in the middle subtree indicates that its leftmost subtree runs once, then the choice is made in the loop to run the rightmost subtree and the leftmost again or go back to the root node. The leftmost subtree of the $\circlearrowleft$ node consists of the $\times$ operator that allows to either collect reviews (activity d) or exceed a timeout (τ). Note that the process tree is unable to tie the decision to invite an extra reviewer (e) with the timeout, and this decision must be made based on the other decision model. The $\times$ operator in the rightmost subtree of the root node consists of the decision on the paper, i.e., only one of the activities f, g, and h runs. Running either f, g, and h ends the process.

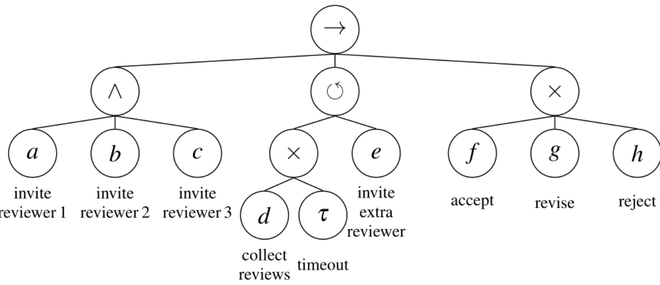


Fig. 1. An exemplary process tree for a paper review process

The process tree from Fig. 1 can be written in the prefix notation as $\rightarrow(\wedge(a,b,c), \circlearrowleft(\times(d,\tau),e), \times(f,g,h))$. We will use this notation later on.

In this study, we assume that all activities except the silent activities τ in a process tree are uniquely named. This imposes a certain bias of representation, as processes with duplicate activities cannot be represented except when the duplicate happens in a loop. To workaround, assign unique names to instances of the same activity.

The process tree is sound by definition. It must not contain deadlocks and livelocks – there always exists a path from the current execution state to the end of the process [3]. Every such path guarantees that the process completes properly, without leaving unfinished or pending activities behind. These properties make the process tree well-suited for modeling of business processes.

2.3. Process tree discovery problem

We first define the problem of batch process tree discovery and then use it to define the problem of process tree update.

Definition 4. Given an event log L the batch process tree discovery problem is to find a process tree T such that two functions are maximized: fitness $f(T,L)$ and precision $p(T,L)$.

We adopt the definition of $f(T,L)$ from the work [12] (there called f_L) and the definition of $p(T,L)$ from the work [13] (there called $etc_P(EL, PN)$). Both functions attain values from 0 (the worst) to 1 (the best). $f(T,L)$ measures the fraction of L reproducible using T, and $f(T,L) = 1$ iff $L \subseteq \mathscr{L}(T)$. In turn, $p(T,L)$ estimates the probability that a business case allowed by T occurs in L, and $p(T,L) = 1$ iff $\mathscr{L}(T) \subseteq L$. Calculating these functions relies on the conversion of T to the Petri net [14] and replying traces from L on this net, simultaneously calculating the statistics. These are well-known techniques with lengthy definitions and we omit the details for brevity.

Definition 4 poses the multi-objective optimization problem. Each objective can be trivially maximized at the expense of the other. For example, the process tree $T_1 = \circlearrowleft(\tau, a, b, \ldots, z)$ can produce any business case consisting of activities $a, b, \ldots, z$ and thus $f(T_1, L) = 1$ for every event log L of activities $A \subseteq \{a, b, \ldots, z\}$, however, $p(T_1, L) \approx 0$ because most event logs are finite. In contrast, a process tree $T_2 = \rightarrow(a, b, c, d, f)$ that effectively represents a single business case results in $p(T_2, L) = 1$ and $f(T_2, L) \approx \frac{1}{|L|}$ if $[a, b, c, d, f]$ occurs exactly once in L.

$f(T,L) = p(T,L) = 1$ is theoretically possible when, e.g., T remembers all traces of L and nothing else. However, process trees like that likely generalize poorly the behavior recorded in the event log. In this study, we assume that the goal is to find the descriptive process tree of the past reality, e.g., for auditing. By requiring $f(T,L) = 1$, the process tree guarantees to reproduce all business cases, including rare abuses and frauds. Note that by clustering the event log beforehand model discovery as in [15, 16], one can divide the event log into the mainstream behavior and the exceptional behavior and produce separate perfect fit process trees for them to spot the differences. The maximization of $p(T,L)$ under $f(T,L) = 1$ and maintaining generalization rather than memorizing L remain the challenge. In the experimental part, we assess generalization by calculating fitness using the separate *test* event log.

The solution to the problem of batch process tree discovery is useful when auditing a process on demand. However, processes live long and generate events in real-time, and the discovered process tree may become outdated over time. We handle this use case using the problem of process tree update, where an

[3] The 0-ary Cartesian power yields the empty word language $\mathscr{L} = \{[]\}$.

existing process tree T is to be updated to a process tree T' that reflects the changes in the event log.

Definition 5. The process tree update problem is a tuple (T, L, L^+, L^-), where T is a process tree, L is an event log such that $f(T,L) = 1$, L^+ and $L^- \subseteq L$ are differential logs of the traces to be inserted into and to be deleted from L, respectively. The goal is to find a process tree T' that maximizes $f(T', L \cup L^+ \setminus L^-)$ and $p(T', L \cup L^+ \setminus L^-)$.

We also require the perfect fit, i.e., $f(T', L \cup L^+ \setminus L^-) = 1$.

L in Definition 5 stands for the time window, and the differential logs L^+ and L^- represent the changes in this window due to the shift. The update algorithm is required to fit the given tree T to new traces in L^+ and drop the parts of T corresponding to L^-, which otherwise become redundant and hinder precision.

We require the update algorithm to build T' based solely on T, L^+, and L^- to prevent excessive memory usage due to storing time window L.

3. RELATED WORK

Section 3.1 reviews the algorithms for the discovery of process trees proposed to date. To our knowledge, very little has been done for the efficient updating of process trees as formalized in Definition 5. Therefore, Section 3.2 reviews process model updating techniques in general, including other representations. Section 3.3 lists the works that do not fit the first two categories but are related to this study in an another way.

3.1. Process tree discovery

The representation of the process tree originates in [3], which introduces five control flow operators but does not define their semantics formally. They are the four from Definition 3 and an OR operator. In this study, we do not use OR, as the Inductive Miner family of algorithms [5–8] does not use it, either. The work [3] shows that thanks to the guarantee of soundness, the search space of the process trees is smaller by hundreds of orders of magnitude than the search space of Petri nets [14] for the same number of activities. This property makes the process tree a well-suited representation for (meta-)heuristic search algorithms.

An Evolutionary Algorithm (EA) for process trees [17] uses a different definition of the $\circlearrowleft$ operator than in Definition 3. It is guided by the user's preferences and adapts the process trees for different criteria.

The Inductive Miner (IM) algorithm [5] builds in a polynomial-time process tree perfectly fitting the event log. IM uses the control flow operators from Definition 3. Given a process tree T and a large-enough event log produced by this tree $L \subseteq \mathscr{L}(T)$, IM builds another process tree T' based on L such that $\mathscr{L}(T') = \mathscr{L}(T)$. The disadvantage of IM is the requirement to store the sublogs corresponding to the subtrees, resulting in large memory usage.

The Inductive Miner for incomplete logs (IMin) [6] extends IM with the probabilistic fitting of the control flow operators to the event log. It employs an arbitrary handcrafted probability distribution. An experiment shows that IMin requires 87.5% of the actual directly-follows relations in the event log to rediscover the original tree. IMin trades the precision of the tree for generalization. It uses a Satisfiability Modulo Theories (SMT) solver to pick the right operators and so its run-time is no longer polynomial. IMin is also biased by the log size, as it may return different trees given the same log with every trace duplicated.

The Inductive Miner for infrequent behavior (IMi) [7] extends IM with several levels of heuristic filtering of noise. It drops rare traces, directly-follows relations, and base cases, etc., resulting in larger precision but imperfect fitness. IMi still remembers the parts of the log with the tree nodes.

The Inductive Miner directly-follows-based (IMd) [8] is an extension to IM, IMin, and IMi that removes the requirement of remembering the log with tree nodes. It detects the control flow operators using the directly-follows graph built from the log. IMd processes the log in a single-pass and runs in $O(|L| + |A|^3)$ time, effectively handling the logs of millions of traces and thousands of activities. IMd pays for that by poor handling of base cases — it is unable to detect whether the activity to be put in a leaf is, e.g., optional, certain, repeatable at least zero times, or repeatable at last once. Hence, IMd produces worse fit models than IM, IMin, and IMi, respectively.

Indulpet Miner (InM) [18] is an ensemble of IM, EA, and local process miner [19] that produces process trees. It maintains a trade-off between different quality characteristics offered by its components: high fitness of IM, better precision of EA, and smaller computational cost of the local process miner. However, it no longer provides the guarantees of IM except for the soundness of the resulting process trees. These characteristics make InM a good choice for mining the common behavior in the process but make it unable to reliably detect rare abuses and frauds.

Constructs Competition Miner (CCM) [20] produces models in a representation very similar to process trees. Contrary to IM, CCM finds the best-fit process construct for each tree node using trial-and-error. CCM produces well-fit trees but lacks the guarantee of the perfect fit.

None of the above algorithms offers all of these desirable properties together: the guaranteed soundness and the perfect fit of the resulting process model (only IM and IMd), the single-pass log processing (IMd), no need to remember log parts (IMd), well-handling of the base cases (IM, IMin, IMi, CCM). In contrast, the algorithm proposed in Section 5 has all these properties and allows for the efficient update of a process tree using the differential event logs.

3.2. Process model update

Dynamic CCM (DCCM) [21] is an extension of CCM [20] that handles the rediscovery of process tree-like models. It collects new events and traces as soon as available and periodically recalculates the model. Similar to CIM, DCCM identifies the parts of the model to recalculate. In contrast to CIM, the model update in DCCM is scheduled and not triggered by the change in the event log.

The work [22] detects the concept drift in Petri nets using adaptive windowing [23]. It does not localize the part affected by the concept drift and rediscovers the entire model if required.

Event-stream process discovery using abstract representations [24] is a framework for adopting the batch process discovery algorithms to the update scenario. It is based on the observation that most discovery algorithms build an intermediate representation of the event log and transform this representation into the process model. The general idea is to maintain this representation using an event stream and apply the transformation routines of the batch algorithm at hand. The authors verify their framework using the directly-follows graph as an intermediate representation and IM as the discovery algorithm. This framework does not maintain some guarantees of IM, e.g., the perfect fit, as it uses fast approximates of the directly-follows graph and the start and end activities. In contrast, CIM from Section 5 holds this guarantee.

The work [25] characterizes sudden, gradual, recurring, and incremental concept drifts in business processes and proposes the algorithm to detect the point in time and location in the Petri net of the occurrence of the concept drift.

Process Histories [26] is a technique of detecting the above-mentioned types of concept drift in the business processes. A process history consists of a series of process models in a stream-based abstract representation [24]. The authors employ IM to produce process trees and then transform them into that representation. The classification of the concept drift type is made based on several (heuristic) metrics calculated for the models in history.

The work [27] employs a declarative model of relations between activities and adapts it based on incoming events.

Several variants of Heuristics Miner (HM) for the update of Causal nets employ lossy counting and time windows to maintain several heuristic measures of support of the elements of the Causal net [28]. HM does not provide any guarantees for fitness, precision, or other criteria.

Another approach to updating Causal nets is Online Miner (OM) [29]. It uses a similar problem statement as in Definition 5, expect that it involves the Causal net representation. OM provides several guarantees that are not available in [28], e.g., the resulting Causal nets are guaranteed to be sound, perfect fit, and use the maximal bindings (be the most precise). The experimental comparison to [28] reveals the superiority of OM in several aspects.

3.3. Other related work

The Refined Process Structure Tree (RPST) [30, 31] is an algorithm for transforming unstructured process graphs into block-structured, tree-like process models. RPST relates to CIM in that both detect the structure in the process graph. However, in this study, we operate on the directly-follows graph with an arbitrary structure, and RPST is limited to certain classes of graphs, e.g., acyclic.

The work [32] extends Heuristics Miner [33] with the detection of the block structure in the process models, where each block corresponds to a subprocess. Contrary to CIM, it produces models in the BPMN representation [34] and lacks the guarantee of producing the perfect fit models.

Abstraction Workflow Schema (AWS) [35] is a technique for clustering large business processes into a tree-like hierarchy of simpler processes. AWS is similar to CIM in that both produce trees that reflect the hierarchy in business processes. AWS builds a non-executable decomposition of the process models mined using other algorithms, e.g., Heuristics Miner [33] and α algorithm [36]. On the contrary, CIM produces fully executable trees whose every subtree corresponds to an executable subprocess.

The survey [37] empirically compares some state-of-the-art algorithms for batch discovery of process models in the real-world application to modeling customer service in a European telecom. The comparison includes the α-family algorithms [36], Heuristics Miner [33], Genetic Miner [38], and AGNE [39], and reveals several deficiencies in these algorithms related to poor handling of bad-structured processes and noise.

The α-algorithm [36] is a very simple, yet fast algorithm to discover the Petri net. It features many deficiencies [4], hence many extensions exist. The extension [40] handles concurrency and short loops that occur at the same time. Another extension [41] detects several types of silent activities that occur in the process but are not logged. These extensions beat the base α-algorithm [36] on processes with these constructs.

Another survey [42] systematically reviews the batch discovery algorithms published in 2011-2017. It also empirically compares seven algorithms for the discovery of Petri nets and BPMN models having public available open-source implementations. The comparison includes, e.g., IM [5], the base variant of Inductive Miner, and Split Miner [43]. Virtually all evaluated algorithms fare poorly on event logs with infrequent behavior and require preprocessing of the event log. They favor different quality measures, e.g., IM wins the comparison on fitness, generalization, complexity but falls short on precision and execution time. In turn, Split Miner [43] wins on precision and execution time.

ProDiGen [44] is a multi-objective EA for Petri net discovery. It offers highly fit, precise, and small models at the same time, albeit with no guarantees for these criteria. In the experiment, it outperforms several other algorithms on one or more criteria. Although not demonstrated in [44], ProDiGen, like all EAs, naturally fits the update scenario.

CSC4.5 [45] is the algorithm for modeling business processes from event logs based on decision tree learning. This is a quite different technique, where the decision tree is an intermediate representation between the log and the resulting Linear Programming model [46].

4. BATCH DISCOVERY OF PROCESS TREES

This section details Inductive Miner directly-follows-based (IMd) [8] – the state-of-the-art algorithm for batch discovery of process trees that forms a base for CIM introduced in the next section.

Algorithm 1 shows IMd that transforms the given event log L into process tree. IMd works in two steps. In line 2, IMd calculates the directly-follows graph G from L, in which nodes correspond to the activities in L and arcs correspond to the pairs of activities directly following each other in at least one $t \in L$. In line 3, IMd recursively cuts G into disjoint subgraphs such

Algorithm 1. Inductive Miner directly-follows-based (IMd)

```
1: function INDUCTIVEMINER(L)
2:   G ← (A_L, ↦_L)                                   ▷ Construct DFG
3:   return SPLIT(G)                                   ▷ Build tree
4: function SPLIT(G)
5:   if |{a ∈ A}| = 1 : A ∈ G then
6:     return a
7:   if ⊙-cut applies to G, where ⊙ ∈ [×, →, ∧, ↻] then
8:     return ⊙(SPLIT(G_1), SPLIT(G_2), ..., SPLIT(G_n))
9:   return ↻ (τ, a_1, a_2, ..., a_n), where a_i ∈ G
```

that they run in an order corresponding to a control flow operator from Definition 3. The hierarchy of cuts transforms into the process tree. The following sections detail these steps.

4.1. Directly-follows graph

Definition 6. Let A_L be a set of activities in event log L, let A_L^s and A_L^k be the sets of activities that *start* and *end* at least one trace $t \in L$, respectively. Let $\mapsto_L$ be a directly-follows relation over L, i.e., $\mapsto_L = \left\{(a,b) \in A_L \times A_L : \exists_{[e_1,e_2,\ldots,e_n] \in L} \exists_{i=1}^{n-1} e_i = a \wedge e_{i+1} = b\right\}$. Let $\mapsto_L^+$ be the transitive closure of $\mapsto_L$, i.e., $\mapsto_L^+ = \left\{(a,b) : (a,b) \in \mapsto_L \vee ((a,c) \in \mapsto_L^+ \wedge (c,b) \in \mapsto_L^+)\right\}$.

Then, $G = (A_L, \mapsto_L)$ is the directly-follows graph (DFG) for L.

We employ the shorthand notation $a \in G$ to denote that activity a is a node in G. An arc $(a,b) \in \mapsto_L$ from activity a to activity b indicates that in at least one $t \in L$ b directly follows a. A path in G between some a and b corresponds to $(a,b) \in \mapsto_L^+$. Later on, we remove the subscript $_L$ from A_L and $\mapsto_L$ because we consider subgraphs $G_i = (A_i, \mapsto_i)$ of G such that $A_i \subseteq A_L$ and $\mapsto_i \subseteq \mapsto_L$, for which A_i and $\mapsto_i$ no longer correspond to L.

4.2. Splitting the directly-follows graph

The function SPLIT in line 4 of Algorithm 1 builds a process tree T by recursively splitting G using the cuts corresponding to the control flow operators from Definition 3. For G consisting of a single activity, it returns the base case of the degenerated process tree of that activity in lines 5–6. Otherwise, it attempts to apply the cuts (see below) in lines 7–8. The first-found cut splits G and the corresponding control flow operator becomes the root of the subtree. Then, SPLIT calls recursively itself for subgraphs. If no cut applies, it returns in line 9 the fallback model that allows any sequence of the activities in G.

Definition 7. An $\odot$*-cut* of $G = (A, \mapsto)$, where $\odot \in \{\times, \rightarrow, \wedge, \circlearrowleft\}$, divides G into subgraphs $G_1 = (A_1, \mapsto_1), G_2 = (A_2, \mapsto_2), \ldots, G_n = (A_n, \mapsto_n)$ such that $n \geq 2$, $\forall_{i=1}^n A_i \neq \emptyset$, $A = \bigcup_{i=1}^n A_i$, and $\mapsto = \bigcup_{i=1}^n \mapsto_i$. In the below definitions, the domain of the iterator variables i and j is $\{1, 2, \ldots, n\}$.

- The exclusive choice $\times$-cut divides G into connected components, i.e., $\forall_{i \neq j} \forall_{(a,b) \in A_i \times A_j} (a,b) \notin \mapsto$.
- The sequential $\rightarrow$-cut divides G into unions of strongly connected components such that the activities from different components G_i and G_j are in the transitive closure of the directly-follows relation iff $i < j$, i.e., $\forall_{i<j} \forall_{(a,b) \in A_i \times A_j} (a,b) \in \mapsto^+ \wedge (b,a) \notin \mapsto^+$.
- The parallel $\wedge$-cut divides G such that each subgraph contains a start and an end activity of G, and the directly-follows relation exists for all pairs of activities from different subgraphs, i.e., $\forall_i A_i^s \neq \emptyset \wedge A_i^e \neq \emptyset$ and $\forall_{i \neq j} \forall_{(a,b) \in A_i \times A_j} (a,b) \in \mapsto$.
- The loop $\circlearrowleft$-cut divides G such that G_1 consists of all start and all end activities of G, only the end activity $a \in A^e$ may be the predecessor in the directly-follows relation with activities from $G_{\geq 2}$, only the start activity $a \in A^s$ may be the successor in the directly-follows relation with activities from $G_{\geq 2}$, activities from different $G_{\geq 2}$ must not be in the directly-follows relation, if the end activity $a \in A^e$ is in the directly-follows relation with activity $b \in G_{\geq 2}$ then all end activities must be in the directly-follows relation with b, and accordingly for the start activities, i.e., $A^s \cup A^e \subseteq A_1$ and $\{a \in A_1 : \exists_{i \geq 2} \exists_{b \in A_i} (a,b) \in \mapsto\} \subseteq A^e$ and $\{a \in A_1 : \exists_{i \geq 2} \exists_{b \in A_i} (b,a) \in \mapsto\} \subseteq A^s$ and $\forall_{i,j \geq 2; i \neq j} \forall_{(a,b) \in A_i \times A_j} (a,b) \notin \mapsto$ and $\forall_{i \geq 2} \forall_{b \in G_i} \exists_{a \in A^e} (a,b) \in \mapsto \implies \forall_{a' \in A^e} (a',b) \in \mapsto$ and $\forall_{i \geq 2} \forall_{b \in G_i} \exists_{a \in A^s} (b,a) \in \mapsto \implies \forall_{a' \in A^s} (b,a') \in \mapsto$.

The cuts correspond one-to-one to the control flow operators from Definition 3. The $\odot$-cut divides G into subgraphs G_i such that the language $\mathscr{L}(T)$ of the process tree $T = \odot(T_1, \ldots, T_n)$ is the respective function from Definition 3 of the languages $\mathscr{L}(T_i)$ of the process trees T_i produced by recursively splitting G_i.

To efficiently find cuts, we use dedicated graph algorithms. We identify the connected components for $\times$-cut using the *Flood fill* algorithm in $O(|A| + |\mapsto|)$ time. We find the strongly-connected components for $\rightarrow$-cut using Tarjan's algorithm [47] with Nuutila's extension [48] in $O(|A| + |\mapsto|)$ time. The $\wedge$-cut can be found by dropping all bidirectional arcs, inserting new bidirectional arcs between every pair of unconnected activities, and finding strongly connected components, like above. The $\circlearrowleft$-cut is constructed by first assigning all start and end activities to G_1, and then finding connected components with extra constraints from the definition of $\circlearrowleft$-cut in $O(|A| + |\mapsto|)$ time.

IMd conducts the cuts in the order of $\times, \rightarrow, \wedge, \circlearrowleft$ allowing the reuse of the partial computation results when attempting to apply successive cuts.

5. UPDATE OF PROCESS TREES

This section introduces Continuous Inductive Miner (CIM), an algorithm that extends IMd [8] in three directions:

- Update of process trees as posed in Definition 5,
- Handling of base cases using statistics,
- The perfect fitting to the event log.

CIM holds the guarantees of IMd: soundness and single-pass processing of the event log, and hybridizes them with the guarantee of IM [5] of producing perfect fit process trees. Although primarily designed for update scenario, CIM applies to the batch discovery problem from Definition 4 too, given the input of the empty process tree.

Algorithm 2 shows CIM. The red lines come from Algorithm 1, and the black lines are the novel contribution of this study.

Algorithm 2. Continuous Inductive Miner (CIM); $\text{REPLACE}(T, T_1, T_2)$ replaces the (indirect) subtree T_1 in T with T_2; lines 5, 19, 25, 28-31 come from IMd

1: **function** CONTINUOUSINDUCTIVEMINER(T, L^+, L^-)
2: $\quad \Delta A^+, \Delta A^-, \Delta A^s, \Delta A^e, \Delta\mapsto_{L^+}, \Delta\mapsto_{L^-} \leftarrow \text{DIFF}(T, L^+, L^-)$
3: $\quad G \leftarrow (A_{L \cup L^+ \setminus L^-}, \mapsto_{L \cup L^+ \setminus L^-})$ ▷ Update DFG
4: $\quad$ **if** $\Delta A^+ \neq \emptyset \vee \Delta A^- \neq \emptyset \vee \Delta A^s \neq \emptyset \vee \Delta A^e \neq \emptyset$ **then**
5: $\quad\quad$ **return** SPLIT(G) ▷ Rebuild the entire tree
6: $\quad$ **if** $\Delta\mapsto_{L^+} \neq \emptyset \vee \Delta\mapsto_{L^-} \neq \emptyset$ **then**
7: $\quad\quad \Delta A_{\mapsto} \leftarrow \{a, b \in \Delta\mapsto_{L^+} \cup \Delta\mapsto_{L^-}\}$
8: $\quad\quad \Delta T, \Delta G \leftarrow$ GETMINCOMMONSUBTREE$(T, \Delta A_{\mapsto})$
9: $\quad\quad$ **return** REPLACE$(T, \Delta T, \text{SPLIT}(\Delta G))$ ▷Rebuild subtree
10: $\quad$ **return** T ▷ No changes
11: **function** DIFF(T, L^+, L^-)
12: $\quad \Delta A^+ \leftarrow A_{L^+} \setminus A_L$
13: $\quad \Delta A^- \leftarrow A_{L^-} \setminus A_{L \cup L^+ \setminus L^-}$
14: $\quad \Delta A^s \leftarrow (A^s_{L^+} \setminus A^s_L) \cup (A^s_{L^-} \setminus A^s_{L \cup L^+ \setminus L^-})$
15: $\quad \Delta A^e \leftarrow (A^e_{L^+} \setminus A^e_L) \cup (A^e_{L^-} \setminus A^e_{L \cup L^+ \setminus L^-})$
16: $\quad \Delta\mapsto_{L^+} \leftarrow \mapsto_{L^+} \setminus \mapsto_L$
17: $\quad \Delta\mapsto_{L^-} \leftarrow \mapsto_{L^-} \setminus \mapsto_{L \cup L^+ \setminus L^-}$
18: $\quad$ **return** $\Delta A^+, \Delta A^-, \Delta A^s, \Delta A^e, \Delta\mapsto_{L^+}, \Delta\mapsto_{L^-}$
19: **function** SPLIT(G)
20: $\quad$ **if** $|\{a \in A\}| = 1 : A \in G$ **then**
21: $\quad\quad$ **if** $ds(a) \geq 1 \wedge gs(G) \geq gs(G^{\triangle})$ **then return** $\circlearrowleft(a, \tau)$
22: $\quad\quad$ **if** $ds(a) \geq 1 \wedge gs(G) < gs(G^{\triangle})$ **then return** $\circlearrowleft(\tau, a)$
23: $\quad\quad$ **if** $T^{\triangle} \in \{\rightarrow, \wedge\} \wedge gs(G) < gs(G^{\triangle})$ **then return** $\times(a, \tau)$
24: $\quad\quad$ **return** a
25: $\quad$ **if** $\times$-cut applies to G **then**
26: $\quad\quad$ **if** $\sum_{i=1}^{n} gs(G_i) < gs(G)$ **then**
27: $\quad\quad\quad$ **return** $\times(\tau, \text{SPLIT}(G_1), \text{SPLIT}(G_2), \ldots, \text{SPLIT}(G_n))$
28: $\quad\quad$ **return** $\times(\text{SPLIT}(G_1), \text{SPLIT}(G_2), \ldots, \text{SPLIT}(G_n))$
29: $\quad$ **if** $\odot$-cut applies to G, where $\odot \in [\rightarrow, \wedge, \circlearrowleft]$ **then**
30: $\quad\quad$ **return** $\odot(\text{SPLIT}(G_1), \text{SPLIT}(G_2), \ldots, \text{SPLIT}(G_n))$
31: $\quad$ **return** $\circlearrowleft(\tau, a_1, a_2, \ldots, a_n)$, where $a_i \in G$
32: **function** GETMINCOMMONSUBTREE$(T, \Delta A_{\mapsto})$
33: $\quad$ **if** $\exists_{T_i \in T} \Delta A_{\mapsto} \subseteq T_i$ **then**
34: $\quad\quad$ **return** GETMINCOMMONSUBTREE(T_i)
35: $\quad$ **return** T

Given a possibly empty process tree T and differential logs L^+ and L^- it operates in three steps. First, in line 2 it calculates the differential sets: ΔA^+ and ΔA^- of activities to add to and remove from T, respectively, ΔA^s and ΔA^e of activities that become or cease to be the start and end, respectively, $\Delta\mapsto_{L^+}$ and $\Delta\mapsto_{L^-}$ of directly-follows relations introduced by L^+ and removed by L^-, respectively. Function DIFF in line 11 shows equations for the above-mentioned symbols. Second, in line 3 it updates DFG corresponding to T. Technically, it updates the statistics from Definition 8 and then applies changes to DFG as described in Section 5.1. Finally, in lines 4-10 it picks one out of three options:

- If new activities are added, or old activities are removed, or start or end activities change, then rebuild the entire tree in line 5.
- If directly-follows relation changes for at least one pair of activities, identify the set of affected activities in line 7, find the minimal common subtree for them in line 8, and update that subtree in line 9.
- Otherwise, return T intact in line 10.

5.1. Statistics

DFG from Definition 6 is a form of lossy compression because duplicate evidence of the directly-follows relation in L does not influence DFG. Hence, DFG is typically much smaller than L but loses information on decision points and repetitions, hence the paths in DFG may represent traces not included in L.

The lack of this information prevents an efficient update of DFG, since from the differential event log L^- of the traces leaving the time window, it is unknown whether a particular directly-follows relation in L^- remains in the time window or is to be removed. This also prevents proper calculation of the base cases for the process tree, e.g., optionality and repetitions (see below).

To prevent information loss, we introduce statistics that compactly store data needed to update T and properly handle base cases.

Definition 8. The activity support $as(a) = |\{t \in L : a \in t\}|$ is the number of traces in L involving activity a at least once. The directly-follows support $dfs(a,b) = |\{[e_1, e_2, \ldots e_n] \in L : \exists_{i=1}^{n-1} e_i = a \wedge e_{i+1} = b\}|$ is the number of traces in L, where activity b directly follows a. The duplicate support $ds(a) = dfs(a,a)$ is the number of traces in L, where activity a directly follows itself. The graph support $gs(G) = |\bigcup_{a \in G}\{t \in L : a \in t\}|$ is the total number of traces involving the activities from G.

Both G and statistics can be calculated in linear time by reading every event $e \in L$ exactly once. Statistics for L can be updated in the linear time given differential event logs L^+ and L^- as defined in Definition 5 by incrementing them using L^+ and decrementing them using L^-. An activity a belongs to G after update if $as(a) > 0$, and an arc (a,b) exists if $dfs(a,b) > 0$. Note that we store both G and the statistics together with T for an efficient update.

5.2. Augmented splitting of the directly-follows graph

The SPLIT function in line 19 of Algorithm 2 extends the SPLIT function of Algorithm 1 with the use of statistics to properly handle base cases and the optionality of branches. In lines 20–24 it handles DFG consisting of a single activity and corresponding to one of the four base cases by ending recursion with the base case-specific subtree. Otherwise, it attempts in lines 25–30 to apply the same cuts as in Algorithm 1. However, for the $\times$-cut it detects in lines 25–27 the special case, where the choice of a subtree is optional. Similarly to IMd, if no cut applies, it returns in line 31 the fallback model of all activities in DFG.

We consider four base cases of a business process of a single activity each shown in Table 1. The first column provides the interpretation of the process tree in the second column. The middle column shows the graphical representation of the corresponding DFG. The last two columns show the conditions to detect each case. $ds(a)$ determines whether a runs in loop ($ds(a) \geq 1$) or not ($ds(a) = 0$). $gs(G)$ determines whether a is optional ($gs(G) < gs(G^{\triangle})$) or not ($gs(G) = gs(G^{\triangle})$), where $G^{\triangle}$ is the *parent* graph of G, i.e., the graph before split; if G has no parent, then $gs(G^{\triangle}) = |L|$.

Table 1
Base cases of process trees

Base case	T	G	$ds(a)$	$gs(G)$
Certain activity	a	$\rightarrow a \rightarrow$	0	$= gs(G^{\triangle})$
Optional activity	$\times(a,\tau)$	$\rightarrow a \rightarrow$ (with bypass arc)	0	$< gs(G^{\triangle})$
Loop at least once	$\circlearrowleft(a,\tau)$	$\rightarrow a \rightarrow$ (with self-loop)	≥ 1	$= gs(G^{\triangle})$
Loop at least zero	$\circlearrowleft(\tau,a)$	$\rightarrow a \rightarrow$ (with bypass arc and self-loop)	≥ 1	$< gs(G^{\triangle})$

SPLIT produces process tree a that executes activity a exactly once when a never repeats and runs in all traces involving $G^{\triangle}$. If the latter condition does not hold, a is optional and SPLIT produces $\times(a,\tau)$ running either a or the silent activity τ. SPLIT produces $\circlearrowleft(a,\tau)$ if a repeats in at least one trace and is certain, or $\circlearrowleft(\tau,a)$ if optional.

5.3. Minimal common subtree

CIM updates only the minimal common subtree of T in which all changes occur and leaves the remaining parts of T intact.

Definition 9. The *minimal common subtree* T' of T given the set of activities A is the smallest subtree of T containing all activities from A. The shorthand notation $A \subseteq T$ denotes that T contains all activities from A.

CIM finds the minimal common subtree using the function in line 32 of Algorithm 2. Given the process tree T and the set of the activities affected by the change $\Delta A_{\mapsto}$, it conducts the breadth-first search starting from the root node. First, it looks in line 33 for a direct subtree containing all activities from $\Delta A_{\mapsto}$, and recursively calls itself for this subtree in line 34 if found. Otherwise, it returns T in line 35.

The containment test in line 33 runs in $O(|\Delta A_{\mapsto}|)$ time for each T_i, thanks to storing the activities in T_i using a hash set. Thus, SPLIT executes in $O(|\Delta A_{\mapsto}| \cdot |T|)$ time, where $|T|$ is the total number of nodes in T.

5.4. Guarantee for the perfect fitting

The original IMd does not guarantee the perfect fit for the event log. Consider an event log $L = \{[a,b,c],[a,b,b,c]\}$ for which IMd produces the DFG

$$G: \quad a \longrightarrow b \longrightarrow c. \quad (\text{with a self-loop on } b)$$

IMd applies the $\rightarrow$-cut to G resulting in subgraphs

$$G_1: \quad a \rightarrow \qquad G_2: \quad \rightarrow b \rightarrow \ (\text{with a self-loop on } b) \qquad G_3: \quad \rightarrow c.$$

For all G_1, G_2, G_3 the base case of producing the degenerated process tree in line 6 of Algorithm 1 applies, resulting in the process tree $T_1 = \rightarrow (a,b,c)$. T_1 reproduces the trace $[a,b,c]$ but must not reproduce the trace $[a,b,b,c]$. This is because IMd ignores the self-loop over activity b.

In contrast, CIM guarantees the perfect fit. It is aware of the self-loops and picks from Table 1 the base case that matches the graph structure and the statistics of activity repetitions. For L, CIM calculates $ds(b) = 1$, $gs(G) = 2$, $gs(G_2) = 2$. These values match the condition for the loop at least once base case, and thus CIM transforms G_2 into $\circlearrowleft(b,\tau)$ and produces $T_2 = \rightarrow (a, \circlearrowleft(b,\tau), c)$.

The base cases in Table 1 cover all possibilities of executing a single activity. Hence, we provide the proofs of the perfect fit for each base case separately.

Proof. Let G denote the subgraph containing a single activity (the base case) and let $G^{\triangle}$ denote the parent graph of G. Let $L_G = \bigcup_{a \in G}\{t \in L : a \in t\}$ be the subset of an event log L containing the traces involving the activities from G.

1. (Certain activity) Assume that $L_{G^{\triangle}} = \{t : t = [\ldots,a,\ldots]\}$ and $ds(a) \neq 0$ or $gs(G_b) \neq gs(G^{\triangle})$. By Definition 8 $ds(a) = 0$ and $gs(G) = |\{t \in L_{G^{\triangle}} : a \in t\}| = |L_{G^{\triangle}}| = gs(G\triangle)$ that contradicts with the assumption.
2. (Optional activity) Assume that $L_{G^{\triangle}} = \{t : a \notin t\} \cup \{t : t = [\ldots,a,\ldots]\}$ and $ds(a) \neq 0$ or $gs(G) \geq gs(G^{\triangle})$. By Definition 8 $ds(a) = 0$ and $gs(G) = |\{t \in L_{G^{\triangle}} : a \in t\}| < |L_{G^{\triangle}}| = gs(G^{\triangle})$ that contradicts with the assumption.
3. (Loop at least once) Assume that $L_{G^{\triangle}} = \{t : t = [\ldots,a,\ldots]\} \cup \{t : t = [\ldots,a,a,\ldots]\}$ and $ds(a) < 1$ or $gs(G) \neq gs(G^{\triangle})$. By Definition 8 $ds(a) \geq 1$ and $gs(G) = |\{t \in L_{G^{\triangle}} : a \in t\}| = L_{G^{\triangle}} = gs(G^{\triangle})$ that contradicts with the assumption.
4. (Loop at least zero) Assume that $L_{G^{\triangle}} = \{t : a \notin t\} \cup \{t : t = [\ldots,a,a,\ldots]\}$ and $ds(a) < 1$ or $gs(G) \geq gs(G^{\triangle})$. By Definition 8 $ds(a) \geq 1$ and $gs(G) = |\{t \in L_{G^{\triangle}} : a \in t\}| < gs(G^{\triangle})$ that contradicts with the assumption.

The proof of the perfect fit of the cuts comes directly from Definitions 3 and 7. □

6. EXPERIMENT

6.1. Preliminaries

We verify the main hypothesis from Section 1.2 by answering in Sections 6.2–6.6 several detailed research questions:

- Do CIM, IM, and IMd produce the same process trees?
- How good trees does CIM create compared to IM and IMd?
- Does it pay off to update the process tree?
- How often does CIM rebuild the entire process tree?
- How do the produced process trees look like?

We choose 21 real-world event logs from the repository [56] using the steps:

1. Omit the logs with the words *synthetic*, *artificial*, *benchmark*.
2. Omit the logs not compliant with the XES standard.
3. Select the logs with at least 200 and at most 10000 traces.

We focus on real-world event logs to keep the experiment nontrivial and realistic. The chosen event logs consist of typical process structures, e.g., sequences, decisions, parallelism, optionality, and loops. They also consist of noise and thus make the achieved results better reflecting the algorithm performance

in real-world use cases. We apply the upper bound from step 3 to keep the computational cost at bay. Table 2 summarizes the event logs. These refer to the processes of handling incidents in an IT department [49], building permit applications [50], environmental permit applications [51, 53], medical procedures [52, 54], and software execution [55].

Table 2
Event logs; numbers of traces and unique activities

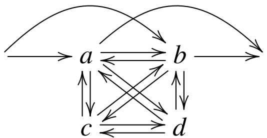
Event log	# traces	# activities
BPIC13_cp [49]	1487	4
BPIC13_i [49]	7554	4
BPIC15_1 [50]	1199	398
BPIC15_1f [50]	902	70
BPIC15_2 [50]	832	410
BPIC15_2f [50]	681	82
BPIC15_3 [50]	1409	383
BPIC15_3f [50]	1369	62
BPIC15_4 [50]	1053	356
BPIC15_4f [50]	860	65
BPIC15_5 [50]	1156	389
BPIC15_5f [50]	975	74
CoSeLoG_WABO_1 [51]	937	381
CoSeLoG_WABO_2 [51]	645	376
CoSeLoG_WABO_3 [51]	1087	369
CoSeLoG_WABO_4 [51]	787	331
CoSeLoG_WABO_5 [51]	892	350
Hospital_log [52]	1143	624
Receipt_phase_of_an_enviro... [53]	1434	27
Sepsis_Cases [54]	1050	16
nasa-cev-complete-splitted [55]	2566	47

To pose the instances of the update problem, we use a time window consisting of $n \in \{10, 20, 30, 40, 50, 75, 100, 150, 200\}$ traces. Initially, we set the window at the beginning of an event log and repeatedly shift by one trace, producing thus differential event logs L^+ and L^- as in Definition 5. In this way, we simulate the real-world use case, where one runs CIM as soon as a new trace is available.

For the comparison with the batch predecessors, each time CIM runs using the differential event logs when the time window shifts, the predecessors run using the contents of the entire time window. We collect the same statistics for all algorithms.

6.2. Do CIM, IM, and IMd produce the same process trees?

No. The process trees produced by CIM and IMd differ due to the different handling of base cases and optionality in $\times$-cut. The process trees by CIM and IM sometimes differ too because CIM updates only the part of the process tree that is inconsistent with L^+ and L^-. The resulting process tree depends on the history of updates. In contrast, IM builds the process tree by applying the cuts in a predefined order, which may be different than in the existing process tree. For instance, for DFG:



both $\wedge$-cut and $\circlearrowleft$-cut apply. IM does the former, producing $\wedge(\circlearrowleft(a,b), \circlearrowleft(c,d))$. However, if the existing tree is already rooted with the $\circlearrowleft$ operator, CIM builds $\circlearrowleft(\wedge(a,b), \wedge(c,d))$.

6.3. How good trees does CIM create compared to IM and IMd?

CIM and IM produce process trees perfectly fitting the training time window. IMd fits slightly worse due to the simplified handling of base cases. The differences in the trees produced by these algorithms influence generalization and precision. We assess generalization using $f(T,L)$ and precision using $p(T,L)$ from Definition 4 calculated on the *test time window* of the same size and shifted forward w.r.t. the training time window by the window size. Table 3 shows the means and standard deviations of test fitness and precision over training window positions for different window sizes. The last row shows the p-values of the Wilcoxon signed-rank test [57] for pairwise comparisons of the best algorithm and the others at the family-wise significance level $\alpha = 0.05$. The Holm-Bonferroni method [58] shows that CIM achieves test fitness better than IMd and equal IM for all window sizes. The existing differences between CIM and IM are very rare and small. IMd scores significantly better precision than CIM and IM for window sizes ≤ 100. For large windows of 150 and 200 traces, the differences between the algorithms are insignificant.

6.4. Does it pay off to update the process tree?

Yes. Updating a process tree based on differential event logs using CIM is noticeably faster than building the process tree based on the entire time window using IM and IMd when the time window is large enough. Figure 2 shows the mean over 15 runs and window positions of the runtime [ms] for different window sizes, obtained using Core i7-8700 CPU, macOS 11.6, and OpenJDK 17. Even if smaller than the entire time window, the differential event logs require extra processing to detect incompatibilities with the existing process tree. This extra processing turns out beneficial w.r.t. discovering the entire process tree from scratch for time windows larger than 10–75 traces, depending on the problem instance.

The runtimes in Fig. 2 vary from milliseconds to seconds and refer to a single-window shift by a single trace. The total runtimes are the product of these values and the total numbers of shifts (which roughly equal the total numbers of traces in Table 2). The total runtimes amount to seconds to minutes for CIM, and minutes to an hour for IM and IMd.

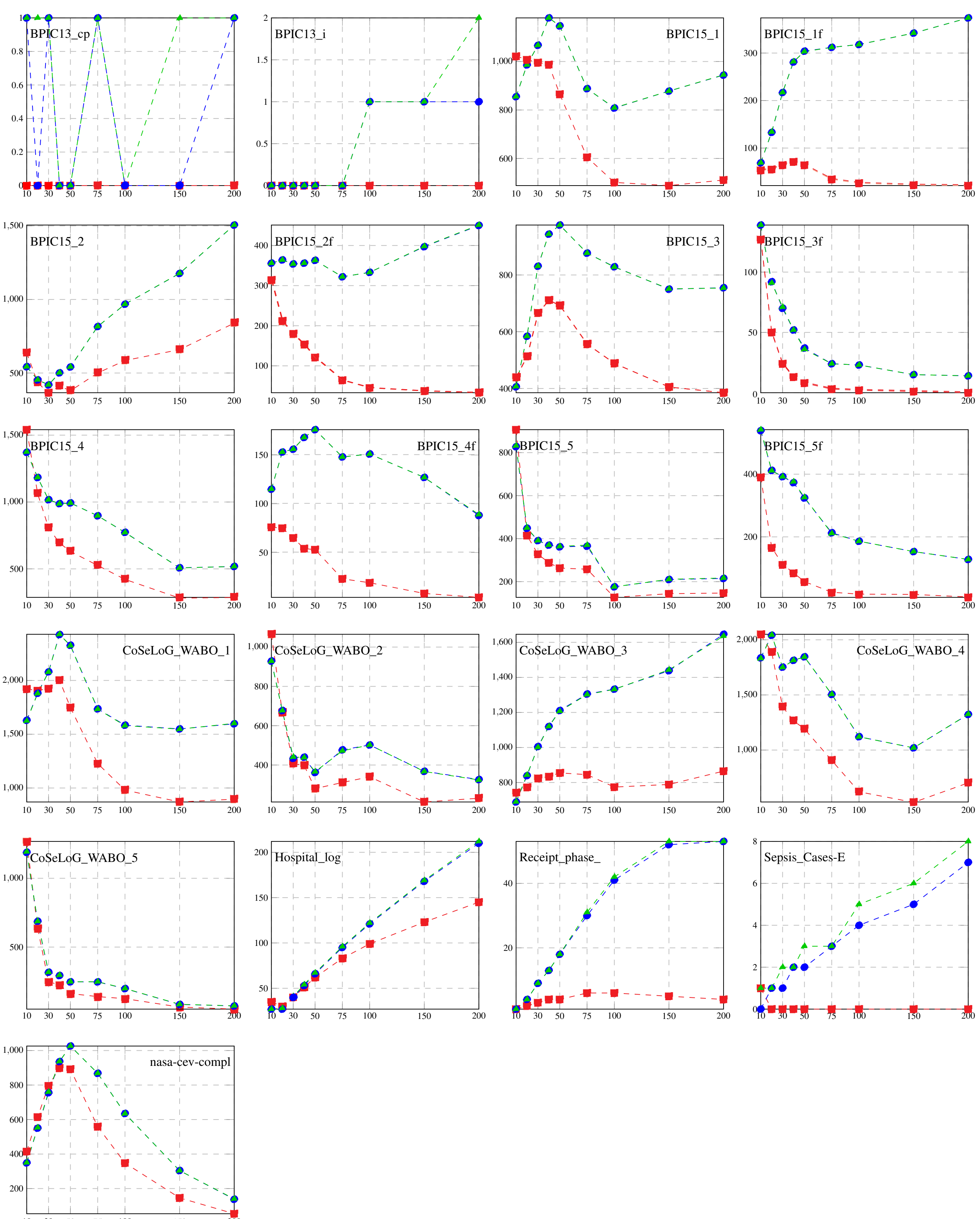


Fig. 2. Mean over runs and window positions of runtimes [ms] of CIM (red square), IM (blue circle), and IMd (green triangle) depending on window size

Table 3

Mean (A) fitness and (B) precision on test time window w.r.t. window size; the vertical bars reflect standard deviation, where the bar of cell height reflects 0.5; green marks a larger value than the others, red smaller, white no difference with at least one other value; the p-values of the Wilcoxon signed-rank test for the best algorithm vs others; the underlining marks significant differences using the Holm-Bonferroni method for $\alpha = 0.05$

(A) Window size	10			20			30			40			50			75			100			150			200		
Event log	CIM	IM	IMd	CIM	IM	IMd	CIM	IM	IMd	CIM	IM	IMd	CIM	IM	IMd	CIM	IM	IMd	CIM	IM	IMd	CIM	IM	IMd	CIM	IM	IMd
BPIC13_cp	0.993	0.993	0.893	0.997	0.997	0.941	0.998	0.998	0.961	0.999	0.999	0.966	0.999	0.999	0.971	0.999	0.999	0.987	1.000	1.000	0.997	1.000	1.000	1.000	1.000	1.000	1.000
BPIC13_i	0.996	0.996	0.884	0.997	0.997	0.926	0.998	0.998	0.947	0.999	0.999	0.960	0.999	0.999	0.967	0.999	0.999	0.974	0.999	0.999	0.973	0.999	0.999	0.970	0.999	0.999	0.971
BPIC15_1	0.993	0.993	0.983	0.996	0.996	0.988	0.997	0.997	0.990	0.997	0.997	0.990	0.997	0.997	0.990	0.997	0.997	0.990	0.998	0.998	0.990	0.998	0.998	0.990	0.998	0.998	0.990
BPIC15_1f	0.971	0.971	0.905	0.974	0.974	0.890	0.976	0.977	0.889	0.979	0.979	0.890	0.979	0.979	0.892	0.983	0.983	0.890	0.984	0.984	0.881	0.983	0.983	0.865	0.980	0.980	0.857
BPIC15_2	0.996	0.996	0.988	0.998	0.998	0.993	0.998	0.998	0.994	0.998	0.998	0.994	0.999	0.998	0.994	0.999	0.999	0.993	0.998	0.998	0.992	0.998	0.998	0.991	0.997	0.997	0.991
BPIC15_2f	0.984	0.984	0.943	0.993	0.993	0.957	0.995	0.995	0.965	0.996	0.996	0.967	0.996	0.996	0.969	0.996	0.996	0.968	0.996	0.996	0.970	0.996	0.997	0.981	0.997	0.997	0.983
BPIC15_3	0.995	0.995	0.986	0.997	0.997	0.989	0.996	0.996	0.989	0.996	0.996	0.989	0.996	0.996	0.989	0.998	0.998	0.990	0.998	0.998	0.990	0.999	0.999	0.992	0.999	0.999	0.993
BPIC15_3f	0.981	0.981	0.953	0.989	0.989	0.969	0.991	0.991	0.974	0.993	0.993	0.978	0.995	0.995	0.981	0.998	0.998	0.987	0.998	0.998	0.990	0.999	0.999	0.991	0.999	0.999	0.995
BPIC15_4	0.993	0.993	0.982	0.996	0.996	0.989	0.997	0.997	0.990	0.996	0.996	0.990	0.996	0.996	0.991	0.997	0.997	0.991	0.997	0.997	0.992	0.999	0.999	0.993	0.998	0.998	0.994
BPIC15_4f	0.966	0.966	0.919	0.986	0.986	0.944	0.990	0.990	0.953	0.992	0.993	0.961	0.993	0.993	0.965	0.995	0.995	0.968	0.997	0.997	0.973	0.999	0.999	0.981	0.999	0.999	0.983
BPIC15_5	0.997	0.997	0.990	0.998	0.998	0.993	0.999	0.999	0.994	0.999	0.999	0.994	0.999	0.999	0.994	0.999	0.999	0.995	1.000	1.000	0.995	1.000	1.000	0.995	1.000	1.000	0.995
BPIC15_5f	0.985	0.985	0.951	0.988	0.988	0.956	0.991	0.991	0.960	0.993	0.993	0.960	0.994	0.994	0.959	0.995	0.995	0.959	0.995	0.995	0.958	0.995	0.995	0.953	0.992	0.992	0.950
CoSeLoG_WABO_1	0.993	0.993	0.982	0.996	0.996	0.988	0.997	0.997	0.990	0.997	0.997	0.990	0.998	0.998	0.990	0.998	0.998	0.990	0.999	0.999	0.990	0.999	0.999	0.989	0.998	0.998	0.988
CoSeLoG_WABO_2	0.996	0.996	0.989	0.999	0.999	0.994	0.999	0.999	0.995	0.999	0.999	0.995	0.999	0.999	0.994	0.999	0.999	0.994	0.998	0.998	0.993	0.999	0.999	0.992	0.998	0.998	0.992
CoSeLoG_WABO_3	0.995	0.995	0.985	0.997	0.997	0.989	0.997	0.997	0.989	0.996	0.996	0.990	0.997	0.997	0.990	0.998	0.998	0.990	0.998	0.998	0.990	0.998	0.998	0.991	0.999	0.999	0.992
CoSeLoG_WABO_4	0.993	0.993	0.980	0.996	0.996	0.988	0.997	0.997	0.990	0.997	0.997	0.990	0.997	0.997	0.991	0.997	0.997	0.992	0.997	0.997	0.992	0.998	0.998	0.993	0.998	0.998	0.992
CoSeLoG_WABO_5	0.996	0.996	0.989	0.998	0.998	0.992	0.998	0.998	0.993	0.998	0.998	0.994	0.999	0.999	0.995	0.999	0.999	0.995	1.000	1.000	0.995	1.000	1.000	0.996	1.000	1.000	0.997
Hospital_log	1.000	1.000	0.998	1.000	1.000	0.997	1.000	1.000	0.997	1.000	1.000	0.998	1.000	1.000	0.998	1.000	1.000	0.998	1.000	1.000	0.998	1.000	1.000	0.998	1.000	1.000	0.998
Receipt_phase_	0.994	0.994	0.968	0.997	0.996	0.966	0.996	0.996	0.960	0.996	0.996	0.956	0.996	0.996	0.955	0.995	0.995	0.951	0.994	0.994	0.946	0.992	0.992	0.940	0.995	0.995	0.939
Sepsis_Cases-E	0.998	0.998	0.980	0.999	0.999	0.982	0.999	0.999	0.983	0.999	0.999	0.982	0.999	0.999	0.982	1.000	1.000	0.974	1.000	1.000	0.964	1.000	1.000	0.948	1.000	1.000	0.931
nasa-cev-compl	0.992	0.992	0.911	0.994	0.994	0.926	0.995	0.995	0.938	0.995	0.995	0.949	0.995	0.995	0.956	0.996	0.996	0.965	0.997	0.997	0.969	0.998	0.998	0.977	0.999	0.999	0.982
Mean	0.992	0.992	0.938	0.995	0.995	0.956	0.996	0.996	0.964	0.996	0.996	0.969	0.997	0.997	0.972	0.997	0.997	0.976	0.998	0.998	0.976	0.998	0.998	0.975	0.998	0.998	0.975
p-value		0.053	0.000		0.286	0.000	0.093		0.000	0.267		0.000		0.687	0.000		0.071	0.000	0.041		0.000	0.173		0.000	0.152		0.000

(B) Window size	10			20			30			40			50			75			100			150			200		
Event log	CIM	IM	IMd	CIM	IM	IMd	CIM	IM	IMd	CIM	IM	IMd	CIM	IM	IMd	CIM	IM	IMd	CIM	IM	IMd	CIM	IM	IMd	CIM	IM	IMd
BPIC13_cp	0.632	0.633	0.690	0.650	0.650	0.685	0.669	0.669	0.688	0.689	0.689	0.701	0.704	0.704	0.707	0.724	0.724	0.719	0.735	0.735	0.733	0.749	0.749	0.749	0.757	0.757	0.757
BPIC13_i	0.532	0.532	0.431	0.547	0.547	0.479	0.557	0.557	0.512	0.566	0.566	0.534	0.574	0.574	0.549	0.591	0.591	0.575	0.606	0.606	0.590	0.627	0.627	0.617	0.640	0.640	0.630
BPIC15_1	0.023	0.023	0.083	0.012	0.012	0.046	0.010	0.010	0.030	0.009	0.009	0.028	0.009	0.009	0.032	0.008	0.008	0.033	0.008	0.008	0.028	0.007	0.007	0.012	0.007	0.007	0.040
BPIC15_1f	0.582	0.582	0.801	0.452	0.452	0.756	0.382	0.381	0.728	0.322	0.322	0.713	0.284	0.284	0.700	0.236	0.235	0.700	0.213	0.212	0.715	0.144	0.144	0.739	0.103	0.099	0.768
BPIC15_2	0.017	0.017	0.057	0.011	0.011	0.024	0.009	0.009	0.010	0.008	0.008	0.009	0.008	0.008	0.007	0.007	0.007	0.006	0.007	0.007	0.005	0.006	0.006	0.005	0.005	0.005	0.005
BPIC15_2f	0.220	0.220	0.441	0.134	0.133	0.282	0.098	0.098	0.211	0.078	0.078	0.185	0.068	0.068	0.177	0.047	0.046	0.165	0.061	0.060	0.179	0.055	0.053	0.197	0.086	0.078	0.234
BPIC15_3	0.021	0.021	0.093	0.012	0.012	0.054	0.011	0.011	0.058	0.010	0.010	0.045	0.010	0.010	0.036	0.009	0.009	0.060	0.008	0.008	0.039	0.007	0.007	0.007	0.007	0.007	0.006
BPIC15_3f	0.219	0.219	0.385	0.140	0.139	0.257	0.108	0.107	0.209	0.086	0.086	0.167	0.065	0.065	0.135	0.046	0.046	0.082	0.038	0.038	0.068	0.029	0.029	0.054	0.027	0.027	0.034
BPIC15_4	0.025	0.026	0.087	0.013	0.013	0.041	0.011	0.011	0.026	0.010	0.010	0.017	0.009	0.009	0.012	0.009	0.009	0.009	0.008	0.008	0.008	0.007	0.007	0.007	0.007	0.007	0.007
BPIC15_4f	0.206	0.207	0.384	0.085	0.085	0.230	0.068	0.068	0.215	0.059	0.059	0.210	0.061	0.061	0.212	0.057	0.057	0.215	0.042	0.042	0.172	0.030	0.030	0.080	0.030	0.030	0.038
BPIC15_5	0.019	0.019	0.069	0.012	0.012	0.050	0.011	0.011	0.035	0.010	0.010	0.022	0.010	0.010	0.011	0.008	0.008	0.008	0.008	0.008	0.008	0.007	0.007	0.007	0.007	0.007	0.007
BPIC15_5f	0.369	0.369	0.547	0.276	0.277	0.461	0.237	0.237	0.387	0.221	0.221	0.349	0.198	0.198	0.326	0.140	0.140	0.286	0.109	0.109	0.267	0.071	0.071	0.241	0.102	0.102	0.298
CoSeLoG_WABO_1	0.026	0.026	0.097	0.012	0.012	0.065	0.010	0.010	0.040	0.009	0.009	0.035	0.009	0.009	0.040	0.008	0.008	0.047	0.008	0.008	0.040	0.007	0.007	0.018	0.007	0.007	0.056
CoSeLoG_WABO_2	0.014	0.014	0.042	0.010	0.010	0.018	0.009	0.009	0.009	0.008	0.008	0.009	0.008	0.008	0.007	0.007	0.007	0.006	0.006	0.006	0.005	0.006	0.006	0.004	0.005	0.005	0.004
CoSeLoG_WABO_3	0.023	0.023	0.114	0.013	0.013	0.057	0.011	0.011	0.063	0.010	0.010	0.051	0.010	0.010	0.046	0.009	0.009	0.042	0.008	0.008	0.034	0.008	0.008	0.007	0.007	0.007	0.007
CoSeLoG_WABO_4	0.029	0.029	0.098	0.014	0.014	0.056	0.011	0.011	0.035	0.010	0.010	0.019	0.009	0.009	0.010	0.009	0.009	0.009	0.008	0.008	0.008	0.007	0.007	0.007	0.007	0.007	0.007
CoSeLoG_WABO_5	0.022	0.022	0.086	0.013	0.013	0.063	0.011	0.011	0.046	0.011	0.011	0.028	0.010	0.010	0.012	0.009	0.009	0.009	0.008	0.008	0.008	0.008	0.008	0.008	0.007	0.007	0.007
Hospital_log	0.009	0.009	0.075	0.007	0.007	0.057	0.006	0.006	0.031	0.006	0.006	0.045	0.005	0.005	0.006	0.005	0.005	0.006	0.004	0.004	0.006	0.004	0.004	0.003	0.003	0.003	0.003
Receipt_phase_	0.266	0.266	0.322	0.217	0.217	0.248	0.201	0.201	0.227	0.187	0.187	0.212	0.172	0.173	0.197	0.152	0.152	0.173	0.142	0.142	0.160	0.126	0.127	0.141	0.117	0.117	0.128
Sepsis_Cases-E	0.135	0.135	0.140	0.132	0.132	0.134	0.131	0.131	0.132	0.133	0.134	0.135	0.137	0.137	0.137	0.142	0.142	0.150	0.146	0.146	0.165	0.152	0.152	0.186	0.156	0.156	0.204
nasa-cev-compl	0.302	0.311	0.391	0.242	0.249	0.360	0.205	0.208	0.337	0.180	0.182	0.316	0.165	0.165	0.288	0.147	0.144	0.238	0.143	0.139	0.184	0.140	0.134	0.125	0.133	0.131	0.080
Mean	0.281	0.282	0.320	0.259	0.260	0.296	0.254	0.254	0.292	0.252	0.252	0.290	0.251	0.251	0.289	0.254	0.254	0.292	0.262	0.261	0.293	0.277	0.276	0.300	0.297	0.296	0.317
p-value	0.000	0.000		0.000	0.000		0.000	0.000		0.000	0.000		0.000	0.000		0.002	0.002		0.002	0.002		0.093	0.079		0.091	0.091	

6.5. How often does CIM rebuild the entire process tree?

Less than half of the time and this percentile decreases with the size of the time window. Figure 3 shows the mean number of times CIM rebuilds the entire process tree, its part, or leaves the given process tree intact depending on window size. Large time windows are less likely to change significantly on shift than small ones. This property is efficiently used by CIM that rarely rebuilds the entire process tree given a large-enough window. The number of partial updates decreases with window size too, except for Hospital_log. It is problem-dependent whether partial updates or no updates at all is more common.

6.6. What do the produced process trees look like?

The size and the complexity of the process trees vary and depend on how well the language of the control flow operators reflects the event log. CIM typically produces the fallback model for some branches of the process tree.

For reference, Fig. 4 shows the process trees produced by CIM for the Sepsis_Cases event log at different window positions for the window size of 200 traces. The position value in the figure refers to the number of traces preceding the window. Initially, for positions 0 and 100, CIM produces well-structured process trees, with only one deep subtree containing the fall-

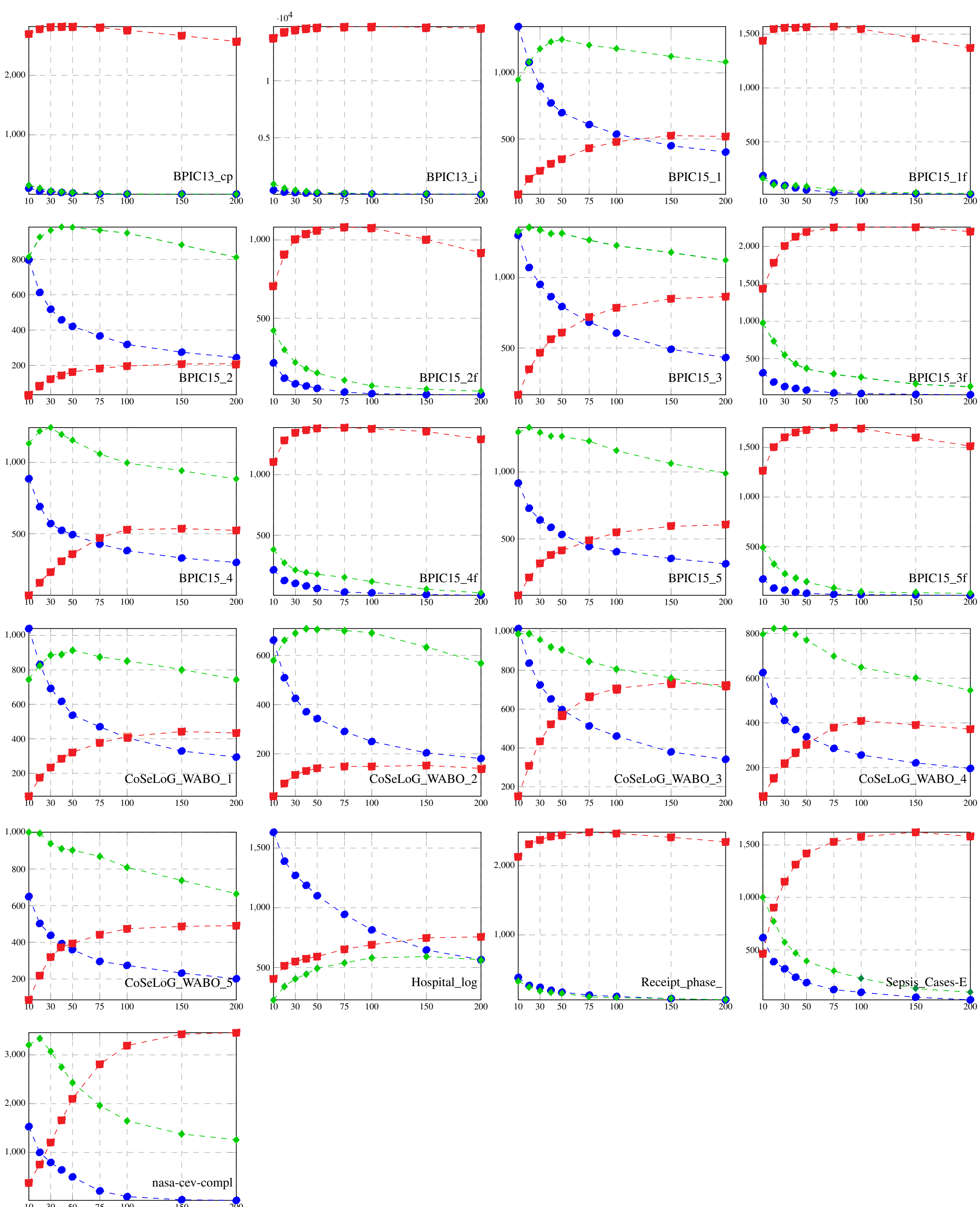


Fig. 3. Mean number of times CIM rebuilds the entire tree (blue circle), rebuilds a part of it (green triangle), and does not change the tree (red square) depending on window size

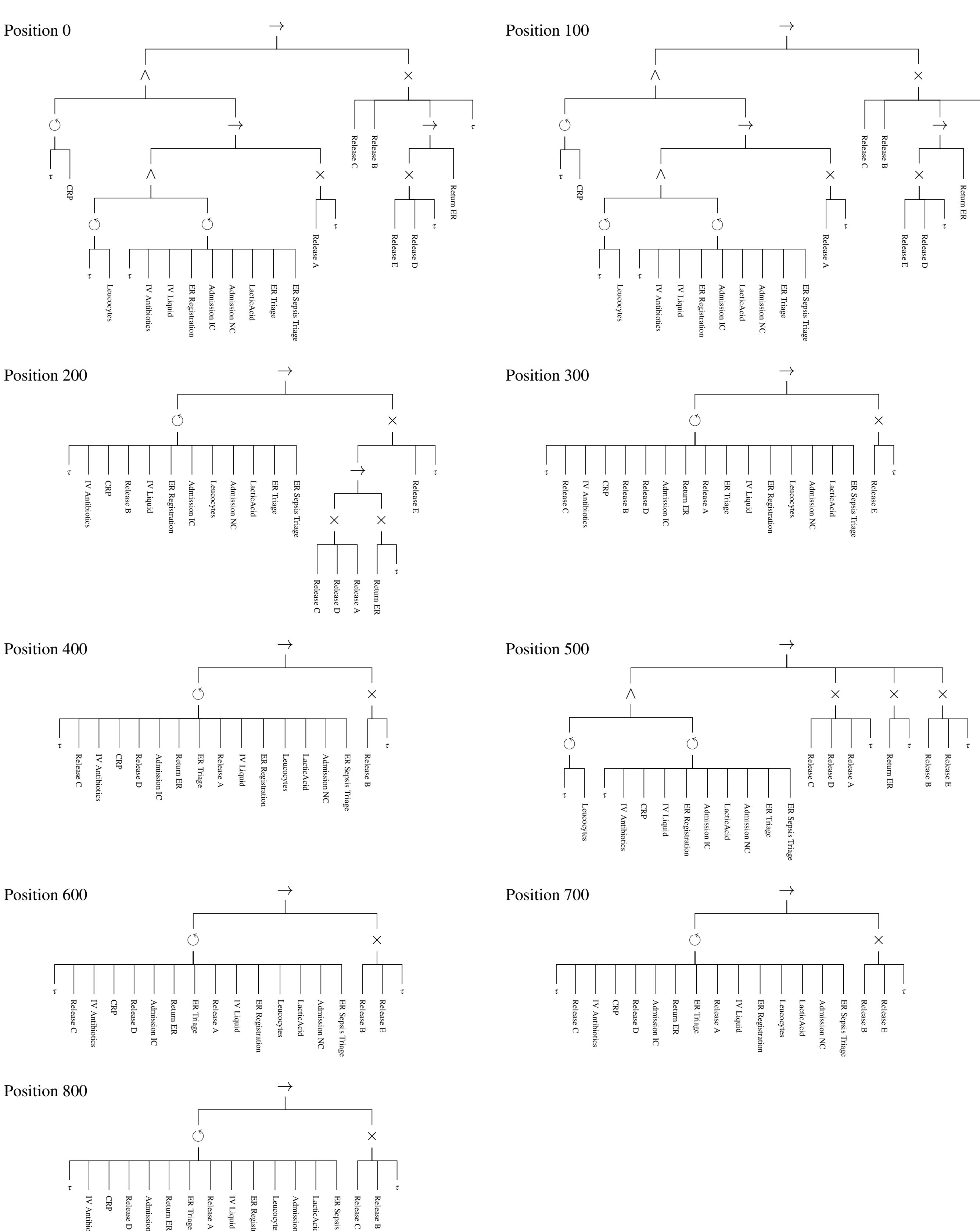


Fig. 4. The evolution of the process tree for the Sepsis_Cases event log; window position refers to the number of traces preceding the window; window size of 200

back model $\circlearrowleft$ (τ, IV Antibiotics, IV Liquid, ER Registration, Admission IC, Admission NC, LacticAcid, ER Triage, ER Sepsis Triage) indicating that CIM cannot find better order relation for these activities. As the window shifts to positions 200, 300, and 400, the fallback model embraces larger parts of the process trees. For position 500, CIM finds a better-structured process tree. For positions 600, 700, and 800 again the fallback model embraces larger parts of the process trees. Note that the results are stable in the sense that the process trees obtained for the windows sharing some content equal, e.g., for positions 0 and 100; 300 and 400; 600 and 700 and 800, respectively.

7. DISCUSSION

The problem of the update of process tree from Definition 5 is very general. The differential event logs L^+ and L^- can be constructed in many ways. The construction of L^+ and L^- depends on the specific use case and domain and lies beyond the scope of this work. For instance, L^+ may consist of newly observed traces in some period or in real-time. L^+ may be empty too, e.g., when one attempts to remove the outdated behavior specified using L^- from the process tree. In turn, L^- can consist of only old traces or traces believed to be infrequent, incorrect, or outdated. Gradual removal of the outdated behavior may involve a classifier that periodically yields the outdated traces based on its assessment. L^- can be also empty, when one aims at incremental learning of the process tree for all observed behavior.

Definition 5 relates to the problem of online discovery of a process model, as posed, e.g., in [21, 24]. That problem aims at the adaptation of an existing process model given a continuous stream of events rather than traces, thus requiring the discovery algorithm to detect end events. Definition 5 transfers the task of grouping events into traces from the discovery algorithm to event log preprocessing. The end-to-end behavior is given. The discovery algorithm may use this information to augment a process model with certain guarantees. In contrast, providing guarantees like perfect fit is meaningless for event streams that do not hold guaranteed end-to-end behavior. For instance, consider the Sepsis_cases event log [54] containing the traces of sepsis diagnosis and treatment. Among the events of normal medical operations, there are events that indicate the return of patients months or years after release. By flattening this event log into an event stream the discovery algorithm cannot judge whether the release event is the last one for sure, even months after its occurrence. However, the correct judgment is crucial to build a perfect fit model and generalize to the returning cases.

CIM imposes no additional restrictions on the differential event logs L^+ and L^- and applies to all these use cases of Definition 5. The unquestionable advantages of CIM are the guarantee of producing perfectly fit and sound process trees, single-pass event log processing, and the well-handling of base cases. The first property makes CIM the perfect tool for auditing real-world processes, where one looks for abuses and frauds, which are rare and otherwise might remain hidden. The single-pass event log processing opens the way to efficiently handle large event logs, as every piece of information must be read once. CIM also offers better handling of the base cases than its batch predecessor IMd [8], resulting in better fit process trees.

The experiment shows that despite the extra cost of computing differences, CIM updates the existing process tree faster than IM [5] and IMd [8] build a new process tree from scratch given a large-enough time window. The pay-off point is problem-dependent. We observe it for time windows as small as 10–50 traces. This is very little, as typically, we audit processes having hundreds, thousands, or even millions of traces. CIM offers this performance gain without loss of generalization and precision compared with IM. The differences in the process trees produced by CIM w.r.t. IM result from the entire history of updates. Hence, the produced process trees hold extra information on the process that is not available to IM.

CIM embraces a mechanism of the detection and identification of the concept drift in the given process tree T. When the set of distinct traces included in the time window changes, CIM finds the minimal common subtree of all changes in T (cf. Section 5.3). The minimal common subtree identifies the location of the concept drift. The position of the time window indicates the time of the drift occurrence. Figure 3 shows that in our experiments, only an event-log-dependent fraction of window shifts causes updates. This mechanism of the concept drift detection may yield false positives. The actual part of T to change may be smaller, and the updated process tree may equal T. The assessment of the properties of this mechanism requires future experimentation using synthetic event logs with known ground truth process trees and moments and locations of the concept drift occurrence.

CIM is not free of the disadvantages inherited from its predecessors [5, 8]. A process tree is unable to model long dependencies among decision operators ($\times$ and $\circlearrowleft$) in different branches. This can be alleviated using decision models associated with the decision operators. However, exploring this opportunity is beyond the scope of this study.

CIM falls short on precision. However, this is a common downside of all compared algorithms. The root cause is the fallback model, e.g., $\circlearrowleft(\tau, a, b, \ldots, z)$, produced wherever it cannot find a cut. Although the fallback model is perfectly fit by definition, it is of little use in practical scenarios, where one looks for more precise models. The elimination of the fallback model is a direction of future research. This can be achieved using e.g., new control flow operators and cuts, probabilistic cuts like in IMin [6], filtering of infrequent arcs in DFG like in IMi [7], and preprocessing of the event log [59]. The techniques adopted in IMin and IMi drop the guarantee of achieving a perfect fit. In turn, preprocessing of the event log is useful when auditing the most frequent behavior of the process. Adopting an event log filtering technique depends on the area of application and specifics of the process and thus lies beyond the scope of this work. Precision can be increased too in favor of a little decrease of generalization by substituting four base cases of CIM with the most common base case of a certain activity (cf. Sections 4.2 and 6.3).

CIM requires uniquely labeled activities: for two events in a trace referring to the same activity name, CIM assumes that this is the same activity corresponding to the same leaf in the

resulting tree. In this way, CIM avoids the need to solve the problem of activity correlation, at the expense of losing on some criteria, e.g., precision, generalization, size.

The experiment in Section 6 involves fixed-size time windows only. This simple approach is sufficient in many use cases, where new traces come at a similar rate over time. For more complex scenarios, an adaptive windowing strategy that resizes the window based on the event rate in the data stream may behave better. In general, there is no simple answer to which strategy fares best, and picking the best strategy requires problem domain-specific research. However, designing the windowing strategy is beyond the scope of this work. When solving a specific process discovery problem one may design a suitable windowing strategy and easily incorporate it with CIM. CIM processes every pair of differential logs independently, and they may vary in size.

The process tree representation, albeit well-suited to the problem of discovery of process models, is not industry-standard and thus may not be widely supported in software. However, process trees transform without loss of information to other representations, e.g., Business Process Modeling and Notation (BPMN) standard [34], and Petri nets [14]. This paves the way to the integration scenario, where CIM maintains internally a process tree, and another system uses this process tree transformed into its representation.

8. CONCLUSION AND FUTURE WORK

We propose the Continuous Inductive Miner (CIM) algorithm, as the proof of the validity of the main research hypothesis from Section 1.2. Given an existing process tree and the differential event logs, CIM adapts this process tree to reflect the changes in the differential event logs. The resulting process tree perfectly fits the current contents of the time window, and is no worse at generalizing than the process tree built from scratch based on the same data. For time windows larger than a few tens of traces, CIM is faster than its batch predecessors [5, 8]. CIM hybridizes other advantageous properties of [5, 8] too: the soundness of the resulting process trees, single-pass log processing, and well-handling of base cases.

CIM does not solve all issues of [5, 8]. It sometimes produces the fallback model and is unable to handle duplicate activities. It also does not address noise and incompleteness in the event logs. Future research should follow these directions, e.g., including new control flow operators and cuts, noise filtering, and probabilistic detection of cuts in the update scenario.

ACKNOWLEDGEMENTS

This work was supported by the National Centre for Research and Development, Poland, grant no. LIDER/14/0086/L-10/18/NCBR/2019 (see https://processm.cs.put.poznan.pl).

REFERENCES

[1] M. Zaborowski, "Data processing in self-controlling enterprise processes," *Bull. Pol. Acad. Sci. Tech. Sci.*, vol. 67, no. 1, pp. 3–20, 2019.

[2] XES Working Group, "IEEE Standard for eXtensible Event Stream (XES) for Achieving Interoperability in Event Logs and Event Streams," *IEEE Std 1849-2016*, pp. 1–50, 2016.

[3] W. van der Aalst, J. Buijs, and B. van Dongen, "Towards improving the representational bias of process mining," in *Data-Driven Process Discovery and Analysis*. Berlin, Heidelberg: Springer, 2012, pp. 39–54.

[4] W.M.P. van der Aalst, *Process Mining: Data Science in Action*, 2nd ed. Heidelberg: Springer, 2016.

[5] S.J.J. Leemans, D. Fahland, and W.M.P. van der Aalst, "Discovering block-structured process models from event logs - a constructive approach," in *Application and Theory of Petri Nets and Concurrency*. Springer, 2013, pp. 311–329.

[6] S. Leemans, D. Fahland, and W. van der Aalst, "Discovering block-structured process models from incomplete event logs," in *Application and Theory of Petri Nets and Concurrency*. Springer, 2014, pp. 91–110.

[7] S.J.J. Leemans, D. Fahland, and W.M.P. van der Aalst, "Discovering block-structured process models from event logs containing infrequent behaviour," in *Business Process Management Workshops*. Springer, 2014, pp. 66–78.

[8] S.J.J. Leemans, D. Fahland, and W.M.P. van der Alast, "Scalable process discovery with guarantees," in *Enterprise, Business-Process and Information Systems Modeling*. Cham: Springer, 2015, pp. 85–101.

[9] J.A. Bergstra and J.W. Klop, "Acpτ a universal axiom system for process specification," in *Algebraic Methods: Theory, Tools and Applications*. Berlin, Heidelberg: Springer, 1989, pp. 445–463.

[10] D. Bechet, P. de Groote, and C. Retoré, "A complete axiomatisation for the inclusion of series-parallel partial orders," in *Rewriting Techniques and Applications*. Berlin, Heidelberg: Springer, 1997, pp. 230–240.

[11] G. Campero-Arena and J. Truss, "1-transitive cyclic orderings," *J. Comb. Theory Ser. A*, vol. 116, no. 3, pp. 581–594, 2009.

[12] A. Berti and W.M.P. van der Aalst, "Reviving token-based replay: Increasing speed while improving diagnostics," in *ATAED@ Petri Nets/ACSD. 2019*, ser. CEUR Workshop Proceedings, vol. 2371. CEUR-WS.org, 2019, pp. 87–103.

[13] J. Munoz-Gama and J. Carmona, "A fresh look at precision in process conformance," in *BPM 2010*, ser. Lecture Notes in Computer Science, vol. 6336. Springer, 2010, pp. 211–226.

[14] T. Murata, "Petri nets: Properties, analysis and applications," *Proc. IEEE*, vol. 77, no. 4, pp. 541–580, 1989.

[15] F. Folino, G. Greco, A. Guzzo, and L. Pontieri, "Mining usage scenarios in business processes: Outlier-aware discovery and run-time prediction," *Data Knowl. Eng.*, vol. 70, no. 12, pp. 1005–1029, 2011.

[16] X. Liu, M. Alshangiti, C. Ding, and Q. Yu, "Log sequence clustering for workflow mining in multi-workflow systems," *Data Knowl. Eng.*, vol. 117, pp. 1–17, 2018.

[17] J.C.A.M. Buijs, B.F. van Dongen, and W.M.P. van der Aalst, "A genetic algorithm for discovering process trees," in *2012 IEEE Congress on Evolutionary Computation*, 2012, pp. 1–8.

[18] S.J.J. Leemans, N. Tax, and A.H.M. ter Hofstede, "Indulpet miner: Combining discovery algorithms," in *On the Move to Meaningful Internet Systems. OTM 2018 Conferences*. Cham: Springer, 2018, pp. 97–115.

[19] N. Tax, N. Sidorova, R. Haakma, and W.M. van der Aalst, "Mining local process models," *J. Innovation Digital Ecosyst.*, vol. 3, no. 2, pp. 183–196, 2016.

[20] D. Redlich, T. Molka, W. Gilani, G. Blair, and A. Rashid, "Constructs competition miner: Process control-flow discovery of bp-domain constructs," in *Business Process Management*. Cham: Springer, 2014, pp. 134–150.

[21] D. Redlich, T. Molka, W. Gilani, G.S. Blair, and A. Rashid, "Scalable dynamic business process discovery with the constructs competition miner," in *SIMPDA 2014*, 2014, pp. 91–107.

[22] J. Carmona and R. Gavaldà, "Online techniques for dealing with concept drift in process mining," in *Advances in Intelligent Data Analysis XI*. Berlin, Heidelberg: Springer, 2012, pp. 90–102.

[23] A. Bifet and R. Gavaldà, "Learning from time-changing data with adaptive windowing," in *In SIAM International Conference on Data Mining*, 2007.

[24] S. van Zelst, B. van Dongen, and W. van der Aalst, "Event stream-based process discovery using abstract representations," *Knowl. Inf. Syst.*, no. 54, pp. 407–435, 2018.

[25] R.P.J.C. Bose, W.M.P. van der Aalst, I. Žliobaitė, and M. Pechenizkiy, "Dealing with concept drifts in process mining," *IEEE Trans. Neural Networks*, vol. 25, no. 1, pp. 154–171, 2014.

[26] F. Stertz and S. Rinderle-Ma, "Process histories - detecting and representing concept drifts based on event streams," in *On the Move to Meaningful Internet Systems. OTM 2018 Conferences*. Springer, 2018, pp. 318–335.

[27] A. Burattin, M. Cimitile, F.M. Maggi, and A. Sperduti, "Online discovery of declarative process models from event streams," *IEEE Trans. Serv. Comput.*, vol. 8, no. 6, pp. 833–846, 2015.

[28] A. Burattin, A. Sperduti, and W.M.P. van der Aalst, "Control-flow discovery from event streams," in *2014 IEEE Congress on Evolutionary Computation (CEC)*, 2014, pp. 2420–2427.

[29] J. Potoniec, D. Sroka, and T.P. Pawlak, "Continuous discovery of causal nets for non-stationary business processes using the online miner," *Eur. J. Oper. Res.*, vol. 303, pp. 1304–1320, 2022.

[30] A. Polyvyanyy, J. Vanhatalo, and H. Völzer, "Simplified computation and generalization of the refined process structure tree," in *Web Services and Formal Methods*. Berlin, Heidelberg: Springer, 2011, pp. 25–41.

[31] A. Polyvyanyy, L. García-Bañuelos, D. Fahland, and M. Weske, "Maximal Structuring of Acyclic Process Models," *Comput. J.*, vol. 57, no. 1, pp. 12–35, 09 2012.

[32] A. Augusto, R. Conforti, M. Dumas, M. La Rosa, and G. Bruno, "Automated discovery of structured process models from event logs: The discover-and-structure approach," *Data Knowl. Eng.*, vol. 117, pp. 373–392, 2018.

[33] A.J.M.M. Weijters and J.T.S. Ribeiro, "Flexible heuristics miner (FHM)," in *Computational Intelligence and Data Mining (CIDM), 2011 IEEE Symposium on. IEEE*, 2011, pp. 310–317.

[34] OMG, "Business Process Model and Notation (BPMN), Version 2.0," Object Management Group, 2011.

[35] G. Greco, A. Guzzo, and L. Pontieri, "Mining taxonomies of process models," *Data Knowl. Eng.*, vol. 67, no. 1, pp. 74–102, 2008.

[36] W. van der Aalst, T. Weijters, and L. Maruster, "Workflow mining: discovering process models from event logs," *IEEE Trans. Knowl. Data Eng.*, vol. 16, no. 9, pp. 1128–1142, 2004.

[37] S. Goedertier, J. De Weerdt, D. Martens, J. Vanthienen, and B. Baesens, "Process discovery in event logs: An application in the telecom industry," *Appl. Soft Comput.*, vol. 11, no. 2, pp. 1697–1710, 2011.

[38] A. Alves De Medeiros, A. Weijters, and W. Aalst, van der, "Genetic process mining: an experimental evaluation," *Data Min. Knowl. Discovery*, vol. 14, no. 2, pp. 245–304, 2007.

[39] S. Goedertier, D. Martens, J. Vanthienen, and B. Baesens, "Robust process discovery with artificial negative events," *J. Mach. Learn. Res.*, vol. 10, pp. 1305–1340, 2009.

[40] H. Sun, W. Liu, L. Qi, Y. Du, X. Ren, and X. Liu, "A process mining algorithm to mixed multiple-concurrency short-loop structures," *Inf. Sci.*, vol. 542, pp. 453–475, 2021.

[41] L. Wen, J. Wang, W.M. van der Aalst, B. Huang, and J. Sun, "Mining process models with prime invisible tasks," *Data Knowl. Eng.*, vol. 69, no. 10, pp. 999–1021, 2010.

[42] A. Augusto, R. Conforti, M. Dumas, M.L. Rosa, F.M. Maggi, A. Marrella, M. Mecella, and A. Soo, "Automated discovery of process models from event logs: Review and benchmark," *IEEE Trans. Knowl. Data Eng.*, vol. 31, no. 4, pp. 686–705, 2019.

[43] A. Augusto, R. Conforti, M. Dumas, and M. La Rosa, "Split miner: automated discovery of accurate and simple business process models from event logs," *Knowl. Inf. Syst.*, no. 59, p. 251–284, 2019.

[44] B. Vázquez-Barreiros, M. Mucientes, and M. Lama, "Prodigen: Mining complete, precise and minimal structure process models with a genetic algorithm," *Inf. Sci.*, vol. 294, pp. 315–333, 2015.

[45] P. Kudła and T.P. Pawlak, "One-class synthesis of constraints for mixed-integer linear programming with C4.5 decision trees," *Appl. Soft Comput.*, vol. 68, pp. 1–12, 2018.

[46] H. Williams, *Model Building in Mathematical Programming*. Wiley, 2013.

[47] R. Tarjan, "Depth-first search and linear graph algorithms," *SIAM J. Comput.*, vol. 1, no. 2, pp. 146–160, 1972.

[48] E. Nuutila and E. Soisalon-Soininen, "On finding the strongly connected components in a directed graph," *Inf. Process. Lett.*, vol. 49, no. 1, pp. 9–14, 1994.

[49] W. Steeman, "BPI challenge 2013," Apr 2014.

[50] B.B. van Dongen, "BPI challenge 2015," May 2015, doi: 10.4121/uuid:31a308ef-c844-48da-948c-305d167a0ec1.

[51] J. Buijs, "Environmental permit application process ('WABO'), CoSeLoG project," May 2014, doi: 10.4121/uuid:26aba40d-8b2d-435b-b5af-6d4bfbd7a270.

[52] B. van Dongen, "Real-life event logs – Hospital log," 3 2011, doi: 10.4121/uuid:d9769f3d-0ab0-4fb8-803b-0d1120ffcf54.

[53] J. Buijs, "Receipt phase of an environmental permit application process ('WABO'), CoSeLoG project," 8 2014, doi: 10.4121/uuid:a07386a5-7be3-4367-9535-70bc9e77dbe6.

[54] F. Mannhardt, "Sepsis cases – event log," Dec 2016, doi: 10.4121/uuid:915d2bfb-7e84-49ad-a286-dc35f063a460.

[55] M. Leemans, W.M.P. van der Aalst, and M.G.J. van den Brand, "Recursion aware modeling and discovery for hierarchical software event log analysis," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2018, pp. 185–196.

[56] 4TU, "4TU.ResearchData," 2022. [Online]. Available: https://data.4tu.nl

[57] G. Kanji, *100 Statistical Tests*. SAGE Publications, 1999.

[58] S. Holm, "A simple sequentially rejective multiple test procedure," *Scand. J. Stat.*, vol. 6, no. 2, pp. 65–70, 1979.

[59] R. Conforti, M.L. Rosa, and A.H.M. t. Hofstede, "Filtering out infrequent behavior from business process event logs," *IEEE Trans. Knowl. Data Eng.*, vol. 29, no. 2, pp. 300–314, 2017.