Robust Drift Characterization from Event Streams of Business Processes

ALIREZA OSTOVAR, The University of Melbourne, Australia
SANDER J.J. LEEMANS, Queensland University of Technology, Australia
MARCELLO LA ROSA, The University of Melbourne, Australia

Process workers may vary the normal execution of a business process to adjust to changes in their operational environment, e.g. changes in workload, season or regulations. Changes may be simple, such as skipping an individual activity, or complex, such as replacing an entire procedure with another. Over time, these changes may negatively affect process performance; hence it is important to identify and understand them early on. As such, a number of techniques have been developed to detect *process drifts*, i.e. statistically significant changes in process behavior, from process event logs (offline) or event streams (online). However, detecting a drift without characterizing it, i.e. without providing explanations on its nature, is not enough to help analysts understand and rectify root causes for process performance issues. Existing approaches for drift characterization are limited to simple changes that affect individual activities. This paper contributes an efficient, accurate and noise-tolerant automated method for characterizing complex drifts affecting entire process fragments. The method, which works both offline and online, relies on two cornerstone techniques: one to automatically discover process trees from event streams (logs), the other to transform process trees using a minimum number of change operations. The operations identified are then translated into natural language statements to explain the change behind a drift. The method has been extensively evaluated on artificial and real-life datasets, and against a state-of-the-art baseline method. The results from one of the real-life datasets have also been validated with a process stakeholder.

CCS Concepts: • **Information systems** → *Data analytics*; • **Applied computing** → **Business process monitoring**;

Additional Key Words and Phrases: concept drift, process drift, drift characterization, business process, process mining, business process management

ACM Reference Format:
Alireza Ostovar, Sander J.J. Leemans, and Marcello La Rosa. 2019. Robust Drift Characterization from Event Streams of Business Processes. *ACM Trans. Knowl. Discov. Data.* 1, 1 (December 2019), 56 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Process mining [29] is concerned with creating effective tools to extract actionable knowledge from the execution records of business processes as logged by information systems, so that process performance or compliance issues can be identified and rectified. Such records may be in the form of historical event logs or live event streams. A wide range of process mining techniques have been

Authors' addresses: Alireza Ostovar, The University of Melbourne, Level 10, Doug McDonell Building, Parkville, VIC, 3010, Australia, alireza.ostovar@unimelb.edu.au; Sander J.J. Leemans, Queensland University of Technology, 2 George St. Brisbane, QLD, 4000, Australia, sander.leemans@qut.edu.au; Marcello La Rosa, The University of Melbourne, Level 10, Doug McDonell Building, Parkville, VIC, 3010, Australia, marcello.larosa@unimelb.edu.au.

1556-4681/2019/12-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

developed over the past few years. For example, there are techniques for discovering a process model from an event log [3], or techniques for predicting properties of ongoing process cases, such as remaining time or outcome, from an event stream [27, 31].

Most process mining techniques assume processes are in steady state [10]. However, in reality business processes evolve over time in response to various changes in the business environments in which they operate. For example, changes in regulations, competition, supply, demand, and technological capabilities, or internal changes such as resource capacity and workload, or even changes in seasonal factors can all impact business processes. Some process changes are planned ahead and documented, while others such as ad-hoc workarounds initiated by individuals or changes due to the replacement of human resources may be unintentional and undocumented. Over time, these changes may negatively impact process performance, and more generally hamper process improvement initiatives.

In light of the above, a number of techniques have been developed to detect, localize and characterize *process drifts*, i.e. statistically significant changes in the process behavior, either from event logs or from event streams [1, 5, 6, 19, 20, 23, 24, 35]. However, while such techniques are quite reliable when it comes to detecting and pinpointing the location of the drift, drift characterization suffers from several limitations. First, techniques to characterize drifts [23], and more generally techniques to characterize differences in the behavior of process variants, e.g. [4, 28], rely on a low abstraction level to capture the process behavior. This results in a proliferation of low-level changes being reported, each referring to an individual activity, e.g. removing or adding an activity. Second, when the drift relates to multiple activities, such as skipping a process fragment containing two concurrent activities[1] or removing a fragment of two conditional activities, these changes are completely missed or only partially explained. Another limitation is the inability to characterize overlapping changes, i.e. changes that share some behavioral relations, as well as nested changes, i.e. overlapping changes where each change is applied to the fragment resulting from the application of the previous change. The consequence of these limitations is that process drift chacterization hardly works in real-life settings.

To address the above limitations, this paper proposes a *robust* method for charactering process drifts from event streams, i.e. a method that fulfils the following criteria:

- *Efficient*: Drifts should be characterized within reasonable time bounds, especially if employed in online settings, i.e. in the context of event streams.
- *Accurate*: Drifts should be characterized as accurately as possible. An incorrect characterization may not only confuse users, but could also lead to incorrect process improvement decisions. An example of an inaccurate drift characterization is reporting the insertion of a loop as activity duplication.
- *Noise-tolerant*: Real-life event logs and event streams often contain noise, e.g. in the form of infrequent events. Drift characterization should not be affected by noise: sporadic changes cased by noise should not be described as part of a drift.
- *Understandable*: Drifts should be characaterized in a way that is easy to understand by business analysts, especially in the context of complex changes. For example, a drift can be represented by outputting a sample of traces before and after the drift point. However, this does not provide an effective mechanism to easily understand the drift.
- *Automated*: Drifts should be characterized with no manual intervention. As drifts may occur as a result of unplanned or undocumented process changes, drift characterization should not rely on user knowledge.

[1] In this context, two activities are *concurrent* if they are independent of each other and as such can be performed in any order.

The core idea is to discover two *process trees*, i.e. block-structured process models, from the portion of the event stream before and after the drift, and use a process tree transformation technique to find a minimum-cost sequence of edit operations that transforms the pre-drift process tree to the post-drift process tree. The underpinning assumption is that edit operations within such a sequence manifest control-flow changes of the process underlying the drift. Each process (sub-)tree represents a single-entry single-exit process fragment. As such, we define a set of fragment-based edit operations, each capturing a (complex) change affecting one or more process fragments. The definition of the edit operations and the cost of applying them is such that a minimum-cost sequence of edit operations provides a detailed yet concise explanation of the process changes. That is, if a change involves an individual activity within the process, then it is explained by one change in the sequence referring to that activity. On the other hand, if a change involves a fragment of multiple activities, then it is explained by one change in the sequence referring to that fragment as a whole, hence limiting the total number of changes reported. Moreover, the hierarchical structure of a process tree allows the characterization of more complex changes such as overlapping and nested changes. The identified fragment-level changes are translated into natural language statements based on typical business process change patterns [33].

We evaluated the accuracy and conciseness of the statements reported by our method by characterizing drifts from artificial and real-life datasets in various settings. We then compared these results with those obtained by a baseline method for drift characterization. In addition, we validated the results from one the real-life datasets with a process stakeholder.

Against this backdrop, the rest of this paper is structured as follows. Section 2 discusses related work in the areas of process mining and data mining, while Section 3 provides preliminary definitions such as event logs and process trees. Next, Sections 4–6 illustrate the proposed method in three steps: i) process tree discovery, ii) process tree transformation, and iii) computation of characterization statements. Sections 7 and 8 present the evaluation on artificial and real-life logs, before Section 9 concludes the paper. Three appendices provide supporting definitions and summarize the notation adopted in the paper.

2 RELATED WORK

Various techniques have been developed for detecting process drifts [1, 5, 6, 19, 20, 24, 35]. The main idea of these techniques is extracting features (e.g. patterns) from the process behavior recorded in event logs or in event streams and performing certain analysis to detect a drift. For example, Bose et al. [5] propose to perform statistical tests over feature vectors, where the features are to be selected by user, assuming that they have a-priori knowledge of the possible nature of the drift. Maaradji et al. [19] detect a drift by performing a statistical test on distributions of partially ordered runs in two sliding windows on a stream of traces. However, using partial ordered runs to encode the process behavior limits the application of this technique to logs of complete traces with low trace variability, i.e. the ratio of distinct traces to the total number of traces. To address this limitation Ostovar et al. [24] use α^+ binary relations to encode process behavior in event streams of highly variable processes. This technique is able to detect inter-trace drifts, i.e. drifts that occur between the executions of a process, as well as intra-trace drifts that occur during the execution of a process.

A possible approach to characterize a drift is to compare two sub-logs extracted from the event stream before and after the drift and identify their differences. In this context, Bolt et al. [4] propose a technique for comparing the behavior of different variants of the same process based on observed executions of such variants in event logs. Given two event logs each corresponding to a process variant, they follow a three-step approach. In the first step, they build a transition system from the event logs and annotate each of its states and transitions with the measurements of the variants with respect to a certain process metric. In the second step, they perform a statistical test between every two sets of measurements of each metric on each state or transition to identify the differences that are statistically significant. Finally, the identified differences are highlighted by changing the appearance of the states or transitions. For example, if the difference between the frequency of executing an activity in two compared variants is statistically significant, the arc corresponding to that activity in the transition system is thickened. By using the sub-logs extracted from before and after a drift as input to this technique, we can identify some of the significant differences between the pre-drift and post-drift process variants. However, this technique has several limitations. With respect to the control-flow differences, it is only able to identify that a certain activity (transition) occurs after a sequence of activities (state) in one process but not in the other, while missing the structural differences of the processes, e.g. the occurrence of two activities in an XOR construct in one process but not in the other. Furthermore, this technique is not meant to work with event streams where each sub-log before or after a drift contains partial traces, i.e. traces whose start events are removed from the stream and/or whose end events are yet to arrive on the stream. Assuming that we know the start and end activities of the process, one possible workaround is to build a transition system by only using complete traces within the pre-drift and post-drift sub-logs. However, this may lead to an incomplete or even inaccurate transition system as fractions of process behavior that are only captured by the discarded partial traces are missed by the transition system. This problem is worsened in the event streams of highly variable processes, as almost every trace of such processes exhibits a unique execution of the process. A sub-log extracted from such an event stream is likely to only contain partial traces.

Existing approaches to log-based process variant comparison such as the one by Bolt et al., are restricted to intra-case relations, and more specifically, directly-follows relations such as “a task directly follows another one” or a “resource directly hands-off to another resource” within the same case. Nguyen et al. [22] developed a more general approach based on so-called perspective graphs. A perspective graph is a graph-based abstraction of an event log where a node represents any entity in an event log (task, resource, location, etc.) and an arc represents an arbitrary relation between these entities (e.g. directly-follows, co-occurs, hands-off to, works-together with, etc.) within or across cases. Statistically significant differences between two perspective graphs are captured in a so-called differential perspective graph, allowing users to compare two event logs from any given perspective. The technique takes as input two event logs and a set of user-defined parameters for comparison; the output is a matrix-based visualization of differences between the two logs, which needs to be manually inspected, and refined, by the user (e.g. drilling down into a specific difference). Similar to the technique proposed by Bolt et al. [4], this technique requires visual inspection of the differences (two transition systems in the case of Bolt et al., two matrices in the case of Nguyen et al.). Moreover, it does not cater for partial traces, i.e. the input must be two complete (sub-)logs, and thus cannot work in the context of event streams.

Van Beest et al. [28] propose a technique for diagnosing behavioral differences between two event logs. The idea is to use two prime event structures to losslessly encode the process behavior captured by two event logs, and by comparing the two event structures report their differences as natural language statements. An event structure is a directed graph composed of events and behavioral relations (causality, conflict and concurrency) between events, so they focus on the control-flow perspective of a business process. A problem of this technique when used for drift characterization is that it reports all differences between the pre-drift and post-drift sub-logs regardless of the significance of their association with the occurrence of the drift. Furthermore, the identified differences are at the level of individual activities. Consequently, a problem is that this technique reports a large number of differences, specially when the changes are applied to process fragments, or when they occur in a nested way. For example, for a simple fragment-level change, where we parallelize two sequential fragments, each consisting of four activities, this method would report 16 differences, each capturing the parallelization of two activities. Obviously, it is not easy to understand and analyze such a large number of differences. Another limitation of this technique is its high execution time, specially when it needs to compare two large event structures with several differences. Finally, similar to the technique proposed by Bolt et al. [4], this technique does not work with partial traces, hence missing the fraction of process behavior that is captured by partial traces.

Another approach to drift characterization is to first discover two process models, one from the pre-drift sub-log and the other from the post-drift sub-log, and compare them using a model-to-model comparison technique. In this context, Armas-Cervantes et al. [2] propose a method for diagnosing behavioral differences between two process models based on canonically reduced event structures. The idea is similar to that of Van Beest et al. [28], though this time the two event structures are built from the process models, and then compared with each other to distill a set of natural language statements that capture their differences. As such, all the shortcomings related to the use of event structures outlined for the method by Van Beest et al., also apply to this method. Moreover, no automated discovery technique is able to guarantee a perfect discovery accuracy [3]. These techniques strike different tradeoffs between fully capturing the behavior of the log and over-generalizing that behavior. Because of this, some differences between the pre-drift and post-drift sub-logs may be missed (as the related behavior is not captured by the discovered model) or spurious differences may be added (as extra behavior has been added in the discovered model).

By building upon the drift detection technique in [24], Ostovar et al. introduce a drift characterization method in [23]. The α^+ binary relations used for the detection of a drift are first filtered using a statistical test to remove relations that do not have a significant statistical association with the drift. The remaining relations are then mapped to a predefined set of typical change templates and the best matching templates are translated into characterization statements. This method is designed to characterize changes to individual activities, such as adding a new activity or parallelizing two sequential activities. However, given the low-level of abstraction employed by the α^+ relations, this method fails to fully characterize more complex changes, including changes that involve multiple activities at once, overlapping changes, and nested changes.

Concept drift detection has also been studied in the field of data mining [9], where a drift mainly refers to the change in the relation between the input and the target variables in an online supervised learning scenario. As such, a widely studied challenge is that of devising learning algorithms that can detect a concept drift as quickly as possible and adapt to the new concept (a.k.a. adaptive learning). In this context, the term *drift characterization* is used for describing different properties of a drift as well as explaining concept changes. For example, some studies focus on analyzing a specific metric of a drift, e.g. severity, predictability, and frequency [11, 21]. Webb et al. [32] propose a comprehensive framework for quantitative analysis of a drift, e.g. measuring drift magnitude or drift duration. They also qualitatively categorize drifts into different types based on their occurrence with respect to time, e.g. sudden or gradual. On the other hand, some studies have explored techniques for the identification of features that explain the drift. For instance, in [25], the authors use brushed parallel histograms to visualize concept drifts in multidimensional problem spaces. In this paper, we also focus on the identification of process changes with the occurrence of a drift from a control-flow perspective. However, the methods developed for characterizing a drift in data mining deal with simple data structures (e.g. numerical or categorical variables and vectors thereof), while in business process drift characterization we seek to characterize changes in more complex structures, e.g. changes to process model fragments, where each fragment involves multiple activities connected via behavioral relations (causality, concurrency or conflict relations).

3 PRELIMINARIES

This section introduces basic notions such as event logs, process trees and fragments. The notation used in this paper is summarized in Appendix C.

Event logs are at the core of all process mining techniques. An event log is a set of traces, each recording the sequence of events originated from a given process instance. Each event represents an occurrence of an activity. The configuration where these events are read individually from an online source is known as event streaming. An event stream is a potentially infinite sequence of events, where events are ordered by time and indexed. The consecutive events of an event stream do not need to belong to the same trace, i.e. traces can be "overlapping". Formally:

Definition 1 (Event log, Trace, Event stream). *Let L be an* event log *over the set of labels $\mathscr{L}$, i.e. $L \in \mathscr{P}(\mathscr{L}^*)$. Let $\mathscr{E}$ be the set of event occurrences and $\lambda : \mathscr{E} \to \mathscr{L}$ a labeling function. A* trace *$\Sigma \in L$ is a sequence of events $\mathscr{E}_\Sigma \subseteq \mathscr{E}$ with $|\mathscr{E}_\Sigma| = n$ such that $\Sigma = \langle \lambda(e_0), \lambda(e_1), \ldots, \lambda(e_{n-1}) \rangle$. An* event stream *is a partial bijective function $S : \mathbb{N} \to \mathscr{E}$ that maps every element from the index $\mathbb{N}$ to $\mathscr{E}$.*

□

For example, the following represents an event log with a total of six traces, with two distinct traces: $L = \{\langle a, b, d\rangle^2, \langle a, b, c, d\rangle^4\}$.

A *tree* is an acyclic, connected graph. For a tree T, the sets containing nodes and edges are denoted by $V(T)$ and $E(T)$, respectively. The size of T is $|V(T)|$ and is denoted by $|T|$. We sometimes denote $v \in V(T)$ as $v \in T$. The root node of a tree T is denoted by $root(T)$. We denote the subtree of T rooted at $v \in T$ by $T\langle v\rangle$.

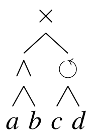
For each non-root node v in T, let $Down_T(v) \subset V(T)$ be the sequence of nodes on the shortest path from $root(T)$ to v. The *parent* of v is its adjacent node in $Down_T(v)$. The parent of root r is undefined. We say v is a *child* of u if u is the parent of v. The nodes in $Down_T(v)$ preceding v are called *ancestors* of v in T. We say v is a *descendant* of u if u is an ancestor of v. The nodes with the same parent are called *siblings*. A node with no children is called a *leaf*. A non-leaf node is called an *internal* node. The set of leaves under an internal node $v \in T$ is denoted by $leaves(v)$. We denote the label of a node v by $l(v)$.

The *depth* of v in T, is denoted by $dep(v)$ and equals to $|Down_T(v)| - 1$. The depth of T, denoted by $dep(T)$ equals to the maximum depth of its nodes, i.e. $dep(T) = \max_{v \in V(T)} dep(v)$. For the nodes $v_1, \ldots, v_n$ in T we define a *common ancestor (CA)* as a node in $Down_T(v_1) \cap \ldots \cap Down_T(v_n)$. Also, we define the *lowest common ancestor (LCA)* as the deepest CA, and denote it by $LCA(v_1, \ldots, v_n)$. Accordingly, for the trees $T\langle v_1\rangle, \ldots, T\langle v_n\rangle$ in T the *lowest common ancestor (LCA)* is denoted by $LCA(T\langle v_1\rangle, \ldots, T\langle v_n\rangle)$, and is the same as $LCA(v_1, \ldots, v_n)$.

A *process tree* is a rooted labeled tree that provides an abstract hierarchical representation of a block-structured workflow net [15]. We define its syntax as follows:

Definition 2 (Process tree). *Let $\mathscr{L}$ be a set of activity labels, and $\mathscr{O} = \{\times, \to, \wedge, \circlearrowleft\}$ be a set of operator labels. Then, an activity node t with $l(t) \in \mathscr{L}$ is a process tree, a τ-node with $l(\tau) \in \{\tau\}$ is a process tree, and $\oplus(P_1, \ldots P_n)$ is a process tree, in which $\oplus$ is a process tree* operator node *with $l(\oplus) \in \mathscr{O}$, and $P_1 \ldots P_n$ are process trees.* □

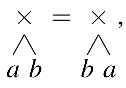
A process tree expresses a language: an activity node t represents the singleton language $l(t)$, a τ-node represents the language with the empty trace, and an operator node represents a certain combination of the languages of its subtrees $P_1 \ldots P_n$, depending on its label. In this paper, we have the following four operator labels: 1) $\times$ expresses the exclusive choice between its subtrees, 2) $\to$ expresses the sequential composition of its subtrees, 3) $\wedge$ expresses the concurrent composition of its subtrees, 4) $\circlearrowleft$ expresses the structured loop of its first subtree (loopbody), followed by the alternative loopback path of its second subtree.

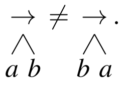
For instance, the process tree 

expresses the language $\{\langle a,b\rangle, \langle b,a\rangle, \langle c\rangle, \langle c,d,c\rangle, \langle c,d,c,d,c\rangle \ldots\}$.

In a process tree P, a leaf node is either an activity node or a τ node, whereas an internal node is always an operator node and must at least have two children. We define $\Gamma = \mathscr{L} \cup \{\tau\} \cup \mathscr{O}$ as a fixed finite alphabet which assigns a label to each node in a process tree. In a process tree, if an activity node has a unique label, we sometimes refer to that activity node by its label. The set of activity nodes under an operator node $v \in P$ is denoted by $C(v)$, and contains the activity nodes in $P\langle v\rangle$. By replaying an event log on top of a process tree we can annotate each node of the tree with its execution frequency. We call the ratio of the frequency of a node v to the frequency of its parent the *relative frequency* of v.

The relation between the nodes $v_1,\ldots,v_n$ in P is defined by the operator of their LCA, i.e. mutually-exclusive ($\times$), concurrent ($\wedge$), sequential ($\rightarrow$), or loop ($\circlearrowleft$). Accordingly, the relation between the process trees $P\langle v_1\rangle,\ldots,P\langle v_n\rangle$ in P is the same as the relation between the nodes $v_1,\ldots,v_n$.

A process tree can contain both ordered and unordered operator nodes. An operator node $\oplus$ is *unordered* if it is commutative, i.e. $\oplus(P_1,\ldots,P_n) = \oplus(P_n,\ldots,P_1)$, it is *ordered* otherwise. The operator nodes $\times$ and $\wedge$ are unordered, whereas $\rightarrow$ and $\circlearrowleft$ are ordered. For example, 

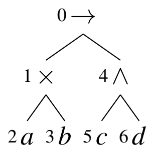
, whereas 

.

The pre-order index of v in a process tree P is denoted by $pre_P(v)$, and is the same as one in an ordered tree when arbitrarily fixing the order of siblings parented by unordered operator nodes in P. We refer to the node with the pre-order index of i in P by $P[i]$. Also, in the process tree examples in this paper, the number on the left side of a node indicates its pre-order index.

For a node v and an ordered operator node $\oplus$ in a process tree P such that v is a descendant of $\oplus$, we define a function *Rank* returning the rank of v in $\oplus$.

DEFINITION 3 (RANK). *Let $\preceq$ be the order on the children of an ordered operator node $\oplus$ in a process tree P. Also, let $v \in P$ be a descendant of $\oplus$ and $c \in P$ be a child of $\oplus$ such that $c \in Down_P(v)$, then $Rank(v,\oplus) = |\{c' \in$ children of $\oplus \mid c' \preceq c\}|$.* □

EXAMPLE 1. *As an example, let us assume the process tree P* 

. The rank of each non-root node $P[i]$ in the $\rightarrow$-node $P[0]$ is as follows.
$Rank(P[1],\ P[0]) = 1$, $Rank(P[2],\ P[0]) = 1$, $Rank(P[3],\ P[0]) = 1$, $Rank(P[4],\ P[0]) = 2$, $Rank(P[5],\ P[0]) = 2$, $Rank(P[6],\ P[0]) = 2$.

There might be multiple process trees with the same language. For example, the tree $\times(a, \times(b,\ c))$ expresses the same language as $\times(\times(a,\ b),\ c)$. As in this paper we compare the structures of two trees to characterize a drift, we need to have one structurally unique tree for each language. A set of structural reduction rules is introduced in [13], which guarantees to preserve the language of a process tree. Repeated application of these rules to a process tree leads to a syntactically unique normal form, i.e. for each language, there is at most one process tree in normal form. In our example, the normal form would be $\times(a,\ b,\ c)$. In this paper, we use a subset of these reduction rules, defined below.

DEFINITION 4 (REDUCTION RULES). *Let M, Q, and R be process trees, and let $\ldots$ be any number of process trees (possibly 0). Then, the reduction rules are as follows:*

$$\begin{array}{ll}
 & \rule{4cm}{0.4pt}\ \textit{singularity rule} \\
(\mathrm{S}) & \oplus(M) \Rightarrow M \textit{ with } \oplus \in \{\times, \rightarrow, \wedge\} \\
 & \rule{4cm}{0.4pt}\ \textit{associativity reduction rules} \\
(\mathrm{A}_{\times}) & \times(\ldots_1, \times(\ldots_2)) \Rightarrow \times(\ldots_1, \ldots_2) \\
(\mathrm{A}_{\rightarrow}) & \rightarrow(\ldots_1, \rightarrow(\ldots_2), \ldots_3) \Rightarrow \rightarrow(\ldots_1, \ldots_2, \ldots_3) \\
(\mathrm{A}_{\wedge}) & \wedge(\ldots_1, \wedge(\ldots_2)) \Rightarrow \wedge(\ldots_1, \ldots_2) \\
 & \rule{4cm}{0.4pt}\ \tau\textit{-reduction rules} \\
(\mathrm{T}_{\rightarrow}) & \rightarrow(\ldots, M, \tau) \Rightarrow \rightarrow(\ldots, M) \\
(\mathrm{T}_{\wedge}) & \wedge(\ldots, M, \tau) \Rightarrow \wedge(\ldots, M)
\end{array}$$

□

A process tree to which no rule can be applied is in *normal form* and is called a *canonical process tree*.

A singularity rule applies to all operators except ↺, as a ↺-node always has two children. This rule is based on the definition of the process tree operators (provided above) that a $\rightarrow$-node, a $\wedge$-node, or an $\times$-node with one child has the same behavior as the child itself. The associativity rule applies to $\times$, $\rightarrow$, and $\wedge$ operators and reduces a tree such as $\times(a, \times(b,\ c))$ to $\times(a,\ b,\ c)$. The τ reduction rules target τ constructs and are defined for $\rightarrow$ and $\wedge$ operators. A τ-node as a child of a $\rightarrow$-node, or a $\wedge$-node does not change the language.

We define a fragment as a process tree representing a single-entry single-exit process fragment. Formally:

DEFINITION 5 (FRAGMENT). *Let P be a process tree rooted at w.*

- *P is a fragment.*
- *Let $S = \{P\langle v_1\rangle, \ldots P\langle v_n\rangle\}$ be the set of process trees under w, where $v_1, \ldots v_n$ are children of w, $\ldots$ is any number of process trees (possibly 0), and $l(w) \in \{\times, \wedge\}$. A process tree $\oplus(P_1,\ \ldots\ P_m)$ parented by w, where $l(\oplus) = l(w)$ and $s = \{P_1, \ldots P_m\}$ is any non-empty proper subset of S, is a fragment. We call such a fragment a* sub-fragment *of w.*
- *Let $S = \{P\langle v_1\rangle, \ldots P\langle v_n\rangle\}$ be the sequence of process trees under w, where $v_1, \ldots v_n$ are children of w, $\ldots$ is any number of process trees (possibly 0), and $l(w) = \rightarrow$. A process tree $\oplus(P_1, \ldots P_m)$ parented by w, where $l(\oplus) = l(w)$ and $s = \{P_1 \ldots P_m\}$ is any nonempty proper subsequence of S, is a fragment provided that any two consecutive elements $P\langle v_i\rangle, P\langle v_{i+1}\rangle$ in s are consecutive in S. We call such a fragment a* sub-fragment *of w.*

Any fragment formed by the nodes within a process tree P is a sub-fragment *of P. We sometimes refer to $P\langle v\rangle$ by $P\langle v\rangle$-fragment. Also, a fragment $P\langle v\rangle$, where v is the child of a node w, is called a child fragment of w. Furthermore, a fragment $f_1 = \tau$ is called a τ-fragment, and as such a fragment $f_2 \neq \tau$ is called a non-τ-fragment.*

□

EXAMPLE 2. *As an example, in the process tree* $\underset{a\ b\ \tau}{\times}$ *the set of all sub-fragments of* $\times$ *is* $\{\underset{a\ b}{\times}, \underset{b\ \tau}{\times}, \underset{a\ \tau}{\times}, a, b, \tau\}$.

EXAMPLE 3. *As an example, consider the simple BPMN model in Figure 1. For the process tree corresponding to this model, i.e.* $\underset{a\quad \underset{\underset{b\ c}{\rightarrow}\ \underset{d\ e}{\circlearrowleft}}{\wedge}\quad f}{\rightarrow}$, *the set of all sub-fragments is* $\{\underset{a\quad \underset{\underset{b\ c}{\rightarrow}\ \underset{d\ e}{\circlearrowleft}}{\wedge}\quad f}{\rightarrow}, \underset{a\quad \underset{\underset{b\ c}{\rightarrow}\ \underset{d\ e}{\circlearrowleft}}{\wedge}}{\rightarrow}, \underset{\underset{\underset{b\ c}{\rightarrow}\ \underset{d\ e}{\circlearrowleft}}{\wedge}\quad f}{\rightarrow}, \underset{\underset{b\ c}{\rightarrow}\ \underset{d\ e}{\circlearrowleft}}{\wedge}, \underset{b\ c}{\rightarrow}, \underset{d\ e}{\circlearrowleft}, a, b, c, d, e, f\}$.

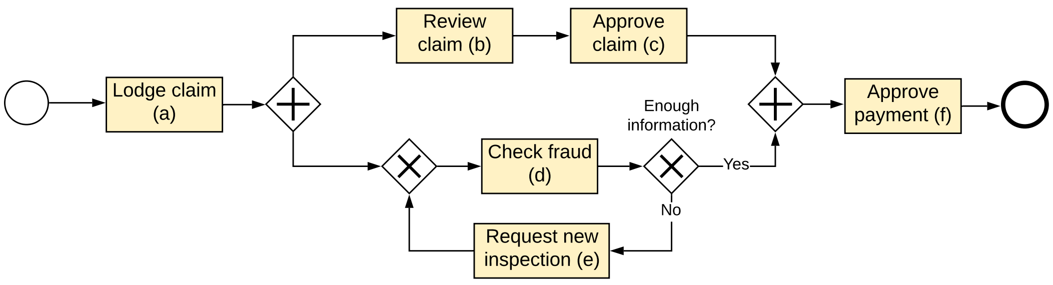


Fig. 1. BPMN model in Example 3.

4 PARTIAL TRACES AND PROCESS TREE DISCOVERY

Given two sub-logs of partial traces, one extracted from before and the other extracted from after a drift, our method characterizes the drift in three steps. In the first step, two process trees P and P' are discovered from the pre-drift and the post-drift sub-logs, respectively. In the second step, a minimum cost sequence of edit operations that transforms P into P' is computed. In the third step, our method constructs characterization statements based on the identified edit operations. An overview of our method is shown in Figure 2. In the rest of this section we illustrate Step 1, while in the next to sections we cover the other two steps.

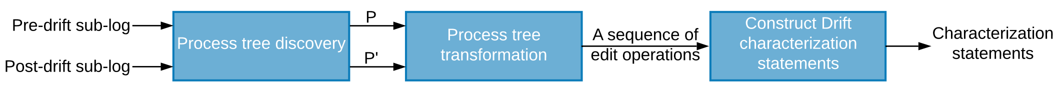
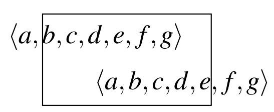

Pre-drift sub-log
Post-drift sub-log
Process tree discovery
P
P'
Process tree transformation
A sequence of edit operations
Construct Drift characterization statements
Characterization statements

Fig. 2. Overview of our method for process drift characterization.

Due to the traces being derived from streams of events, and our application of window-based extraction, we might observe some traces only partially. That is, the start and/or the end of the traces might fall outside of the considered window, as illustrated in Figure 3. Partial traces can be found outside the area of streams as well: if an event log is extracted from a running process, one in fact applies a window to the running process, and every case that is still in progress falls partially outside of the window. Furthermore, cases that were already started before the event log was being captured also fall outside of the window.

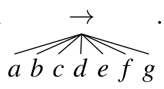
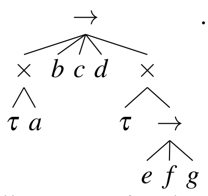


(a) Traces in a window.

$$L_p = [\ |b,c,d,e,f,g\rangle,\\ \langle a,b,c,d,e|\ ]$$

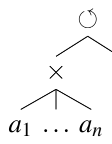
(b) A corresponding log with partial traces (their partiality is denoted by |).

Fig. 3. Example of partial traces. In the window, some traces are observed partially, as they start and/or end outside of the window. In our example, the first and the last trace are only partially observed.

Partially observed traces might influence discovery, which we illustrate using Figure 3. If all traces would have been observed completely, in the log $L = [\langle, a, b, c, d, e, f, g\rangle^3]$, IM would discover the model . However, the event log observed is $L_p = [|b, c, d, e, f, g\rangle, \langle a, b, c, d, e|]$. Without knowledge of partial traces, IM discovers the model . This process tree does not capture the meaning of the partial traces well, as it allows a, e, f and g to be skipped, even though there has not been evidence of this skipping in the event log. One could simply remove the partial traces. However, as seen in our example, these traces add vital information as without them the event log would be empty.

In this section, we first sketch Inductive Miner (IM), which we will later extend to handle partial traces. Second, we describe how partial traces can be detected. Third, we introduce a new process tree discovery algorithm that extends IM by adapting its steps to handle partial traces better.

4.1 Inductive Miner

We chose Inductive Miner (IM) [15] because this algorithm recursively constructs a process tree from an event log, so it naturally lends itself to be adapted to work on partial traces, given that we need to produce process trees as output. Specifically, in the recursion, IM tries to find a *cut* of the event log, consisting of a partition of the activities in the event log and a process tree operator. This cut describes the *most important* behavior in the event log. For instance, the cut $(\rightarrow, \{a, b\}, \{c\})$ denotes that the most important behavior in the event log is 'some behavior with a and b' sequentially followed by 'some behavior with c'. If such a cut can be found, the event log is split accordingly into sub-logs, and on these sub-logs IM recurses, thereby constructing a process tree in a top-down manner. The recursion ends in a *base case*, for instance if only a single activity remains in the event log. Alternatively, if no base case applies and a cut cannot be found, a *fall through* is selected. Several fall throughs have been defined (see [13]), decreasing in precision, in the worst case leading to a *flower model* that allows any behavior with the activities in the event log (e.g. 

).

4.2 Detecting partial traces

Two pieces of information constitute knowledge of partial traces: one needs to decide whether one has seen the first event, and whether the last event has been seen. We refer to a trace of which we have seen the first event as having a *reliable start*, and to a trace of which we have seen the last event as having a *reliable end*. Traces might have both an unreliable start and an unreliable end, or both might be reliable. In Figure 3, the first trace has a reliable end and the second trace has a reliable start.

To detect whether a trace has a reliable start or end, one could manually incorporate domain knowledge. For instance, it could be known that a trace always starts with a "registration" step and always ends with an "archive" step. Then, each trace that starts with "registration" has a reliable start and each trace that ends with "archive" has a reliable end. Other ways to determine reliability include the use of attribute data. For instance, attribute data attached to the trace could indicate whether a trace has been completed.

In our experience, such information is easy to obtain in discussions with domain experts. Nevertheless, in absence of any domain knowledge, certain heuristics could be applied that choose start and end activities based on frequencies, but these are outside the scope of this paper. That is, for the extension of IM that is described in this section, it is simply assumed that reliability is available.

4.3 Discovering process models from partial traces

We introduce an extension of Inductive Miner (IM) to handle partial traces, namely *Inductive Miner - partial traces* (IMpt)

As seen in the example of Figure 3, partial traces may introduce skips in the resulting process model. For each of the steps of IM, we describe the effects of partial traces, and briefly how IMpt addresses them.

IM detects cuts by considering the directly follows graph of the event log. The nodes of this graph are the activities in the log, and the edges denote the activities that were directly followed by other activities in the event log. Furthermore, directly follows graphs contain information about which activities where observed in the log as the start or end of a trace.

As an example, we consider an event log L_l:

$$L_l = [\langle a,b,c,a,b,c,a,b\rangle, \langle a,b,c,a,b,c|, |b,c,a,b\rangle, |c,a|]$$

The directly follows graph of L_l, without considering partial traces, is shown in Figure 4a.

In the directly follows graph, IM identifies characteristic footprints of the process tree operators $\times$, $\rightarrow$, $\wedge$ and $\circlearrowleft$. For instance, in Figure 4a, the cut $(\wedge, \{a,b\}, \{c\})$ can be identified, as a and b are fully connected to c. For more details on cut detection, please refer to [14]. As a final result, IM would discover the process tree $\wedge(c, \circlearrowleft(a,\tau), \circlearrowleft(\tau,b))$.

However, with the available knowledge of partial traces, this tree does not do L_l justice. To take partial traces into account, IMpt considers only reliable start and end activities. For L_l, the directly follows graph then becomes as shown in Figure 4b. In this graph, the cut $(\circlearrowleft, \{a,b\}, \{c\})$ can be identified. As a final result, IMpt would discover the process tree $\circlearrowleft(\rightarrow(a,b),c)$, which matches the intuitive idea of the log better than the model discovered by IM.

$$\begin{tikzcd} \rightleftarrows a \arrow[rr] \arrow[dr, shift left] & & b \leftrightarrows \arrow[dl, shift left] \\ & c \leftrightarrows \arrow[ul, shift left] \arrow[ur, shift left] & \end{tikzcd}$$

(a) Without considering partial traces.

$$\begin{tikzcd} \rightharpoonup a \arrow[rr] \arrow[dr, shift left] & & b \rightharpoondown \arrow[dl, shift left] \\ & c \arrow[ul, shift left] \arrow[ur, shift left] & \end{tikzcd}$$

(b) Considering partial traces.

Fig. 4. Two directly follows graphs for L_1.

After a cut has been detected, the event log is split into several sub-logs, based on the cut that was found. During log splitting, information about the reliability of traces has to be copied to the sub-logs and adjusted.

For $\times$ and $\wedge$, no adjustments are necessary. For instance, for $\wedge$, if the trace had an unreliable end, then, both sub-traces have unreliable ends. That is: $\langle a,b,c|$ split on $(\wedge,\{a,b\},\{c\})$ becomes $\langle a,b|$ and $\langle c|$.

For $\rightarrow$ and $\circlearrowleft$, if the to-be split trace has an unreliable end, the *last* sub-trace will have an unreliable end, but all other sub-traces will have reliable ends. For instance, $\langle a,b|$ split on $(\rightarrow,\{a\},\{b\},\{c\})$ becomes $\langle a\rangle$ for $\{a\}$; and $\langle b|$ for $\{b\}$; and no trace for $\{c\}$.[2] For unreliable starts of traces and for $\circlearrowleft$-cuts, a similar strategy is applied.

Most base cases of IM are unaffected by partial traces. However, if the log contains empty traces, then the base case EMPTYTRACES [13, p195] might remove the empty traces from the log, recurse and return a $\times(\tau,.)$ construct. Like other traces, empty traces might have unreliable starts or ends. If a trace had an unreliable start, then the actual trace might have events that fell before the window of observation (similar for unreliable ends). Therefore, IMpt considers empty traces only if these have a reliable start and a reliable end.

The concepts of IMpt can be straightforwardly extended to handle infrequent behavior (analogous to IM–infrequent [16], which filters noise from the directly follows graph before cut detection and from the log during log splitting), yielding Inductive Miner–infrequent–partial traces (IMfpt), and to handle incomplete behavior (analogous to Inductive Miner–incompleteness [17], which optimises to find the best cut rather than a perfect cut), yielding Inductive Miner–incompleteness–partial traces (IMcpt). IMpt, IMfpt and IMcpt have been implemented as plug-ins of the ProM framework [30] and their source code is publicly available.

5 PROCESS TREE TRANSFORMATION

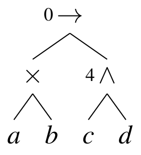
In this section, we present a method for finding a sequence of edit operations with the minimum cost, that transforms the pre-drift process tree P to the post-drift process tree P'. We first define a set of process tree edit operations and the cost of applying them in Section 5.1. A direct approach to solve the process tree transformation problem is then to try all possible sequences of edit operations that transform P into P' and find the cheapest one. However, there are infinite number of such sequences and it may be impossible to enumerate all of them. To prune the search space, we define a notion of *mapping* between two process trees, where a *valid mapping* is one that represents a sequence of edit operations that transforms P into P'. By defining the cost of a valid mapping based on the cost of edit operations, we reformulate our goal as to find a minimum-cost valid mapping between P and P'. By means of a mapping we substantially prune the search space as we only need to try all possible valid mappings between P and P' to find a minimum-cost sequence of edit operations that transforms P into P'. In Section 5.2.1 we present an A* algorithm to compute a minimum-cost valid mapping

[2] If the trace would have a reliable end, then an empty trace would be introduced in the sub-log for $\{c\}$.

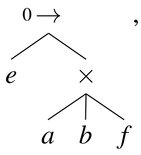
between two process trees. As a faster alternative, a greedy algorithm is presented in Section 5.2.2 to approximate such a mapping.

5.1 Process tree edit operations

A *process tree edit operation* is an edit operation applied to a process tree at any step during its transformation to another process tree. In a process tree transformation problem the goal is to find a minimum-cost sequence of edit operations to transform one process tree into another process tree (optimal solution). Hence, the granularity of process tree changes expressed in the optimal solution depends on the size of process tree constructs based on which the edit operations are defined as well as the cost of each edit operation. For example, consider the transformation of process tree P :



into process tree P' :



, and assume two edit operations, delete/insert a fragment (of any size), where each edit operation has a unit cost. These two edit operations yield the optimal solution consisting of two changes: delete $P\langle P[0]\rangle$-fragment and insert $P'\langle P'[0]\rangle$-fragment, i.e. delete the original process tree and insert the new process tree. However, such an abstract explanation does not provide any detail on the actual changes occurred in the process. On the other hand, assume two edit operations with unit costs which only allow the insertion/deletion of individual nodes in a process tree. For P and P' in the above example, the optimal solution would become: delete activities c and d, and insert activities e and f. This sequence of changes provide detailed characterization of changes in the process trees. However, explaining the changes in the level of activities can become verbose and confusing, specially when changes involve large fragments of a process. As such, we need to define edit operations and their costs such that the optimal solution characterizes process tree changes in enough detail while avoiding verbosity. For example, instead of reporting on the deletion of activities c and d individually, we could report on the deletion of the $P\langle P[4]\rangle$-fragment containing those activities without loss of information.

A process tree edit operation represents a change in its underlying process. Therefore, we define process tree edit operations based on the typical change patterns in business processes outlined in Table 1, obtained from [33]. We classify each change patterns, except "synchronize two fragments", as *simple (S)* or *compound (C)*, where a compound change pattern is one that can be expressed using multiple simple change patterns. Note that the synchronization of two fragments introduces unstructuredness into a process and hence cannot be used as a basis for defining process tree edit operations. This change pattern is illustrated with an example in Section 6. We set our goal as to find a sequence of simple changes that fully explains the transformation of the pre-drift process tree P to the post-drift process tree P', while satisfying three requirements:

(1) To improve the understandability of the changes, a change in the relation between fragments, e.g. from sequential to parallel, should only involve fragments that exist in both P and P'.
(2) The changes within the sequence should not overlap, i.e. any two changes should cover distinct differences between the trees.
(3) The sequence of changes needs to be detailed yet concise. That is, if a change involves an individual activity within the process tree then it should be explained by one change in the sequence referring to that activity. On the other hand, if a change involves a fragment of multiple activities then it should be explained by one change in the sequence referring to that fragment as a whole.

To satisfy these requirements, we first define a set of process tree edit operations based on the simple change patterns in Definitions 6, 7, 8 and 9. The defined edit operations can be applied to fragments of any size, from individual activities to larger fragments. We then search for a minimum-cost sequence of edit operations which transforms the pre-drift process-tree P into the post-drift process tree P'. In this search, we only consider sequences of edit operations in which edit operations that delete (resp., insert) fragments occur before (resp., after) edit operations that change the relation between fragments. Furthermore, by limiting each node within P or P' to be subject to one edit operation we ensure that the edit operations within a sequence do not overlap. We also define the cost of edit operations such that a minimum-cost sequence of edit operations which transforms P into P' provides a detailed description of changes within P. In a post-processing step, we then aggregate the edit operations within a sequence to make it as concise as possible.

In light of the above, based on a defined set of edit operations our goal is to find a minimum-cost sequence of edit operations to transform P into P' and to subsequently make it as concise as possible. In this paper, we use six process tree edit operations: substitution of operators $SUB_{\oplus}$, substitution of activities SUB_{ac}, deletion of fragments (D_f), deletion of $\circlearrowleft$-operator nodes ($D_{\circlearrowleft}$), insertion of fragments (I_f), and insertion of $\circlearrowleft$-operator nodes ($I_{\circlearrowleft}$). The relation with the change patterns is shown in Table 1.

After the application of each edit operation to a process tree, we reduce the resulting tree to normal form by repeatedly applying the reduction rules (cf. Definition 4). We do not report on the changes in a process tree as a result of the application of reduction rules, as they do not change the language of the tree. As such, we also do not associate any cost with the application of these reduction rules. Example 5 shows a sample process tree reduction.

Code	Change pattern	Cat.	Class	Process tree edit operations
sre	Insert/delete a fragment between two fragments	I	S	I_f, D_f
pre	Insert/delete a fragment in/from parallel branch	I	S	I_f, D_f
cre	Insert/delete a fragment in/from conditional branch	I	S	I_f, D_f
cp	Duplicate a fragment	I	C	
rp	Substitute a fragment	I	C	SUB_{ac} (covers activity substitution)
sw	Swap two fragments	I	C	
sm	Move a fragment to between two fragments	I	C	
pm	Move a fragment into/out of parallel branch	I	C	
cm	Move a fragment into/out of conditional branch	I	C	
cf	Make fragments mutually exclusive/sequential	R	S	$SUB_{\oplus}$
pl	Make fragments parallel/sequential	R	S	$SUB_{\oplus}$
cd	Synchronize two fragments	R	-	-
lp	Make a fragment loopable/non-loopable	O	S	$I_{\circlearrowleft}, D_{\circlearrowleft}$
cb	Make a fragment skippable/non-skippable	O	S	I_f, D_f
fr	Change branching frequency	O	-	

Table 1. Change patterns from [33] and their relation to our process tree edit operations.

Definition 6 (Process tree edit operations). *A process tree edit operation γ transforms a canonical process tree P into another canonical process tree P', denoted by $P \xrightarrow{\gamma} P'$.* □

Definition 7 (Substitution operations). *We use the following process tree edit operations for substitution:*

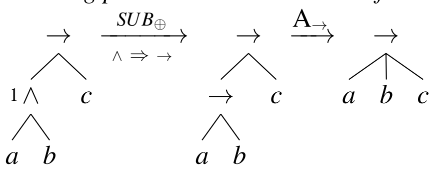
$SUB_{\oplus}$ ***Operator substitution*** *Let $\oplus(M_1, \dots M2)$ be a fragment, where $\dots$ is any number of process trees (possibly 0), $l(\oplus) \in \{\rightarrow, \times, \wedge\}$, and M_1 and M_2 are process trees. Operator substitution replaces the operator of $\oplus$ with a different operator in $\{\rightarrow, \times, \wedge\}$.*
This edit operation cannot be applied to an $\times(\dots, \tau)$-node, where $\dots$ is any number of process trees, as a $\rightarrow$- or $\wedge$-node may not have a τ-child.

SUB_{ac} ***Activity substitution*** *Applies to activity nodes, where it replaces the activity with a different activity.*

□

EXAMPLE 4. *Figure 5a shows two examples of substitution operations, where the operator of the* $\rightarrow(b, \times(c,d))$*-node is substituted with* $\wedge$*, and activity 'a' is substituted with activity 'e'.*

EXAMPLE 5. *As discussed earlier, after the application of each edit operation, we reduce the resulting process tree to its normal form by repeatedly applying the reduction rules. For instance, in*



after the substitution of the operator of the $\wedge$*-node 1 with* $\rightarrow$*, we can reduce the resulting process tree by applying the associativity reduction rule* $A_{\rightarrow}$.

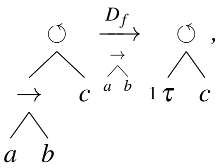
DEFINITION 8 (DELETION OPERATIONS). *We use the following process tree edit operations for deletion:*

D_f ***Fragment deletion*** *Deletes a fragment f.*
If f is a sub-fragment of an operator node $\oplus$ and as a result of deleting it $\oplus$ is left with one child, $\oplus$ will be removed by the singularity reduction rule. For example, in Figure 5b (left to right) after the deletion of Fragment 1 the $\wedge$*-node $P[1]$ is deleted subsequently by the singularity reduction rule* (S)*.*
If a $\circlearrowleft$*-node with less than two children remains after applying a fragment deletion, the deleted construct is replaced with a* τ*-child to keep the number of children of the* $\circlearrowleft$*-node at two. Such* τ*-nodes are called* auxiliary τ-nodes.

$D_{\circlearrowleft}$ $\circlearrowleft$***-operator deletion*** *Let $P = \oplus(\ldots, w(M_1, \tau), \ldots)$, where $\ldots$ is any number of process trees (possibly 0), w is a* $\circlearrowleft$*-node, and M_1 is a process tree. Deletion of the* $\circlearrowleft$*-node w makes $\oplus$ the parent of M_1 and deletes the* τ*-node.*

□

EXAMPLE 6. *Figure 5b (left to right) shows an example of D_f, where Fragment 1 is deleted. In*



, the deleted $\rightarrow$*-fragment is replaced by the* τ*-node 1 to keep the number of children of the* $\circlearrowleft$*-node at two. Figure 5c (left to right) shows an example of* $D_{\circlearrowleft}$*, where the* $\circlearrowleft$*-node $P[2]$ is deleted.*

DEFINITION 9 (INSERTION OPERATIONS). *We use the following process tree edit operations for insertion:*

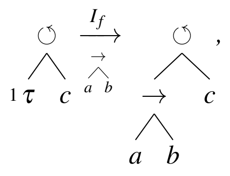
I_f ***Fragment insertion*** *Inserts a fragment (as a child of an operator node or an auxiliary operator node).*
As discussed above, the deletion of a fragment may cause its parent to be deleted as well by the singularity reduction rule. Thus, the fragment insertion operation needs to insert auxiliary operator *nodes again, to ensure that a fragment insertion can offset a fragment deletion. An* auxiliary operator node *is an extra non-*$\circlearrowleft$*-operator node, inserted (as a child of an operator node $\oplus$) in a process tree P (and) as the parent of an inserted fragment and a sub-fragment (of*

$\oplus$) in P. An auxiliary operator node defines the relation between the inserted fragment and the sub-fragment.
As explained before, the deletion operations insert τ-leaves if a $\circlearrowleft$-node would, as a result of the deletion, not have two children. Similarly, when inserting a fragment as the first child of a $\circlearrowleft(\tau, M_1)$-node, the τ-node is replaced (and similar for the symmetric second-child case). Such τ-nodes that are inserted (resp., deleted) as a result of deleting (resp., inserting) child fragments of $\circlearrowleft$-nodes are called auxiliary τ-nodes.

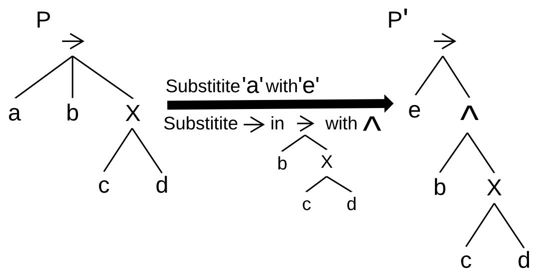
$I_{\circlearrowleft}$ ***$\circlearrowleft$-operator insertion*** *Inserts a $\circlearrowleft$-node n in a process tree P. As a result of this edit operation, one of the non-τ-sub-fragments of P is inserted as the first child (loop body) of n, while the second child (loopback) of n is a τ-node.*

□

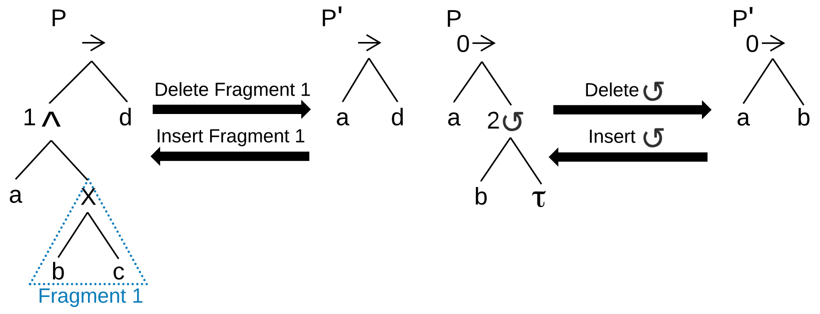
Example 7. *Figure 5b (right to left) shows an example of I_f, where Fragment 1 is inserted as a child of the auxiliary $\wedge$-node $P[1]$, in a concurrent relation with activity 'a'. In*



the τ-node 1 is replaced by the inserted $\rightarrow$-fragment to keep the number of children of the $\circlearrowleft$-node at two. Figure 5c (right to left) shows an example of $I_{\circlearrowleft}$, where the $\circlearrowleft$-node $P[2]$ is inserted as a child of the $\rightarrow$-node $P'[0]$ ($P[0]$), and as the parent of activity 'b'.



(a) SUB_{ac} and $SUB_{\oplus}$



(b) D_f (left to right) and I_f (right to left) (c) $D_{\circlearrowleft}$ (left to right) and $I_{\circlearrowleft}$ (right to left)

Fig. 5. Examples of process tree edit operations.

We defined a set of 6 edit operations based on the simple change patterns in Table 1, which allow to provide detailed characterization of changes in a process tree. Included in this set are the two edit operations, insert/delete a fragment, which alone suffice for explaining any types of changes in the structure of a process tree. Therefore, the set of 6 edit operations defined above is complete, i.e. it is possible to characterize any changes in the structure of a process tree using the edit operations in this set.

Each operation has an associated cost θ; these costs are shown in Table 2. In later uses (Lemma 2), we will use the triangle inequality for our cost function θ. That is, we need to show that:

LEMMA 1. *For all process trees w, u and v and all edit operations x, y and z it holds that $\theta(w \xrightarrow{z} u) \leq \theta(w \xrightarrow{x} v) + \theta(v \xrightarrow{y} u)$.*

PROOF. As shown in Table 2, the cost of an edit operation is at least 1 as the cost of a substitution and the deletion/insertion of a τ-node or a $\circlearrowleft$-node is exactly 1, and the cost of a non-τ fragment deletion/insertion being the number of non-τ nodes in that fragment. We show the triangle inequality by case distinction on the type of z, which can be either a substitution, an insertion, or a deletion:

z is a substitution As defined in Definition 7, a substitution operation can only be applied to an individual node with a cost of 1. As the cost of any edit operation is at least 1, regardless of their types, the sum of the costs of the edit operations x and y is higher than 1.

z is a deletion If $\xrightarrow{z}$ is a τ- or a $\circlearrowleft$-node deletion then its cost is 1. As the cost of an edit operation is at least 1, regardless of their types, the sum of the costs of the edit operations x and y is higher than 1.

If z deletes a fragment f, its cost is the number of non-τ leaves (activities) in f. Any combination of two edit operations x and y resulting in the deletion of fragment f requires the removal of each of its activities. Consequently, as the cost of any modification to an activity node (substitution/deletion/insertion) is 1, the sum of the costs of these two edit operations is at least equal to the number of activities in f, which is the cost of z.

z is an insertion Similar to the deletion case.

□

Edit operation	**Cost θ**
$SUB_{\oplus}$	1
SUB_{ac}	1
D_f	If the deleted fragment is a τ-node, then 1. Otherwise, the number of non-τ leaves in the fragment. Auxiliary nodes have no cost.
$D_{\circlearrowleft}$	1
I_f	If the inserted fragment is a τ-node, then 1. Otherwise, the number of non-τ leaves in the fragment. Auxiliary nodes have no cost.
$I_{\circlearrowleft}$	1

Table 2. Costs associated with the process tree edit operations.

5.1.1 Edit operation sequences. As all edit operations transform a process tree into another valid process tree according to Definition 2, we can define:

DEFINITION 10 (SEQUENCE OF EDIT OPERATIONS). *Let $e_1, \ldots e_n$ be edit operations and let P and P' be process trees. Then $S = e_1, \ldots, e_n$ is a sequence of edit operations for P and P' if it transforms P into P'. That is, there is a sequence of process trees $P_0, \ldots, P_n$ such that $P = P_0$, $P' = P_n$, and $P_{i-1} \xrightarrow{e_i} P_i$ for $1 \leq i \leq n$.*

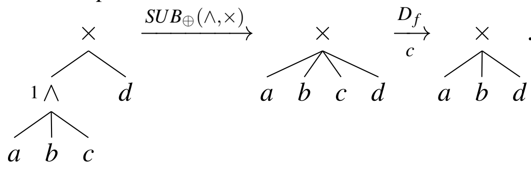
By extending θ the cost of the sequence S is given by $\theta(S) = \sum_{i=1}^{n} \theta(e_i)$.

The edit distance $d(P, P')$ from process tree P to process tree P' is defined to be the minimum cost of all sequences of edit operations which transform P into P', i.e.

$$d(P,P') = min\{\theta(S) \mid S \text{ is a sequence of edit operations which transforms } P \text{ into } P'\}$$

As stated in Section 5.1, to improve the understandability of changes within a sequence of edit operations, it should only be allowed to change the relation between fragments that exist in both P and P'. This is illustrated in the following example.

EXAMPLE 8. *As an example, consider this sequence of edit operations that transforms process tree P into process tree P',*



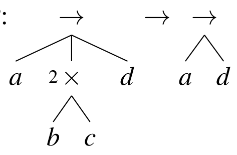
. First the operator of the $\wedge$-node 1 is substituted with $\times$ by a $SUB_{\oplus}$ edit operation, followed by the application of the reduction rule $A_{\times}$ to this node. Then, activity 'c' is deleted by a D_f edit operation. The first edit operation describes that the relation between activities 'a', 'b' and 'c' has changed from concurrent in P to mutually exclusive in P'. However, as activity 'c' is deleted by the subsequent edit operation, and hence does not exist in P', describing a change in the relation between this activity and other activities may be misleading. This problem can be avoided by applying fragment deletion operations before and symmetrically fragment insertion operations after other operations in a sequence of edit operations. In the above example, this could be achieved by reversing the order of the two edit operations.

As such, we define the following condition for a sequence of edit operations.

DEFINITION 11 (FRAGMENT DELETION/INSERTION ORDER). *Let S be a sequence of edit operations that transforms a process tree P into a process tree P'. It should hold that fragment deletion operations precede and fragment insertion operations follow other operations in S.*

□

Given a sequence of edit operations that transforms P into P', we aggregate the edit operations of the sequence as much as possible to obtain a *concise* sequence of edit operations. For example, in P:



the minimum-cost sequence of edit operations $\{D_f(b), D_f(c)\}$ can be reduced to sequence $\{D_f(P\langle P[2]\rangle)\}$ by aggregating the two activity deletion operations into the deletion of the fragment containing these two activities.

In the remainder of this paper, unless otherwise indicated, a sequence of edit operations that transforms P into P' always refers to a concise sequence of edit operations.

5.1.2 Process Tree Mappings. There are infinite numbers of different sequences of edit operations that transform P into P'. Therefore, it may be impossible to enumerate all sequences and find the shortest one. In the next section, we define structures called *process tree mappings* to prune the search space further and solve this problem more efficiently. We adapt the mapping between ordered trees by Tai [26], a.k.a. *Tai mapping*, to work on process trees featuring both ordered and unordered nodes. Figure 6 illustrates a sample mapping between two process trees P and P'.

A dotted line connecting a node $n \in P$ to a node $m \in P'$ indicates that n is to be substituted with m if $l(n) \neq l(m)$, or remain unchanged. Each node in P that is not connected by a dotted line is to be deleted from P, whereas each node in P' not connected by a dotted line is to be inserted in P. To maintain the hierarchical structure of the trees we add two virtual nodes with the same label as the

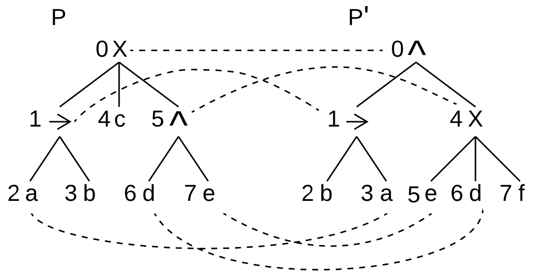


Fig. 6. Sample mapping between process trees P and P'.

roots of the trees and always map them to each other. Formally, a process tree mapping is defined as follows.

Definition 12 (Process tree mapping). *A process tree mapping between two process trees P and P' is defined by a triple $(M,\ P,\ P')$, where M is any set of pairs of integers $(i,\ j)$ satisfying the following conditions:*

1) *A pair $(i, j) \in M$, where $i \neq -1$ and $j \neq -1$, indicates that $P[i]$ needs to be substituted with $P'[j]$ if $l(P[i]) \neq l(P'[j])$; otherwise it remains unchanged. A pair $(i, -1) \in M$ indicates that the node $P[i]$ is to be deleted from P, whereas a pair $(-1,\ j) \in M$ indicates that the node $P'[j]$ is to be inserted in P:*
$$-1 \leq i \leq |P|-1 \wedge -1 \leq j \leq |P'|-1 \wedge (i \neq -1 \vee j \neq -1)$$
2) *Every node of P or P' is in the mapping:*
$$\forall_{0 \leq i_1 \leq |P|-1} \exists_{-1 \leq j_1 \leq |P|-1} (i_1, j_1) \in M \wedge \forall_{0 \leq j_1 \leq |P'|-1} \exists_{-1 \leq i_1 \leq |P|-1} (i_1,\ j_1) \in M$$
3) *Each node of P or P' is mapped at most once:*
$$\forall_{(i_1,\ j_1),(i_2,\ j_2) \in M \wedge (i_1 \neq -1 \vee i_2 \neq -1) \wedge (j_1 \neq -1 \vee j_2 \neq -1)} i_1 = i_2 \Leftrightarrow j_1 = j_2$$
4) *For every pair $(i \neq -1, j \neq -1) \in M$ the following conditions should hold:*
   a) *$P[i]$ is a non-$\circlearrowleft$ operator node iff $P'[j]$ is a non-$\circlearrowleft$ operator node.*
   b) *$P[i]$ is a $\circlearrowleft$-node iff $P'[j]$ is a $\circlearrowleft$-node.*
   c) *$P[i]$ is an activity node iff $P'[j]$ is an activity node.*
   d) *$P[i]$ is a τ-node iff $P'[j]$ is a τ-node.*
   e) *Any two mapped $\circlearrowleft$-nodes w in P and u in P', the nodes on the loopbody (resp. loopback) path of w can only be mapped to the nodes on the loopbody (resp. loopback) path of u: Let $w = P[r]$ and $u = P'[s]$ be ancestors of $P[i]$ and $P'[j]$ in P and P', such that $l(w) = l(u) = \circlearrowleft$ and $(r, s) \in M$, then $P[i]$ is on the loopbody path of w iff $P'[j]$ is on the loopbody path of u.*
5) *For every two pairs $(i_1 \neq -1, j_1 \neq -1), (i_2 \neq -1, j_2 \neq -1) \in M$ the following conditions should hold:*
   a) *$P[i_1]$ is an ancestor (resp., descendant) of $P[i_2]$ iff $P'[j_1]$ is an ancestor (resp., descendant) of $P'[j_2]$.*
   b) *Let w be a common* ordered *ancestor of $P[i_1]$ and $P[i_2]$ in P, and u be a common* ordered *ancestor of $P'[j_1]$ and $P'[j_2]$ in P',*
   if $Rank(P[i_1],\ w) < Rank(P[i_2],\ w)$ then $Rank(P'[j_1],\ u) \leq Rank(P'[j_2],\ u)$
   if $Rank(P[i_1],\ w) > Rank(P[i_2],\ w)$ then $Rank(P'[j_1],\ u) \geq Rank(P'[j_2],\ u)$
   if $Rank(P'[j_1],\ u) < Rank(P'[j_2],\ u)$ then $Rank(P[i_1],\ w) \leq Rank(P[i_2],\ w)$
   if $Rank(P'[j_1],\ u) > Rank(P'[j_2],\ u)$ then $Rank(P[i_1],\ w) \geq Rank(P[i_2],\ w)$

□

Condition 1 ensures that a node in P or P' is either mapped to a node in the other tree or to -1. Condition 2 and 3 ensure that every node in P or P' is exactly mapped once. Conditions 4a-4d ensure that M complies with the constraints of the substitution edit operations (cf. 7). Condition 4e ensures that for any two mapped $\circlearrowleft$-nodes w in P and u in P', respectively, the nodes on the loopbody (resp. loopback) path of w can only be mapped to the nodes on the loopbody (resp. loopback) path of u. For the sample mapping in Figure 6, $M = \{(0,\ 0), (1,\ 1), (2,\ 3), (3, -1), (4, -1), (5,\ 4), (6,\ 6), (7,\ 5), (-1,\ 2), (-1,\ 7)\}$.

Condition 5a in conjunction with the previous conditions are sufficient to ensure that after each touched node $P[i]$ is changed to its paired node $P'[j]$ (if $l(P[i]) \neq l(P'[j])$), untouched nodes of P are deleted and untouched nodes of P' are inserted in P, P and P' are equivalent provided that the two process trees only contain unordered operator nodes. However, as mentioned before, a process tree may contain both ordered and unordered operator nodes. Hence, we add condition 5b to preserve the order among siblings in both P and P'. For instance, for the two process trees P and P' in the sample mapping in Figure 6, two nodes $P[1]$ and $P'[1]$ are the only ordered nodes. Among descendants of $P[1]$, i.e. $\{P[2],\ P[3]\}$, and descendants of $P'[1]$, i.e. $\{P'[2],\ P'[3]\}$, $P[2]$ is mapped to $P'[3]$ in the mapping M, i.e. $(2,\ 3) \in M$. Consequently, $P[3]$ cannot be mapped to $P'[2]$ in M, i.e. $(3,\ 2) \notin M$, as otherwise condition 5b will be violated: $Rank(P[2],\ P[1]) (= 1) < Rank(P[3],\ P[1]) (= 2)$, but $Rank(P'[3],\ P'[1]) (= 2) \nleq Rank(P'[2],\ P'[1]) (= 1)$

To fully comply with the process tree edit operations and sequences thereof defined in Section 5.1, a mapping needs to satisfy further conditions. We call a mapping that satisfies those conditions a *valid process tree mapping* (valid mapping). The formal definition of a valid mapping is presented in Appendix A. In the remainder of this paper, unless otherwise indicated, a mapping always refers to a valid mapping.

To explain how each edit operation is represented in a mapping we define some notions. We provide some intuition here and refer to Appendix A for more details.

In a mapping M between two process trees P and P':

- A *deleted fragment* (resp., *inserted fragment*) is a fragment in P (resp., P') whose nodes are all deleted (resp., inserted). A *maximal deleted fragment* (resp., *maximal inserted fragment*) is a deleted (resp., inserted) fragment that is not a sub-fragment of a larger deleted (resp., inserted) fragment.
- An *auxiliary operator node* is a deleted (resp., inserted) non-$\circlearrowleft$-operator node with exactly one undeleted (resp., uninserted) child fragment. An auxiliary deleted operator node in a mapping corresponds to a node deleted by the singularity reduction rule after a fragment deletion edit operation (cf. Definition 8), whereas an auxiliary inserted operator node corresponds to an auxiliary operator node inserted along with a fragment insertion (cf. Definition 9).
- An *auxiliary τ-node* is a deleted (resp., inserted) τ-node parented by an undeleted (resp., uninserted) $\circlearrowleft$-node. An auxiliary deleted (resp., inserted) τ-node in a mapping corresponds to an auxiliary τ-node deleted (resp., inserted) as a result of inserting (resp., deleting) a child fragment of a $\circlearrowleft$-node by an edit operation D_f (resp., I_f) to keep the number of children of the $\circlearrowleft$-node at 2 (cf. Definitions 8 and 9).
- A *trivial operator node* is a deleted (resp., inserted) non-$\circlearrowleft$-operator node that has the same operator as its deepest undeleted (resp., uninserted) ancestor. A trivial deleted operator node corresponds to an operator node deleted by the associativity reduction rules (cf. Definition 4) after the application of an edit operation. Inversely, a trivial inserted operator node corresponds to an operator node inserted as the root of a sub-fragment of a non-$\circlearrowleft$-operator node as a result of applying an edit operation.

Table 3 illustrates how each edit operation is represented in a mapping.

Edit operation	**Representation in a mapping** $(M,\ P,\ P')$
$SUB_{\oplus}$	A non-$\circlearrowleft$-operator node $n \in P$ mapped to a non-$\circlearrowleft$-operator node $m \in P'$ such that $l(n) \neq l(m)$ or a non-auxiliary nontrivial deleted or inserted non-$\circlearrowleft$-operator node $n \notin$ a maximal deleted or inserted fragment.
SUB_{ac}	An activity node $n \in P$ mapped to an activity node $m \in P'$ such that $l(n) \neq l(m)$.
D_f	A maximal deleted fragment ($\neq$ trivial τ-node).
$D_{\circlearrowleft}$	A deleted $\circlearrowleft(M_1,M_2)$-node n, where M_1 is not a deleted fragment (i.e. $n \notin$ a maximal deleted fragment).
I_f	A maximal inserted fragment ($\neq$ trivial τ-node).
$I_{\circlearrowleft}$	An inserted $\circlearrowleft(M_1,M_2)$-node n, where M_1 is not an inserted fragment (i.e. $n \notin$ a maximal inserted fragment).

Table 3. Process tree edit operations (cf. Definition 6) and their representations in a mapping.

DEFINITION 13 (PROCESS TREE MAPPING COST). *Let M be a mapping between two process trees P to P', We define the cost of M as follows:*

$$\begin{gathered} cost(M) = \textit{total cost of all node substitutions} + \\ \textit{total cost of all maximal deleted and inserted fragments} + \\ \textit{total cost of all deleted and inserted } \circlearrowleft\textit{-nodes} \notin \textit{maximal deleted or inserted fragments} + \\ \textit{total cost of all deleted and inserted non-}\circlearrowleft \textit{ operator nodes} \notin \\ \textit{maximal deleted or inserted fragments.} \end{gathered}$$

We compute each of the first three costs in the same way as we did for the edit operations, while the cost of deleting or inserting a non-$\circlearrowleft$-operator node is 1. *Auxiliary or trivial nodes have no cost.*

□

In order to reduce the problem of finding a minimum-cost sequence of edit operations to finding a minimum-cost mapping, we first show that a minimum-cost sequence of edit operations and a minimum-cost mapping have the same cost:

LEMMA 2. *Let P and P' be process trees, M be a minimum-cost mapping between P and P' and let S be a minimum-cost sequence of edit operations transforming P into P' (Definition 10). Then, M and S have the same cost, i.e.* $\min_M cost(M) = \min_S \theta(S)$.

The proof of this lemma is similar to the proof of Theorem 3.1 in [26] and is omitted, that is, first the triangle inequality is shown to hold for mappings: two mappings can be combined to form a new mapping, and the cost of this combined mapping is at most the sum of the cost of the two mappings (using Lemma 1). Second, it is shown by induction that for any sequence of edit operations, there exists a mapping with an equivalent or lower cost. Then, it can be concluded that the minimum-cost mapping has the same cost as the minimum-cost sequence of edit operations.

A process tree mapping M can be described by a *corresponding* sequence of edit operations as follows:

- a D_f operation for each maximal deleted fragment (excluding trivial τ-nodes) in M;
- a SUB_{ac} operation for each activity node substitution in M;
- a $SUB_{\oplus}$ operation for each operator node substitution or non-auxiliary nontrivial deleted or inserted non-$\circlearrowleft$-operator node in M that is not in a maximal deleted or inserted fragment;
- a $D_{\circlearrowleft}$ operation for each deleted $\circlearrowleft$-operator in M that is not in a maximal deleted fragment;
- a $I_{\circlearrowleft}$ operation for each inserted $\circlearrowleft$-operator in M that is not in a maximal inserted fragment; and
- a I_f operation for each maximal inserted fragment (excluding trivial τ-nodes) in M.

In Section 5.3, we will elaboration on this translation. By construction, a mapping M and its corresponding sequence of edit operations S have the same cost. Thus, if M is a minimum-cost mapping, by Lemma 2, S is a minimum-cost sequence of edit operations.

Hence, the search for a minimum-cost sequence of edit operations has been reduced to a search for a minimum-cost mapping.

5.2 Finding process tree mappings & lower bounding function

In the next two sub-sections we present two algorithms for finding the minimum-cost mapping between two process trees, based on two different search strategies: exhaustive (A*) and greedy. Here we define a *mapping search tree* which is a data structure to capture the search space of the mapping.

DEFINITION 14 (MAPPING SEARCH TREE). *A mapping search tree between two process trees P and P', denoted by $MST(P,\ P')$, is a tree such that the label of the root is 0, the depth is $|P|-1$, and every internal node has a maximum of $|P'|$ children, each labeled by one of $-1,1,2,\ldots,|P|-1$.* □

We say that a node v in $MST(P,\ P')$ is valid if the following set M_v of pairs of integers forms a mapping between P and P':

$$\begin{aligned} M_v = & \{(dep(w),\ l(w)) \mid w \in Down_{MST(P,\ P')}(v)\} \cup \\ & \{(r,-1) \mid r \in \{dep(v)+1,\ldots,|P|-1\}\} \cup \\ & \{(-1,\ s) \mid s \in \{1,\ldots,|P'|-1\} - \{l(w) \mid w \in Down_{MST(P,\ P')}(v)\}\} \end{aligned}$$

In this paper, we refer to a mapping search tree as one consisting of just valid nodes. Hence, M_v is a mapping in which each node on the $Down_{MST(P,\ P')}(v)$ denotes the pair $(i,\ j)$, such that $i = dep(v)$ and $j = l(v)$. Every node m in P, with $r = pre_P(m)$, for which $r > dep(v)$ is deleted in M_v ($(r,-1) \in M_v$), and every node n in P', with $s = pre_{P'}(n)$, for which $s \notin \{l(w) \mid w \in Down_{MST(P,\ P')}(v)\}$ is inserted in M_v ($(-1,\ s) \in M_v$).

EXAMPLE 9. *Consider process trees P and P' in Figure 7a. Figure 7b illustrates the mapping search tree $MST(P,\ P')$. For example, the path $\langle 0,\ 2,-1\rangle$ in $MST(P,\ P')$ represents the mapping $\{(0,\ 0),\ (1,\ 2),\ (2,-1),\ (-1,\ 1)\}$ between P and P'. In this path, the node labeled with "2" does not have a child with the label "1", because the set of pairs $\{(0,\ 0),\ (1,\ 2)\}$, in compliance with the condition 5b of mapping, cannot form a mapping with the pair $(2,\ 1)$.*

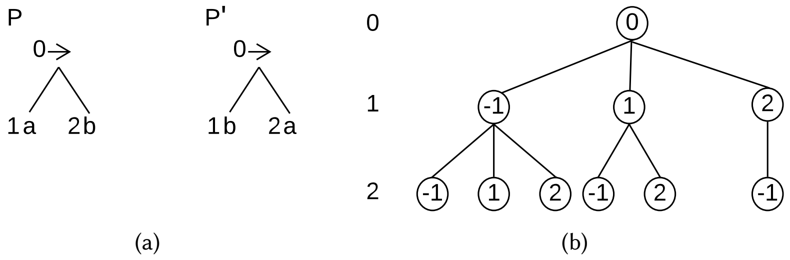


Fig. 7. Process trees P and P' (a) and their mapping search tree (b) in Example 9.

In a mapping M between P and P', each activity node in P can only be mapped to -1 or to an activity node in P', and vise versa (conditions 3 and 4c in Definition 12). Moreover, the cost of substituting, inserting or deleting an activity node in M is always 1, and the cost of mapping two activity nodes with the same label is 0. Therefore, it holds that the cost of M at least equals to the minimum-cost of mapping two sets of activity nodes under P and P'. For example, assume $C_1 =$

$\{a, b, c, d\}$ and $C_2 = \{a, c, e, f, g\}$ are the two sets of activity nodes under P and P', respectively. The cost of M at least equals to the minimum-cost of mapping C_1 and C_2, i.e. 3, obtained from the activity mapping set $S = \{(a,\ a), (b,\ e), (c,\ c), (d,\ f), (-1,\ g)\}$ between C_1 and C_2. Given two sets C_1 and C_2 of activity nodes, Algorithm 1 computes the minimum-cost of mapping C_1 and C_2.

Algorithm 1 Compute the minimum cost of mapping two sets of activity nodes

```
 1: procedure MINMAPPINGCOST($C_1$, $C_2$)
 2:     $cost \leftarrow 0$
 3:     for each $c \in C_1$ do
 4:         for each $d \in C_2$ do
 5:             if $l(c) = l(d)$ then
 6:                 $C_1 \leftarrow C_1 - \{c\}$
 7:                 $C_2 \leftarrow C_2 - \{d\}$
 8:                 break
 9:             end if
10:         end for
11:     end for
12:     $cost \leftarrow \min(|C_1|, |C_2|) + (||C_1| - |C_2||)$
13:     return $cost$
14: end procedure
```

For every node in C_1, Algorithm 1 iterates over all nodes in C_2 to find a node with the same label. If such a node is found it removes the two nodes from their respective sets (lines 3-8). After processing every node in C_1, there is no pair of nodes from C_1 and C_2 with the same label. The remaining nodes in C_1 and C_2 are then mapped injectively to each other, constituting $\min(|B_1|, |B_2|)$ mappings. Finally, the remaining $|B_1| - |B_2|$ nodes in C_1 or C_2 are mapped to -1.

5.2.1 Exhaustive search. In this section we introduce an A* algorithm that finds the minimal cost mapping between process trees P and P', by finding the cheapest path from the root to a leaf in the mapping search tree $MST(P,\ P')$. However, instead of constructing the whole $MST(P,\ P')$ our A* algorithm traverses P in a pre-order manner and only constructs nodes in $MST(P,\ P')$ that are potentially a part of the cheapest path to the leaves.

It is necessary to define two functions $g^*(v)$ and $h^*(v)$ for any instantiation of the A* algorithm. For a node $v \in MST(P,\ P')$, $g^*(v)$ determines the mapping cost up to v, whereas $h^*(v)$ estimates the cost of mapping the nodes that have not yet been mapped up to v. Let v be a node in $MST(P,\ P')$ such that $dep(v) = pre_P(w)$ and $l(v) = pre_{P'}(u)$ or $l(v) = -1$. Let Y_1 and Y_2 be the sets of activity nodes in P and P', respectively. Also, let $C_1 = C(w)$ and $C_2 = C(u)$ be the sets of activity nodes under w and u (if $l(v) \neq -1$), respectively. Furthermore, let P_m and P'_m be the sets of nodes in P and P', respectively, that are already mapped (either to a node in the other process tree or to -1). Then, $h^*(v)$ is defined as follows.

$$h^*(v) = \begin{cases} minMappingCost(C_1,\ C_2) + & \text{if } l(v) \neq -1 \\ \quad minMappingCost(Y_1 \setminus C_1 \setminus P_m,\ Y_2 \setminus C_2 \setminus P'_m) & \\ minMappingCost(Y_1 \setminus P_m,\ Y_2 \setminus P'_m) & \text{if } l(v) = -1 \end{cases}$$

To compute $g^*(v)$ we need to compute and sum the cost of every node substitution, node deletion, and node insertion induced by the mappings on the path from the root to v in $MST(P,\ P')$. However, as specified in Definition 13 for computing the cost of a mapping M between two process trees P and P', the cost of deletion or insertion of an auxiliary or trivial operator node in M is 0. Whether a deleted or inserted operator node is considered auxiliary or trivial in a mapping depends on how its descendants are mapped (cf. Appendix A). For example, a deleted operator node o is auxiliary in M if at most one of its child fragments is not deleted in M. Therefore, to determine if o is an auxiliary operator node we need to know how its descendants are mapped. However, as we construct a mapping search tree by traversing P in a pre-order manner, the descendants of o are still unmapped when computing the mapping cost at v. Thus, to enable computing the cost of a deleted or inserted operator node we assume a mapping of -1 for each of its descendant nodes that is not already mapped. However, this assumption is only to assist computing the cost of mapped operator nodes up to each node on a mapping search tree, and does not imply the deletion or the insertion of those unmapped descendant nodes. Furthermore, this assumption does not result in the overestimation of the mapping cost as it actually leads to the temporary consideration of a deleted or inserted operator node as an auxiliary or trivial operator node, with a cost of 0.

The A* algorithm computes the value of $f^*(v) = g^*(v) + h^*(v)$ for each node v in $MST(P, P')$, and at each step searches for the node with the lowest f^*. The A* algorithm for finding the lowest cost mapping between P and P' is given as Algorithm 2.

Algorithm 2 A*

```
1: procedure ASTAR($P$, $P'$)
2:     /* $MST(P, P')$ is a mapping search tree between $P$ and $P'$*/
3:     /*$L$ is a list of triples*/
4:     add the node $v$ labeled by 0 as the root to $MST(P, P')$
5:     while $dep(v) \neq |P| - 1$ do
6:         $i \leftarrow dep(v) + 1$
7:         add the node $u$ such that $l(u) = -1$ to $MST(P, P')$ as the child of $v$
8:         $L \leftarrow L \cup \{(u, g^*(u), h^*(u))\}$
9:         for each $w \in P'$ do
10:            if $(M_v - \{(i, -1)\}) \cup \{(i, pre_{P'}(w))\}$ forms a mapping between $P$ and $P'$ then
11:                add the node $u$ such that $l(u) = pre_{P'}(w)$ to $MST(P, P')$ as the child of $v$
12:                $L \leftarrow L \cup \{(u, g^*(u), h^*(u))\}$
13:            end if
14:        end for
15:        select $(v, g^*(v), h^*(v)) \in L$ such that $f^*(v)$ is minimum
16:        $L \leftarrow L - \{(v, g^*(v), h^*(v))\}$
17:    end while
18:    return $M_v$
19: end procedure
```

Algorithm 2 starts with constructing the root node v of $MST(P, P')$ from a mapping between two fake root nodes added to P and P' (line 4). These fake nodes have the same label, randomly selected from $\{\rightarrow, \times, \wedge\} \setminus (\{l(root(P)\} \cup \{l(root(P'))\})$. A list L holds child-free nodes in $MST(P, P')$. The algorithm proceeds with adding nodes to $MST(P, P')$ and selecting the node in L with the lowest cost at each step (lines 5-14). Each time a node is selected its subsequent node in the pre-order traverse of P is first deleted (mapped to -1) (lines 7-8), and then mapped to any node of P' that does not lead to the violation of mapping conditions (lines 9-12). At each iteration of the while loop the node v in L with the lowest f^* is selected (line 13). The while loop halts if v is a leaf of $MST(P, P')$. At the end, the algorithm outputs the mapping M_v, i.e. the minimum-cost mapping between P and P' (line 15).

EXAMPLE 10. *Consider process trees P and P' in Figure 8. Two fake $\times$-nodes are added as roots to P and P', to be mapped at the first step of the A* algorithm. We illustrate the run of the A* algorithm to construct the $MST(P, P')$ in Figure 9. Here, the index of a node in $MST(P, P')$ represents the value $g^*(v) + h^*(v)$ for that node. At each step, the node with the minimum cost in L, highlighted with gray, is selected and deleted from L. The children of this node are then added in the following step both to the $MST(P, P')$ and to L. Also, the mapping corresponding to the path from* *the root to this node is illustrated by dotted lines on the two trees on the left. Each number on the left side of the $MST(P,\ P')$ indicates the pre-order index of the node in P that is mapped to nodes in P' via the nodes in that depth of $MST(P,\ P')$.*

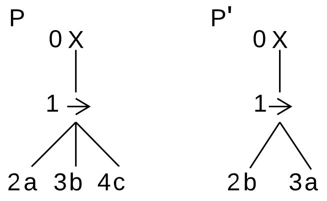
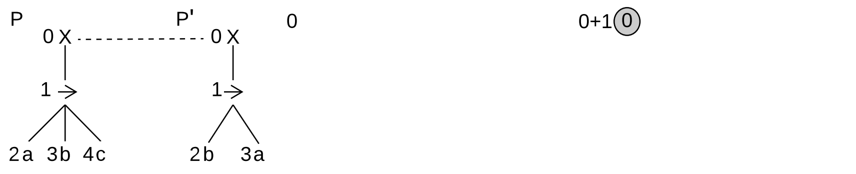


Fig. 8. Process trees P and P' in Example 10.

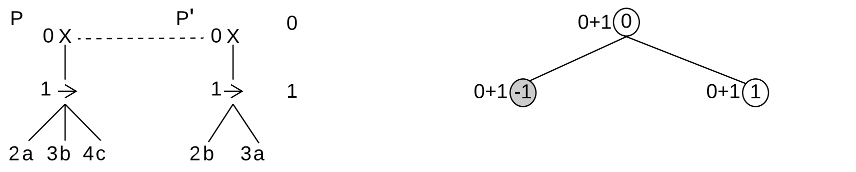
At the step (a) of the running example, the two added fake roots are mapped to each other, constructing the root of $MST(P,\ P')$. At the step (b), the two children of the root of $MST(P,\ P')$ are added by mapping the →-node in P, with the pre-order index of 1, *to* -1 *and to the →-node in P', with the pre-order index of* 1. *As a result of deleting the →-node in P (mapping to* -1*) the →-node in P' is also deleted, since there is no other operator node in P that can potentially be mapped to it. Here, the value of g^* for mapping the →-node in P to* -1 *is* 0. *Because, as explained before, the algorithm initially considers a deleted/inserted operator node as an auxiliary or trivial operator node if it does not know how its descendants are mapped. As these two added children nodes have the same cost one of them is randomly selected, here the mapping to* -1. *At the step (c), the node 'a' in P is mapped to* -1, *to 'b' and to 'a' in P', respectively. Among the existing nodes in L, the mapping of the →-node in P to the →-node in P' is selected as it has the lowest cost of* 1. *At the step (d), the children of this node are added, i.e. again mapping the node 'a' to* -1, *to 'b' and to 'a' in P', respectively, and the node with the lowest cost is selected. At the step (e), the algorithm maps the node 'b' in P to* -1, *to 'b' and to a in P', respectively, and selects the mapping to 'b' as the minimum cost node. At the step (f), the children of this node are added, i.e. mapping the node 'c' to* -1, *and to 'a' in P', respectively. Note that mapping the node 'c' to the node 'a' makes the deleted (resp., inserted) →-node in P (resp., P') on the path* $\langle 0,1,-1,3\rangle$ *non-auxiliary and non-trivial as it does not satisfy the conditions of an auxiliary or a trivial operator node anymore (cf. Appendix A). The minimum-cost node at this step is the mapping of 'a' to* -1 *on the path* $\langle 0,\ 1,-1\rangle$. *At the step (g), the node 'b' in P is again mapped to* -1, *and to 'b' and to 'a' in P', respectively, with the mapping to 'b' being the one with the minimum cost among all nodes. Finally, at the step (h), the node 'c' in P is mapped to* -1 *and the node 'a' in P', with the latter mapping forming the minimum-cost node. At this step the A^* algorithm terminates as the node v with the lowest mapping cost is a leaf in $MST(P,\ P')$. The minimum-cost path from the root to v is* $\langle 0,\ 1,-1,\ 2,\ 3\rangle$, *and the minimum-cost mapping between P and P' is* $M_v = \{(0,\ 0),\ (1,\ 1),\ (2,-1),\ (3,2),\ (4,3)\}$, *with the cost of* 2.

Time complexity. It is known that the problem of computing the tree edit distance between two unordered trees is NP-hard [34]. A process tree may contain unordered operator nodes, such as $\times$-nodes and $\wedge$-nodes as well as ordered operator nodes, such as →-nodes and ↻-nodes. Therefore, the problem of computing the minimum cost mapping between two process trees is also NP-hard.

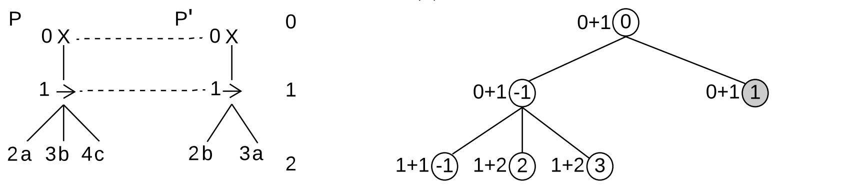
5.2.2 Greedy search. As mentioned in the previous section, the problem of computing the minimum cost mapping between two process trees is NP-hard. Consequently, depending on how different the two process trees are, the A^* algorithm may not be able to compute the optimal solution within a reasonable time. In this section, we introduce a more efficient algorithm based on a greedy heuristic.


P
P'
0 X
0 X
1 →
1 →
2a 3b 4c
2b 3a
0
0+1 0

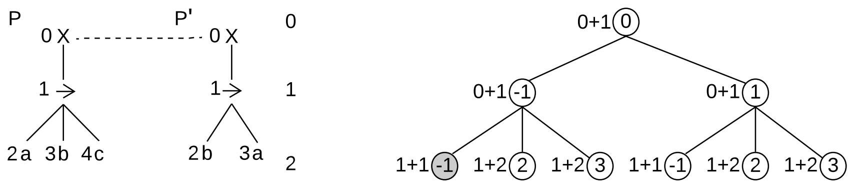
(a)


P
P'
0 X
0 X
1 →
1 →
2a 3b 4c
2b 3a
0
1
0+1 0
0+1 -1
0+1 1

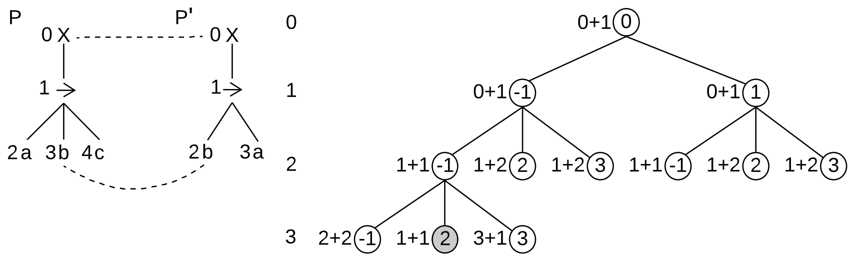
(b)


P
P'
0 X
0 X
1 →
1 →
2a 3b 4c
2b 3a
0
1
2
0+1 0
0+1 -1
0+1 1
1+1 -1
1+2 2
1+2 3

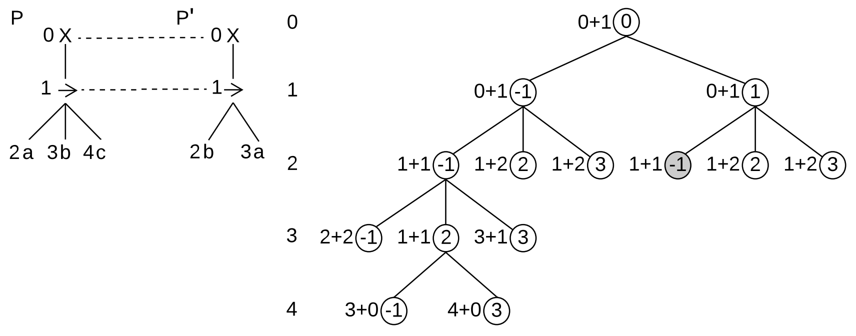
(c)


P
P'
0 X
0 X
1 →
1 →
2a 3b 4c
2b 3a
0
1
2
0+1 0
0+1 -1
0+1 1
1+1 -1
1+2 2
1+2 3
1+1 -1
1+2 2
1+2 3

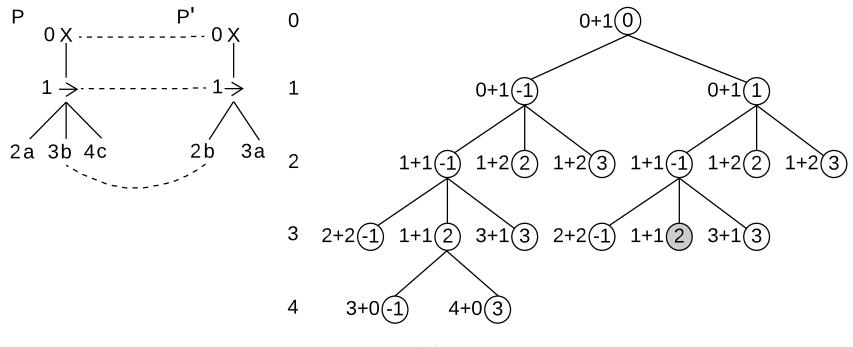
(d)


P
P'
0 X
0 X
1 →
1 →
2a 3b 4c
2b 3a
0
1
2
3
0+1 0
0+1 -1
0+1 1
1+1 -1
1+2 2
1+2 3
1+1 -1
1+2 2
1+2 3
2+2 -1
1+1 2
3+1 3

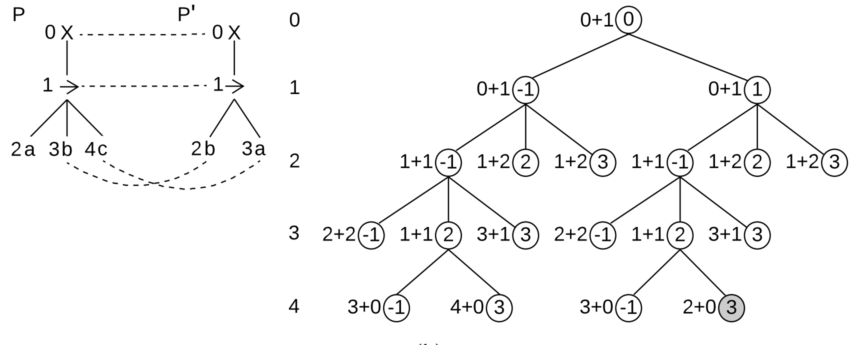
(e)


P
P'
0 X
0 X
1 →
1 →
2a 3b 4c
2b 3a
0
1
2
3
4
0+1 0
0+1 -1
0+1 1
1+1 -1
1+2 2
1+2 3
1+1 -1
1+2 2
1+2 3
2+2 -1
1+1 2
3+1 3
3+0 -1
4+0 3

(f)



(g)



(h)

Fig. 9. The running example of the A* algorithm for the process trees P and P' in Figure 8.

This algorithm has a lower theoretical complexity than the A* algorithm, though it cannot guarantee to find the optimal solution.

The Greedy algorithm is similar to the A* algorithm. The latter finds a mapping between P and P' that has the lowest global cost. As such, it may process each node in P multiple times. In contrast, the greedy algorithm selects and fixes a locally optimal mapping for each node in P at each step of constructing $MST(P,P')$. This is performed by clearing the list L containing the child-free nodes in $MST(P,P')$ at the end of each step (line 26). In addition, the greedy algorithm has a different strategy for mapping operator nodes. For every operator node y in P, this algorithm first finds an operator node y' in P' with the lowest mapping cost (lines 10-14). Next, it computes a matching score between y and y'. The matching score measures the similarity of activity nodes under y and y', and lies in the range $[0,1]$, where 0 indicates that there is no pair (a,b) of activities in $C(y) \times C(y')$ such that $l(a) = l(b)$, whereas 1 indicates $\{l(a) \mid a \in C(y)\} = \{l(b) \mid b \in C(y')\}$. The nodes y and y' are mapped if the matching score between them is above the threshold. However, in case there is no node in P' that can be mapped to y or if the matching score is bellow the threshold, y is mapped to -1, by selecting z as the optimal node in $MST(P,\ P')$ (lines 16-25).

Time complexity. Given two process trees P and P', the time complexity of the greedy algorithm is dominated by the complexity of the while loop (line 5). The complexity of this loop is the maximum of the worst-case complexity of three sequential steps. This loop iterates $|P|$ times. At each iteration, we first map a node $y \in P$ to every node in P' that satisfies the mapping conditions (lines 10-13), thus $O(|P| \cdot |P'|)$. We then select the mapping between y and a node in P' with the lowest cost (line 14). The latter step requires, in the worst case, iterating over $|P'|$ mappings, thus $O(|P| \cdot |P'|)$. If y is an

Algorithm 3 Greedy

```
 1: procedure GREEDY($P$, $P'$, $threshold$)
 2:     /* $MST(P, P')$ is a mapping search tree between $P$ and $P'$*/
 3:     /*$L$ is a list of triples*/
 4:     add the node $v$ labeled by 0 as the root to $MST(P, P')$
 5:     while $dep(v) \neq |P| - 1$ do
 6:         $i \leftarrow dep(v) + 1$
 7:         add the node $z$ such that $l(z) = -1$ to $MST(P, P')$ as the child of $v$
 8:         if $P[i]$ is not an operator node then
 9:             $L \leftarrow L \cup \{(z, g^*(z), h^*(z))\}$
10:         end if
11:         for each $w \in P'$ do
12:             if $(M_v - \{(i, -1)\}) \cup \{(i, pre_{P'}(w))\}$ forms a mapping between $P$ and $P'$ then
13:                 add the node $u$ such that $l(u) = pre_{P'}(w)$ to $MST(P, P')$ as the child of $v$.
14:                 $L \leftarrow L \cup \{(u, g^*(u), h^*(u))\}$
15:             end if
16:         end for
17:         select $(v, g^*(v), h^*(v)) \in L$ such that $f^*(v)$ is minimum
18:         $y \leftarrow P[i]$
19:         if $y$ is an operator node then
20:             if $v \neq null$ then
21:                 $y' \leftarrow P'[l(v)]$
22:                 $ubc \leftarrow max(|C(y)|, |C(y')|)$ /*Upper bound for the cost of mapping two activity
                                                 sets $C(y)$ and $C(y')$*/
23:                 $mmc \leftarrow minMappingCost(C(y), C(y'))$
24:                 $matchingScore \leftarrow (ubc - mmc)/ubc$
25:                 if $matchingScore < threshold$ then
26:                     $v \leftarrow z$
27:                 end if
28:             else
29:                 $v \leftarrow z$
30:             end if
31:         end if
32:         $L \leftarrow \emptyset$
33:     end while
34:     return $M_v$
35: end procedure
```

operator node for which we are able to find a mapping node $y' \in P'$, then we compute the minimum cost of mapping two sets of activity nodes under y and y' (line 20), thus $O(|P|^2 \cdot |P'|)$. Therefore, the worst-case complexity of the while loop and so that of the greedy algorithm is $O(|P|^2 \cdot |P'|)$.

5.3 From process tree mapping to sequence of edit operations

Finally, from a given mapping between two process trees P and P', we extract a concise sequence of edit operations that transforms P into P'. In that, we need to satisfy the fragment deletion/insertion order condition of sequence of edit operations (cf. Definition 11), that requires fragment deletions (resp., insertions) to precede (resp., to follow) other operations. Accordingly, we extract fragment deletions first and fragment insertions last from the mapping. After fragment deletions, we first extract activity substitutions. We then extract operator node deletions followed by operation node insertions and substitutions. To extract nested edit operations we need to traverse P and P' in opposite orders. Specifically, we process deleted fragments and deleted operator nodes by traversing P in a top-down order, while inserted fragments, inserted operator nodes, and substituted operator nodes are processed by traversing P' in a bottom-up order. Activity node substitutions may be processed in any order. We extract a concise sequence of edit operations from a mapping by performing the following steps in this order.

```
1) Fragment deletion
    for $i = 1$ to $dep(P)$ do
        Add a $D_f$ operation for every maximal deleted fragment ($\neq$ trivial $\tau$-node) rooted at depth $i$ of $P$.
    end for
2) Activity Substitution
    Add a $SUB_{ac}$ operation for every activity node substitution.
3) Operator Substitution, $\circlearrowleft$-Operator deletion, and $\circlearrowleft$-Operator insertion
    for $i = 1$ to $dep(P)$ do
        Add a $D_{\circlearrowleft}$ operation for every deleted $\circlearrowleft$-node at depth $i$ of $P$.
        Add a $SUB_{\oplus}$ operation for every non-auxiliary nontrivial deleted non-$\circlearrowleft$ operator node at depth $i$ of $P$.
    end for
    for $i = dep(P')$ to 1 do
        Add a $SUB_{\oplus}$ operation for every non-auxiliary nontrivial inserted non-$\circlearrowleft$ operator node at depth $i$ of $P'$.
        Add a $SUB_{\oplus}$ operation for every operator node substitution at depth $i$ of $P'$.
        Add an $I_{\circlearrowleft}$ operation for every inserted $\circlearrowleft$-node at depth $i$ of $P'$.
    end for
4) Fragment insertion
    for $i = dep(P')$ to 1 do
        Add an $I_f$ operation for every maximal inserted fragment ($\neq$ trivial $\tau$-node) rooted at depth $i$ of $P'$.
    end for
```

6 CONSTRUCT DRIFT CHARACTERIZATION STATEMENTS

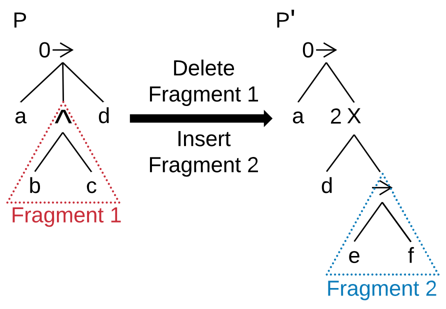
The output of the previous section is a sequence of edit operations that transforms a pre-drift process tree P into a post-drift process tree P'. In this section, we construct a sequence of characterization statements based on a given sequence of edit operations. As explained in Section 5.1, each edit operation describes a simple change in Table 1. By aggregating the simple changes obtained from a sequence of edit operations in a post-processing step we create compound changes (cf. Table 1). This further reduces the number of changes reported to the user and creates higher-level changes that are easier to interpret. Each remaining simple change and each created compound change is then reported to the user as a natural language statement.

6.1 Simple change patterns

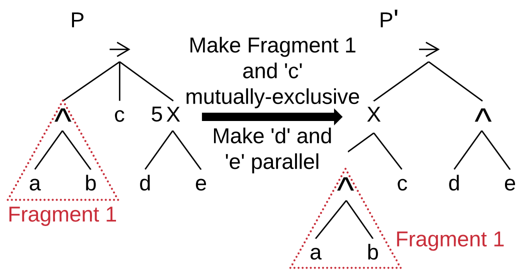
Here we describe how each simple change pattern is captured by an edit operation.

- **Insert/delete a fragment (sre, pre, cre)** The application of a D_f edit operation on a non-τ-fragment represents a fragment deletion, whereas the application of a I_f edit operation on a non-τ-fragment represents a fragment insertion. The fragment insertion/deletion is *serial (sre)*, *parallel (pre)*, or *conditional (cre)* if the parent node of the inserted/deleted fragment is a $\rightarrow$-node, a $\wedge$-node, or an $\times$-node, respectively. Also, the insertion/deletion of a fragment in/from the loopbody (resp., loopback) of a $\circlearrowleft$-node is considered as a serial (resp., conditional) fragment insertion/deletion. For example, in the transformation of P into P' in Figure 10a, Fragment 1 is deleted from between activity ‘a’ and activity ‘c’ (serial deletion), whereas Fragments 2 is inserted in a conditional branch with activity ‘d’.

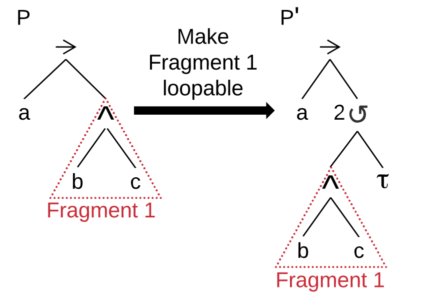
- **Make fragments mutually exclusive/parallel/sequential (cf, pl)** The application of a $SUB_{\oplus}$ operation on an operator node v changes the relation between child fragments of v. For example, in the transformation of P into P' in Figure 10b, Fragment 1 precedes activity ‘a’ in P, but, after the substitution of the operator of the $\rightarrow(\wedge(a,b),c)$-node with $\times$, they are mutually exclusive in P'. In addition to the change patterns cf and pl, the relation between two fragments can also be changed from mutually exclusive to parallel, and vice versa. However, this change pattern is not defined as one of the common change patterns in [33]. For example, in Figure 10b, activities ‘d’ and ‘e’ were mutually exclusive in P, but after the substitution of the operator of the $\times$-node $P[5]$ with the $\wedge$, they are parallel in P'.
- **Make a fragment loopable/non-loopable (lp)** The insertion (resp., deletion) of a $\circlearrowleft$-node by a $I_{\circlearrowleft}$ (resp., $D_{\circlearrowleft}$) edit operation as the parent of a fragment makes that fragment loopable (resp., non-loopable). For example, in the transformation of P into P' in Figure 10c, Fragment 1 in P has become loopable in P' with the insertion of the $\circlearrowleft$-node $P'[2]$.
- **Make a fragment skippable/non-skippable (cb)** The insertion (resp., deletion) of a τ-node by a I_f (resp., D_f) edit operation under an $\times$-node makes other child fragments of the $\times$-node skippable (resp., non-skippable). For example, in the transformation of P into P' in Figure 10d, with the insertion of the τ-node $P'[6]$, Fragment 1 has become skippable in P'.



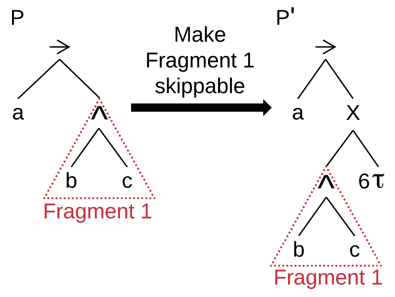
(a) Insert/delete a fragment.


P
→
∧
c
5 X
a
b
d
e
Fragment 1
Make Fragment 1 and 'c' mutually-exclusive
Make 'd' and 'e' parallel
P'
→
X
∧
c
d
e
∧
a
b
Fragment 1

(b) Make fragments mutually exclusive/parallel/sequential.



(c) Make a fragment loopable/non-loopable.



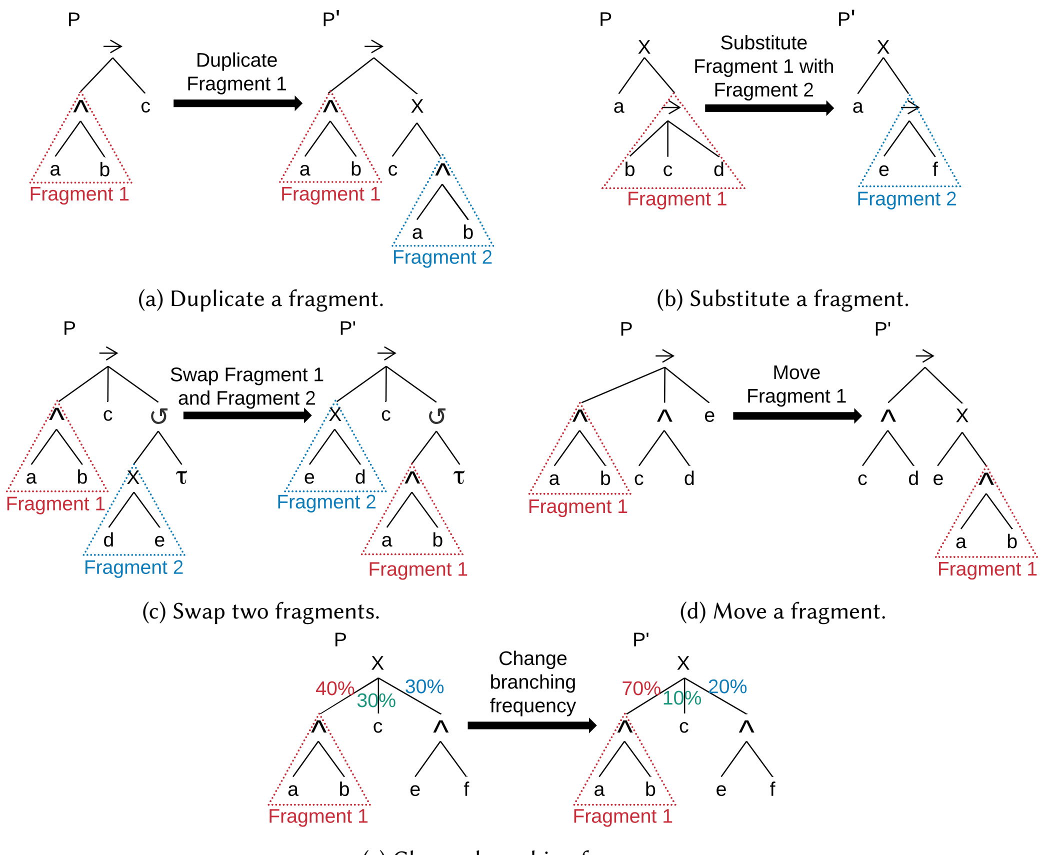
(d) Make a fragment skippable/non-skippable.

Fig. 10. Examples of transforming a process tree P into a process tree P' by the application of simple changes.

6.2 Compound change patterns

By aggregating simple change patterns we can construct compound change patterns, which allow us to provide a higher-level characterization of changes occurred in a drift. For example, the deletion of a fragment from a node in the source tree and its insertion in another node in the target tree can be summarized as a fragment move. We describe these compound patterns below.

- **Duplicate a fragment** An inserted fragment f_2 in a process tree is a duplicate fragment if there is another fragment f_1 in the tree, such that $\mathit{stringify}(f_2) = \mathit{stringify}(f_1)$, where *stringify* is a recursive function that converts a fragment to a unique and stable textual representation as defined in Appendix B. For example, in the transformation of P into P' in Figure 11a, Fragment 2 in P' is a duplicate of Fragment 1, since Fragment 2 is inserted, and $\mathit{stringify}(\mathit{Fragment}\ 2) = \mathit{stringify}(\mathit{Fragment}\ 1)$
- **Substitute a fragment (rp)** The application of a SUB_{ac} edit operation represents an activity substitution, and the application of a $SUB_{\oplus}$ edit operation represents an operator substitution. To discover a fragment substitution we need to abstract from the operator and the activity substitutions within the fragment. A fragment f in P is substituted by a fragment f' in P' if at least one node within f is substituted by a node within f', and every other node within f (resp., f') is either substituted by (resp., either substitutes) a node within f' (resp., f) or is deleted (resp., inserted). For example, in the transformation of P into P' in Figure 11b, fragment 1 in P is substituted with fragment 2 in P'.
- **Swap two fragments (sw)** In the transformation of P into P', two fragments f_1 and f_2 in P are swapped if they are substituted by two fragments f_1' and f_2' in P', respectively, such that $\mathit{stringify}(f_1) = \mathit{stringify}(f_2')$ and $\mathit{stringify}(f_2) = \mathit{stringify}(f_1')$. For example, in the transformation of P into P' in Figure 11c, Fragment 1 and Fragment 2 in P are swapped as they are substituted by Fragment 2 and Fragment 1 in P', respectively, and $\mathit{stringify}(\mathit{Fragment}\ 1_P) = \mathit{stringify}(\mathit{Fragment}\ 2_{P'}) =$ "*ab*" and $\mathit{stringify}(\mathit{Fragment}\ 2_P) = \mathit{stringify}(\mathit{Fragment}\ 1_{P'}) =$ "*bc*".
- **Move a fragment (sm, pm, cm)** The combination of deleting a fragment f from P and inserting a fragment f' in P such that $\mathit{stringify}(f) = \mathit{stringify}(f')$ represents a move of the fragment f within P. The fragment move is *serial (sre)*, *parallel (pre)*, or *conditional (cre)* if the parent node of the inserted fragment f' is a $\rightarrow$-node, a $\wedge$-node, or an $\times$-node, respectively. For example, in the transformation of P into P' in Figure 11d, Fragment 1 has moved to a conditional branch with activity 'e'.
- **Change branching frequency (fr)** In the transformation of P into P', let $v \in P$ be an $\times$-node with no deleted or inserted children. Also let c be a child fragment of v that is not substituted by another fragment. We define the relative frequency of c as the ratio between the frequency of c and the frequency of v, and express it as a percentage by multiplying it by 100. A significant change in the relative frequency of c over the transformation of P into P' represents a change of branching frequency. Let *freqB* and *freqA* be the relative frequencies of c in P and P', respectively. We compute the relative frequency change of c by $|freqB - freqA| * 100 / ((freqB + freqA)/2)$. The significance of the relative frequency change can be defined by the user. To focus on more significant branching frequency changes, in the evaluation sections of this paper we consider a relative frequency change of at least 50% as a significant change, where the relative frequency of the fragment is at least 25% in P or P'. To compute the frequency of nodes in a process tree we replay its underlying event log on top of the process tree. For example, in the transformation of P into P' in Figure 11e, the relative frequency of Fragment 1 has changed from 40% in P to 70% in P'.


P
P'
Duplicate Fragment 1
Fragment 1
Fragment 2
Substitute Fragment 1 with Fragment 2
Swap Fragment 1 and Fragment 2
Move Fragment 1
Change branching frequency
40%
30%
30%
70%
10%
20%

(a) Duplicate a fragment.

(b) Substitute a fragment.

(c) Swap two fragments.

(d) Move a fragment.

(e) Change branching frequency.

Fig. 11. Examples of transforming a process tree P into a process tree P' by the application of compound changes.

6.3 Nested changes

Multiple changes that involve the same behavioral relation between activities, e.g. causality or concurrency, are called *overlapping changes* [23]. For example, in the transformation of the process tree P to the process tree P' in Figure 12, there are two overlapping changes: 1. activity ‘b’ is deleted, 2. the relation between activity ‘c’ and activity ‘d’ changes from sequential in the left process tree to mutually exclusive in the right process tree. When applied in isolation, these two changes share the causal relation $b \rightarrow c$. The application of the first change deletes this behavioral relation, while the application of the second change decreases the frequency of its execution. By abstracting from the low-level behavioral relations between activities, IM discovers relations between fragments within a process tree. Consequently, overlapping changes are isolated from each other, and can be characterized in the same way as the non-overlapping ones. For example in Figure 12, in P, activity ‘b’ precedes the fragment $\rightarrow(c,d)$, whereas in P', activity ‘b’ is deleted and the operator of the root of the fragment $\rightarrow(c,d)$ is substituted with $\times$, resulting in the fragment $\times(c,d)$.

Nested changes are overlapping changes where each change is applied to the process subtree resulting from the application of a previous change. The hierarchical structure of a process tree allows us to characterize the changes applied to the inner structure of a fragment and those applied to

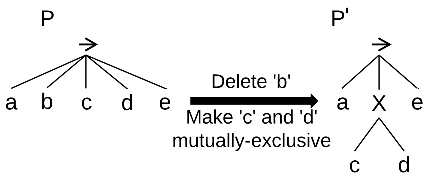


Fig. 12. An example of transforming a process tree P into a process tree P' via the application of overlapping changes.

the fragment as a whole, independently of each other. In 5.3, we explained in what order we traverse process trees to extract a sequence of edit operations from a mapping. We apply the changes in the same order as we extracted them, to transform a process tree P into a process tree P'. For example, to transform the process tree P into the process tree P' in Figure 13, we first make activity ‘b’ and activity ‘c’ sequential, resulting in the fragment $\rightarrow(b,c)$. Next, a loop structure is placed over this fragment, by inserting the $\circlearrowleft$-node $P^{tt}[2]$. Finally, activity ‘d’ is inserted in a parallel branch with the fragment $\wedge(\circlearrowleft(\rightarrow(b,c),\tau),d)$.

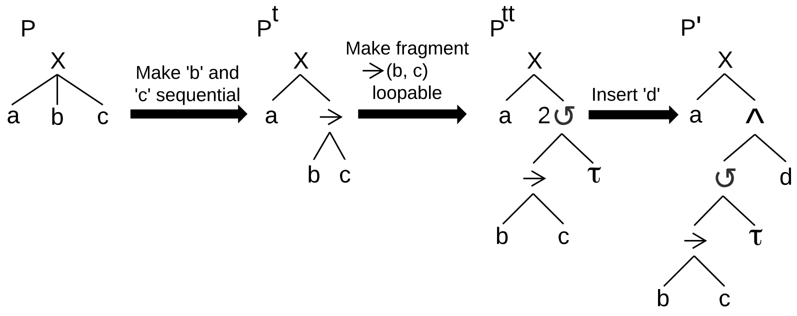


Fig. 13. An example of transforming a process tree P into a process tree P' via the application of nested changes.

Table 4 shows the format of drift characterization statements produced by our method for each change pattern.

6.4 Unsupported patterns

The only change pattern from Table 1 that our method is unable to support is *Synchronize two fragments*. This pattern refers to changes where two parallel fragments are synchronized, or vice versa, de-synchronized. As discussed in Section 5.1, this pattern introduces unstructuredness into a process model and hence cannot be used as a basis for defining process tree edit operations. Figure 14a shows an example of this change pattern. In this example, before the change, activity ‘b’ is performed in parallel with activity ‘c’, while after the change, ‘b’ precedes ‘c’. We observe that this pattern (“cd” in Table 1) is different from the pattern where we sequentialize two parallel fragments (“pl” in Table 1). This latter pattern transforms a parallel block containing the fragments to a sequential block without affecting the structuredness of the model. In constrast, in the synchronization change pattern two parallel fragments are synchronized by directly connecting one fragment to the other, and this results in the loss of structuredness in the parallel block. To discover a structured process model from this fragment, IM needs to generalize the behavior of the fragment. This leads to a process tree that does not precisely represent the change of synchronization applied to the process model, i.e. it

Code	Change pattern	Drift characterization statement format
sre	Insert/delete a fragment between two fragments	After the drift, fragment $f_1 = \ldots$ is inserted (resp., deleted from) between fragments $f_2 = \ldots$ and $f_3 = \ldots$.
pre	Insert/delete a fragment in/from parallel branch	After the drift, fragment $f_1 = \ldots$ is inserted in (resp., deleted from) a parallel branch with fragment $f_2 = \ldots$.
cre	Insert/delete a fragment in/from conditional branch	After the drift, fragment $f_1 = \ldots$ is inserted in (resp., deleted from) a conditional branch with fragment $f_2 = \ldots$.
cp	Duplicate a fragment	After the drift, fragment $f_1 = \ldots$, i.e. a duplicate of fragment $f_2 = \ldots$, is inserted ... (continues with sre, pre, or cre).
rp	Substitute a fragment	After the drift, fragment $f_1 = \ldots$ is substituted by fragment $f_2 = \ldots$.
sw	Swap two fragments	After the drift, fragments $f_1 = \ldots$ and $f_2 = \ldots$ are swapped.
sm	Move a fragment to between two fragments	After the drift, fragment $f_1 = \ldots$ has moved to between fragments $f_2 = \ldots$ and $f_3 = \ldots$.
cm	Move a fragment into/out of conditional branch	After the drift, fragment $f_1 = \ldots$ has moved to a conditional branch with fragment $f_2 = \ldots$.
pm	Move a fragment into/out of parallel branch	After the drift, fragment $f_1 = \ldots$ has moved to a parallel branch with fragment $f_2 = \ldots$.
cf	Make fragments mutually exclusive/sequential	Before the drift, fragments $f_1 = \ldots, \ldots$ and $f_n = \ldots$ were mutually exclusive (resp., sequential), while after the drift they are sequential (resp., mutually exclusive).
pl	Make fragments parallel/sequential	Before the drift, fragments $f_1 = \ldots, \ldots$ and $f_n = \ldots$ were parallel (resp., sequential), while after the drift they are sequential (resp., parallel).
lp	Make a fragment loopable/non-loopable	After the drift, fragment $f_1 = \ldots$ has become loopable/non-loopable.
cb	Make a fragment skippable/non-skippable	After the drift, fragment $f_1 = \ldots$ has become skippable/non-skippable.
fr	Change branching frequency	Before the drift, after the $\times$-node $\oplus$ the branch of fragment $f_1 = \ldots$ was executed $x\%$ of the time, while after the drift it is executed $y\%$ of the time.

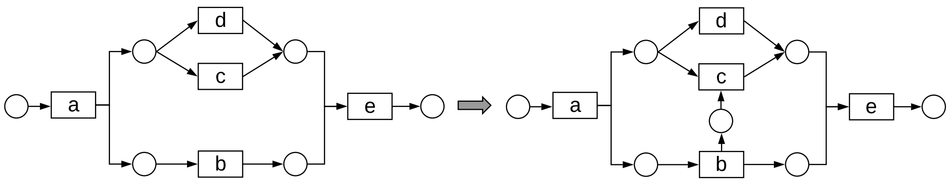
Table 4. Change patterns from [33] and their drift characterization statement format.

captures false-positive changes. Figure 14b shows the process trees corresponding to the process models in Figure 14a discovered by IM. Activity ‘c’, which was in parallel with activity ‘b’ and mutually exclusive to activity ‘d’ in process tree P, is performed after the parallel block in P'. Further, activities ‘c’ and ‘d’ can be skipped in P'. Although the occurrence of ‘c’ after ‘b’ is accurately captured in P', there are several false-positive changes in P' such as the occurrence of ‘c’ after ‘d’, or the occurrence of ‘e’ after ‘b’ by skipping ‘c’.

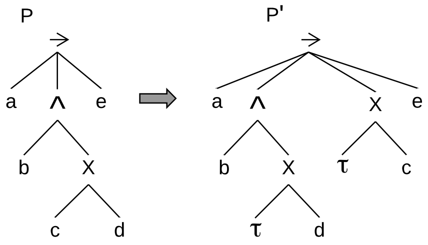
7 EVALUATION ON ARTIFICIAL LOGS

We implemented our method as an extension of the ProDrift plugin for the Apromore platform.[3] This tool is fed with an event stream replayed from an event log, and outputs, for each detected drift, its characterization statements in natural language. To evaluate the effectiveness of our method we used this tool to conduct experiments on artificial and real-life event logs with different parameters settings. In the rest of this section, we present the results of our evaluation on artificial logs. Specifically, we measured the accuracy of drift characterization and the conciseness of the statements produced to characterize such drifts, and compared the results against the technique in [23] (baseline). We chose this technique as a baseline because it has already shown to outperform the technique by Van Beest et al. [28]. Likewise, the technique by [2] has been discarded as a baseline because it would produce similar results as in Van Beest et al., given that it is based on the same underlying structures to capture and compare process behavior. We did however test these two alternative techniques against the event logs used in our evaluation and both techniques do not scale up to these datasets. Finally, we discarded the techniques by Bolt et al. [4] and by Nguyen et al. [22] because they require visual inspection of the results, i.e. they are not fully automated. In the next section, we present the results of our evaluation on real-life logs.

[3]Available at http://apromore.org/platform/tools



(a) Petri net models before and after the synchronization.



(b) Process trees P and P' discovered by IM before and after the synchronization.

Fig. 14. Example of *Synchronizing two fragments* change pattern: activities 'b' and 'c' are synchronized by setting a causality relation between the two activities.

7.1 Setup

We generated an artificial dataset using the CPN[4] base model illustrated in Figure 15. This model represents a block-structured process, consisting of 42 activities, five XOR, six AND, and three loop structures, modeled in an intertwined way, producing highly variable event logs with trace variability of around 80%. For each change pattern, except "Duplicate a Fragment", in Table 1, we generated five logs, each featuring two drifts applied to fragments of a different size between one to five. Note that as IM does not discover process trees with duplicate activities, we do not experiment with logs containing drifts caused by a fragment duplication. Nonetheless, the process tree transformation algorithms presented in this chapter can be applied to process trees with duplicate fragments and are able to identify insertion (resp., deletion) of a duplicate fragment in (resp., from) a process tree. Also, label duplication techniques such as the ones introduced in [8, 18] can be used to pre-process a log before applying IM. For each log we simulated 3,000 traces, with drifts injected during the simulation at 1,000-trace intervals. The first drift is injected by applying a change pattern to the base model, and the second drift is injected by reversing the applied change.

[4]http://cpntools.org

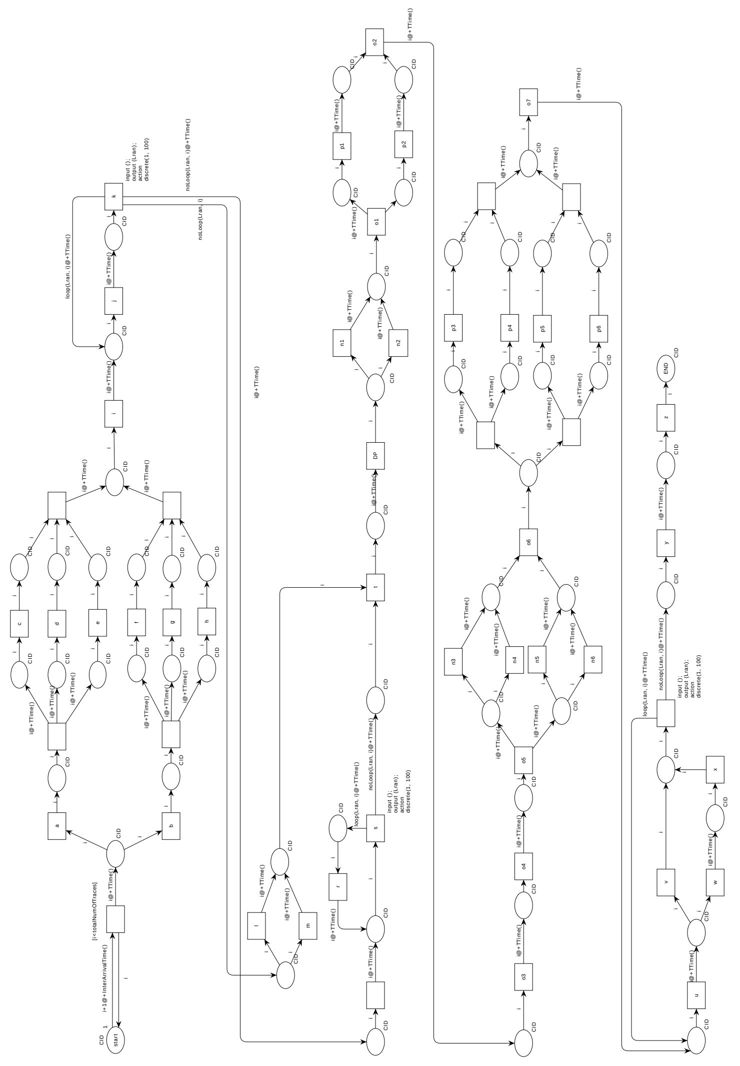


Fig. 15. Artificial process model in CPN Tools, used as a base model to simulate the artificial event logs

We also evaluated our method in more complex settings by simulating logs featuring drifts caused by multiple non-overlapping simultaneous changes (i.e. composite changes) as well as nested changes. To create such logs, we divided our change patterns into three categories: Insertion ("I"), Resequentialization ("R") and Optionalization ("O") (cf. Table 1). Limited to three cross-category changes, these categories make six possible scenarios for each of the composite changes and nested changes ("IOR", "IRO", "OIR", "ORI", "RIO", "ROI"). For each such scenario, five logs were generated by randomly selecting one template from each category and applying them to fragments of a certain size, from one to five. For example, a drift from the composite change scenario of "IOR" could simultaneously delete a fragment of size two ("I"), add a loop over a fragment of size two ("O"), and parallelize two sequential fragments of size two ("R") in three different locations of the process. As another example, a drift in the process from the nested changes scenario of "IOR" could first parallelize two sequential fragments of size three ("R"), then add a loop over the two parallelized fragments ("O"), and finally insert a fragment of size three in a conditional branch with the resulting loop fragment ("I"). This resulted in 30 logs for each of the non-overlapping and nested changes settings. In turn, this resulted in a collection of 65 logs with single changes, 30 logs with composite changes, and 30 logs with nested changes, each containing 3,000 traces with two equidistant drifts involving one or multiple fragments of a certain size.

For each such log, we also generated two variants with 2.5% and 5% noise by inserting random events into the traces of the log. Altogether, the artificial dataset contained 375 logs.[5]

7.2 Accuracy of drift characterization: our method vs baseline

In the first experiment, we evaluate and compare the accuracy of our method in characterizing drifts detected in the artificial logs versus that of the method in [23] (baseline). The baseline method, to the best of our knowledge, is the only method in the literature that is specifically formulated to address the process drift characterization problem. As discussed, this latter method is built on top of a drift detection technique [24] that detects sudden drifts from event streams. To detect a drift, this technique performs a statistical test over distributions of α^+ relations in two adjacent windows, namely reference and detection windows, sliding along the event stream. The size of the windows is adjusted based on the variability of the underlying process behavior, as captured by the α^+ relations. The detection window contains the most recent events arriving on the stream, and therefore represents the latest process behavior. A drift is detected when the *P-value* of the statistical test drops and remains below a certain threshold for a certain number of tests. As the underlying process behavior stabilizes, i.e. the distributions of α^+ relations in the reference and detection windows become similar, the *P-value* of the statistical test rises above the threshold. To characterize a drift, the baseline method first extracts the α^+ relations from the sub-logs before and after a detected drift. The relations with the highest association with the drift are then filtered and matched to a set of predefined change templates, and the best-matching templates are reported to the user via natural language statements. To ensure that we use the same sub-logs as our baseline for drift characterization, we used the same technique in [24] for drift detection in our experiments with the artificial and the real-life event streams in this paper. Furthermore, we also used the same strategy as the baseline to extract pre-drift and post-drift sub-logs after the detection of a drift. Specifically, we use the two sub-logs of partial traces built, respectively, from the events in the reference window as the *P-value* drops below the threshold, and from the events in the detection window as the *P-value* rises above the threshold, to discover the pre-drift and post-drift process trees, respectively. By doing this, we try to obtain pre-drift and post-drift process trees that only represent the actual process behaviors before and after a drift. It is worth noting that our drift characterization method can be applied on top of any drift detection technique that works on event streams or trace streams. The only required input to our method is a pair of sub-logs containing partial or complete traces from before and after a drift.

The output of both our method and the baseline method is a list of statements explaining the changes underpinning a drift. To compare the accuracy of the reported statements by the two methods we use F-score, i.e. the harmonic mean of recall and precision, where recall measures the ratio of reported statements relevant to the drift over the total number of statements required to explain the drift, and precision measures the ratio of reported statements relevant to the drift over the total number of reported statements. The relevance of a statement to a drift is assessed manually such that a statement is considered to be relevant to the drift if it describes at least a fraction of the changes applied to the process in order to inject that drift. We count the number of statements required by a method to explain a drift based on the changes applied to the process to inject that drift and the abstraction level of the characterization statements produced by that method. For example, to explain

[5]All the CPN models used for this simulation, the resulting artificial logs, and the detailed evaluation results are available with the software distribution.

deleting a fragment of three activities, the baseline method needs three statements, one per activity, since it is designed to characterize changes in the level of activities. On the other hand, our method needs only one statement to explain the same fragment deletion, as it is able to generate a statement describing a change to a fragment of multiple activities.

For the experiments in this section we use the A* algorithm (cf. Section 5.2.1) to compute edit operations to transform pre-drift process trees to post-drift process trees. We also use the baseline method with its default parameter settings.

Fragments of different size Figure 16 shows the average F-score over all logs of a certain fragment size, with and without noise, for our method and for the baseline. Figure 16a shows that the accuracy of our method is not influenced by the size of fragments involved in a drift, as the average F-score remains around 0.99 for all fragment sizes, over all noise-free logs. On the other hand, the average F-score of the baseline method drops as the fragment size increases, being on average around 0.85, 0.56, 0.38, 0.28 and 0.21 for fragments of size one, two, three, four and five over all noise-free logs, respectively. This is explained by the fact that the baseline method is limited to characterizing changes to fragments of size one, i.e. individual activities. For a change involving larger fragments this method either fails to characterize the change or can only partially characterize it, resulting in a significant drop in the recall, from 0.82 for fragments of size one to 0.13 for those of size five. However, the precision of the baseline is not influenced as much by the increase in the size of fragments, dropping from 0.98 for fragments of size one to 0.82 for those of size five.

For the experiments with logs that contain noise we used IMfpt, i.e. a variant of Inductive Miner that filters out infrequent behavior in the logs of partial traces, discovering noise-free pre-drift and post-drift process trees. To avoid introducing false differences between the pre-drift and post-drift process trees as a result of filtering, a process behavior is treated as noise if it does not meet the filtering requirements on both sides of the drift. This significantly improved the accuracy of our method in experiments with logs with noise. We set the noise filtering threshold parameter of IMfpt to 10% for the experiments with these logs. The results with the logs with 2.5% and 5% noise, in Figures 16b, and 16c, suggest that both our method and the baseline can to a great extent handle different levels of noise injected in the logs. The accuracy of our method incurs a slight decrease of around 15% for both 2.5% and 5% noise, with F-score being above 0.82 averaged over all logs of the same fragment size. This is mostly caused by a decrease in the precision of our method from 0.99, averaged over all fragment sizes, for noise-free logs, to 0.76 and 0.73 for logs with 2.5% and 5% noise, respectively. The average F-score of the baseline method also drops by around 10% per fragment size for logs with 2.5% and 5% noise. The precision of the baseline also drops from 0.91, averaged over all fragment sizes, for noise-free log to 0.7 and 0.67 for logs with 2.5% and 5% noise, respectively. The baseline method uses a statistical technique to filter out spurious relations from the extracted α^+ relations before matching them with change templates. With regards to the impact of fragment size on the characterization accuracy of the two methods, we observe similar trends as the noise-free logs. The accuracy of our method is not affected by the fragment size, whereas that of the baseline drops significantly as fragments became larger.

Process change patterns Figure 17 reports the average F-score for each single, composite and nested change pattern over all fragment sizes, with and without noise in the logs, for our method and for the baseline. In this figure, we distinguish the composite change patterns from the nested ones by appending "_c" and "_n" to their names, respectively. The results of the experiment in Figure 17a shows that in the absence of noise in the logs, our method has a perfect F-score of 1 for all the single change patterns and for all but four of the composite and nested change patterns, namely IOR_c, IRO_c, IRO_n, and RIO_n. For these four logs, the discovered process trees by IM had minor

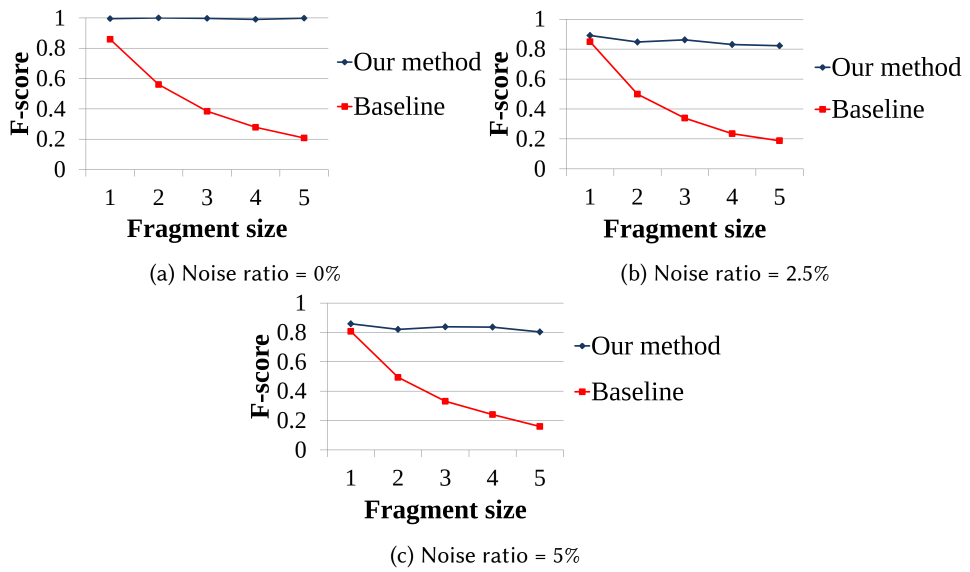


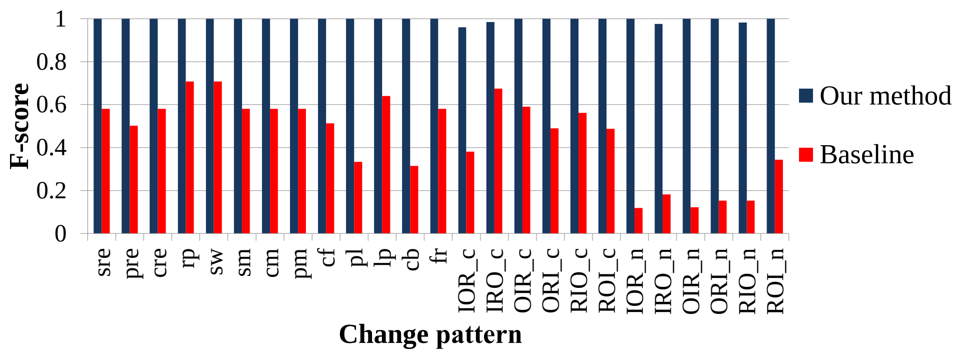
Fig. 16. Average F-score over all logs with different noise ratios per fragment size, obtained with our method vs. [23].

imprecisions, leading to some false statements. On the other hand, the baseline has an F-score in the range of 0.5 – 0.6 for most of the single and composite change patterns, with "cb" having the lowest F-score of 0.31, and "rp" and "sw" having the highest F-score of 0.7.

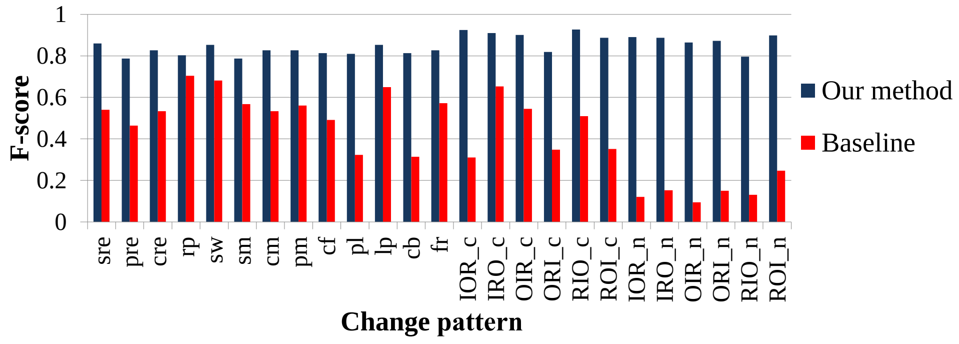
For the nested change patterns the baseline, as expected, performs poorly, with a maximum F-score of 0.34 for "ROI_n". This is due to the inherent inability of this method to characterize nested changes. For the logs with noise, as shown in Figure 17b and Figure 17c, despite a small drop in the accuracy of our method, this could filter out most of the injected noise in the logs, and achieve a higher F-score than the baseline for all single, composite and nested change patterns. The F-score falls to around 0.8 for most single change patterns for both 2.5% and 5% noise, and to around 0.9 and 0.85 for most composite and nested change patterns for 2.5% and 5% noise, respectively. The baseline method also handles the injected noise well and only incurs slight drops in its F-score. As explained before, this method can inherently filter out infrequent relations formed by spurious events.

Singleton fragments The results of the previous experiments show that our method on average outperforms the baseline in all change patterns over different fragment sizes. However, the baseline method is engineered to characterize non-overlapping activity-level changes.

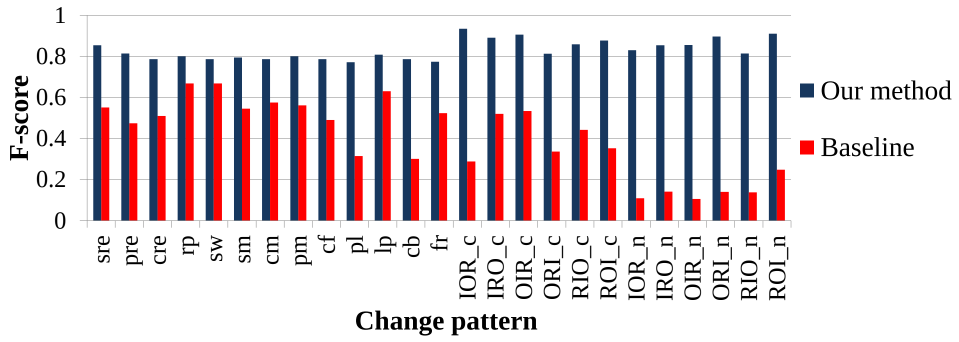
Therefore, in the last experiment in this section we study how our method compares to the baseline in characterizing changes to singleton fragments, i.e. individual activities. Figure 18 shows the F-score for singleton fragments per single, composite and nested change patterns for our method and the baseline. For noise-free logs, as shown in Figure 18a, our method achieves a perfect F-score of 1 for all the change patterns except "IRO_n", for which the discovered process trees by IM were not precise, leading to some false statements. The baseline also has an F-score of 1 for all but two of the single and composite change patterns, namely "lp" and "OIR_c". However, as expected, it still fails to fully characterize the nested changes, with a minimum F-score of 0.18 for "OIR_n", and an F-score of around 0.5 for the rest. For the logs with 2.5% and 5% noise (Figures 18b and 18c), the baseline has better F-scores than our method for almost half of the single and composite change



(a) Noise ratio = 0%



(b) Noise ratio = 2.5%

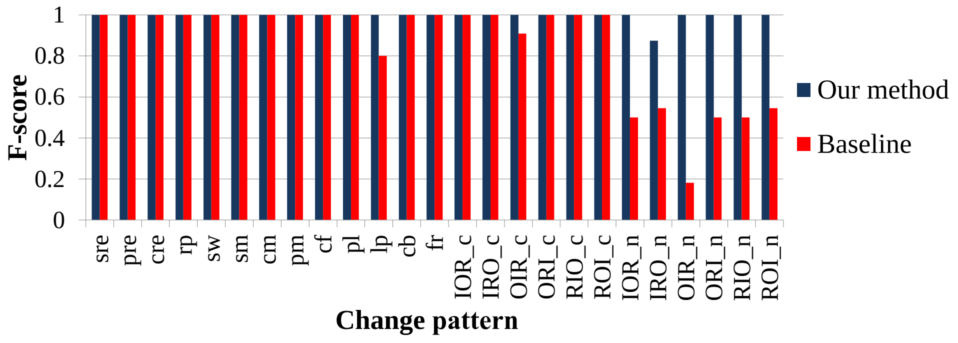


(c) Noise ratio = 5%

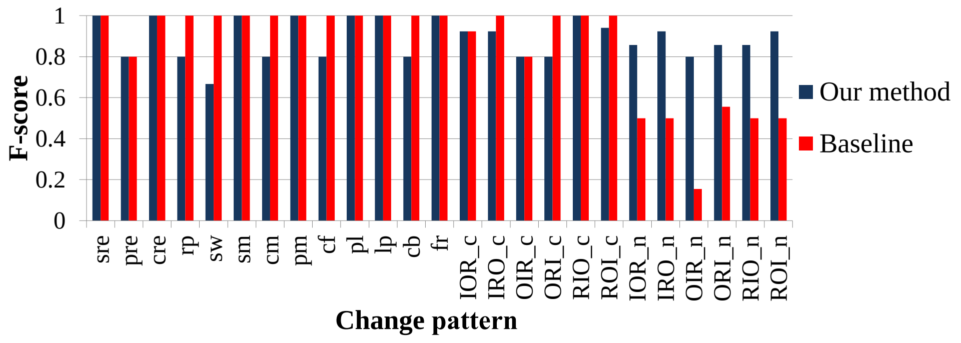
Fig. 17. Average F-score over all fragment sizes per single, composite, and nested change pattern, obtained with our method vs. [23].

patterns. For these patterns our method still achieves an F-score of 0.8, except for four patterns ("sw", "rp", "OIR_c", "ORI_c") where the value is above 0.65. For the remaining patterns the two methods perform equally well. On the other hand, our method outperforms the baseline for all the nested changes.

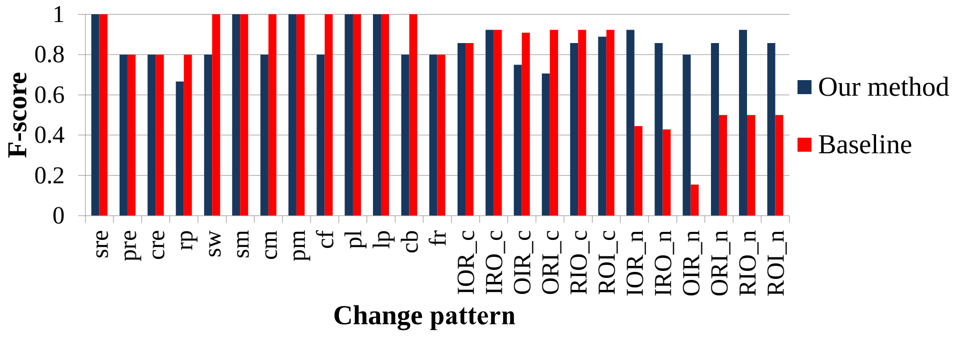
Overall, the experimental results so far show that both methods are noise-tolerant. And while our method is able to accurately characterize single, composite, and nested changes involving fragments of any size, the baseline method is better-suited for non-overlapping activity-level changes. However,



(a) Noise ratio = 0%



(b) Noise ratio = 2.5%



(c) Noise ratio = 5%

Fig. 18. Average F-score for singleton fragments per single, composite and nested change pattern, obtained with our method vs. [23].

this latter method fails to accurately characterize changes involving larger fragments, overlapping changes, as well as nested changes.

7.3 Verbalization conciseness: exhaustive vs greedy

The characterization accuracy of our method is dependent on that of IM in discovering pre-drift and post-drift process trees. If a process tree discovered with IM misrepresents the process behavior recorded in the event log, e.g. due to the imprecision of IM, then that behavior will produce a false characterization statement. Consequently, the choice of the search algorithm for computing the sequence of edit operations that transforms a pre-drift process tree to a post-drift process tree only impacts the number of reported statements to the user. For example, consider two transformations of process trees P and P' in Figure 19. In Figure 19a, P is transformed to P' by moving activity 'a' to a conditional branch with activity 'c', whereas in Figure 19b, P is transformed to P' by first swapping activities 'a' and 'b', and then making activities 'a' and 'c' parallel. Although both of these are correct, the first way is preferred as it is more concise.

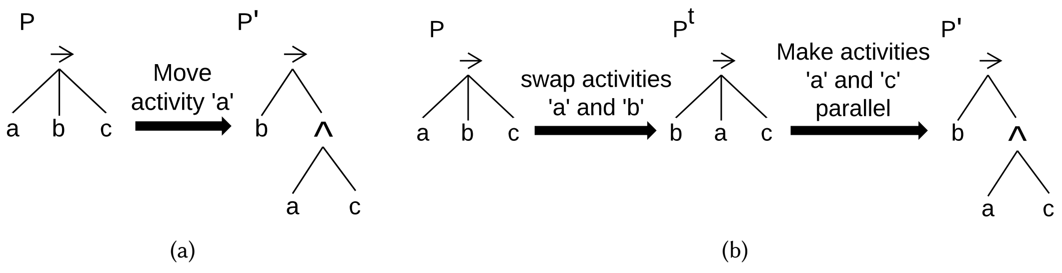
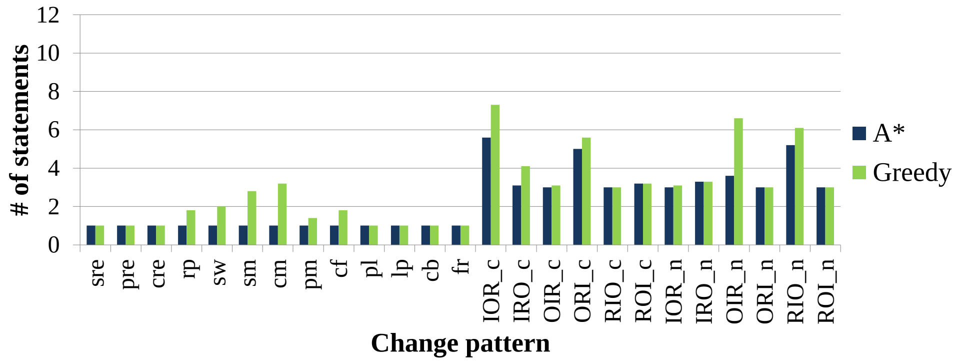

P
→
a b c
Move activity 'a'
P'
→
b
∧
a c
(a)
P
→
a b c
swap activities 'a' and 'b'
P^t
→
b a c
Make activities 'a' and 'c' parallel
P'
→
b
∧
a c
(b)

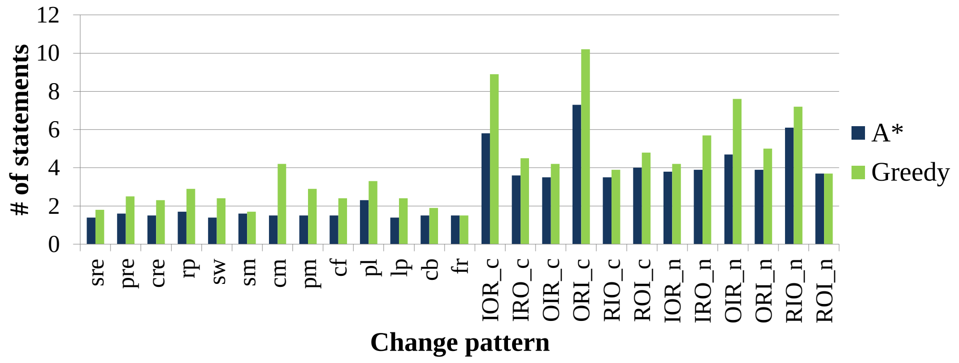
Fig. 19. Two sample transformations of process tree P into process tree P'.

In this section, we evaluate the verbalization conciseness of our method by counting the number of characterization statements reported by our method using the A* versus the greedy algorithm. As explained in Section 6, drift characterization statements are produced based on simple as well as compound changes, where each compound change is an aggregation of multiple simple changes. The threshold parameter of the greedy algorithm, which indicates the minimum matching score between two mapped operator nodes, can be manually set by the user. As the greedy algorithm has a low execution time the user may try different threshold values and select one that results in the lowest number of reported statements. For the experiments in this section, we set the threshold parameter of the greedy algorithm to 0.6, i.e. two operator nodes are mapped if their matching score is at least 0.6. Intuitively, a matching score of 0.6 means that the two nodes are more similar than dissimilar, and therefore they should be matched. This is also consistent with previous experiments on model matching in the context of process model merging, where a value of 0.6 was used [12]. Figure 20 reports the average number of statements produced by our method using the A* vs the greedy algorithm, over all fragment sizes, per change pattern, with and without noise. For noise-free logs, the reported statements by our method for all but 4 of the single, composite and nested change patterns ("IOR_c", "IRO_c", "IRO_n", "RIO_n"), were all accurate as the F-score of our method for these changes was 1 (cf. Figure 17a).

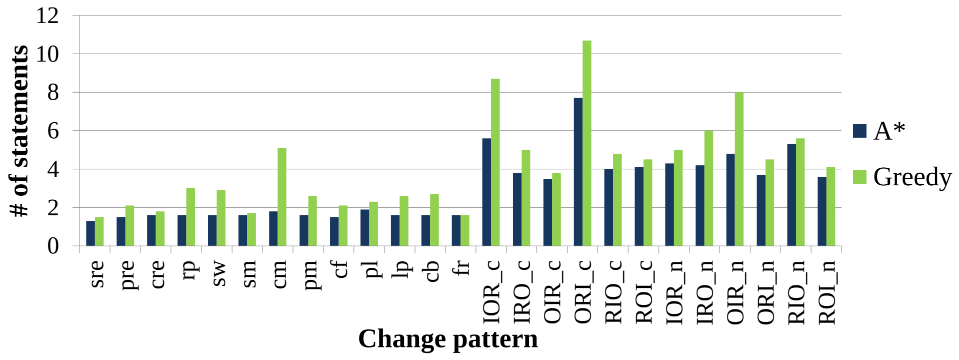
As shown in Figure 20a, using the A* algorithm our method is able to characterize each single change pattern with one statement, averaged over fragments of size one to five. As the F-score of our method was 1 for the same change patterns in noise-free logs (cf. Figure 17a) these results show that the number of statements reported by our method for a change pattern is independent of the size of the fragments to which the change pattern is applied. In regards to the complex change patterns, our method with the A* on average, across all fragment sizes, produces around 3 statements, one per applied change, for all but three of the composite and nested change patterns. For those three change patterns, namely "IOR_c", "ORI_c", and "RIO_n", the pre-drift and post-drift sub-logs for larger fragments did not contain sufficient process behavior for IM to precisely discover the fragments to which the changes were applied. As such, IM split those fragments into smaller fragments, leading to our method producing more statements. For the noise-free logs, our method produces similar number of statements when using the greedy algorithm for most of the single, composite and nested change patterns. However, for some change patterns, e.g. "sm" and "cm", the greedy leads to more



(a) Noise ratio = 0%



(b) Noise ratio = 2.5%



(c) Noise ratio = 5%

Fig. 20. Average number of statements over all fragment sizes per change pattern reported by our method using the A* algorithm vs the greedy algorithm.

statements, with the largest difference being for "OIR_n" with 6.6 statements against 3.6 statements reported by our method when using the A*.

The injection of noise in the logs, as shown in Figures 20b and 20c, slightly increases the average number of statements reported by our method for all change patterns over fragments of size one to five. For these logs, our method produced some false statements, each explaining a change that was not applied to the process as part of the drift injection. Furthermore, in some cases the injected noise caused IM to split a large fragment involved in a change into multiple smaller fragments, causing our method to produce more statements to explain the change. Similar to the noise-free logs, the A* and the greedy algorithms perform similarly for most of the change patterns with 2.5% and 5% noise. The largest difference is for the simple change pattern "cm" with 5% noise, where our method produces 1.8 statements on average using the A* versus 5.1 statements on average using the greedy algorithm.

7.4 Verbalization conciseness: our method vs baseline

Finally, we study how our method compares to the baseline method with regard to the number of statements required to characterize the various change patterns, in the absence of noise. As observed in the previous experiments, the baseline often misses to characterize changes that involve non-singleton fragments and hence does not report any statement, or it may partially identify them, resulting in a small number of statements being reported. Thus, the actual number of reported statements by the baseline is not a good indicator of its verbalization conciseness. To obviate this problem, in Figure 21 we count the number of statements each method would require to report all process model changes behind each change pattern, if it could fully identify them. Further, as the baseline does not support the nested changes we exclude them from the comparison.

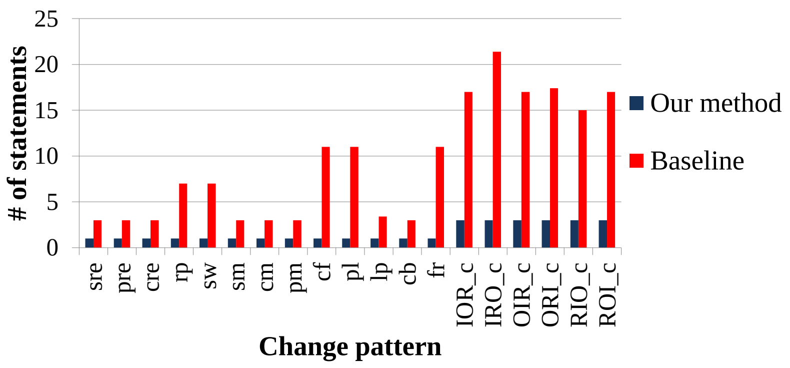


Fig. 21. Average number of statements over all fragment sizes required by our method vs. [23] for characterizing each change pattern.

We can see that the baseline would report a substantially larger number of statements (1 compared to 5.5 on average over all simple change patterns), specially when drifts involve multiple large process fragments, as in the case of composite patterns (3 compared to 17.5 on average). Reporting many activity-level differences is a common limitation of those methods like our baseline that rely on low-level representations of the process behavior.

7.5 Time performance

We conducted all the experiments on an Intel i7 2.20GHz with 16GB RAM (64 bit), running Windows 7 and JVM 8 with a heap space of 10GB. The time required to discover two process trees from the pre-drift and post-drift sub-logs and compute the edit operations to transform the pre-drift process tree to the post-drift process tree using the A* algorithm for each drift ranged from a minimum of 2ms to a maximum of 68sec with an average of 870ms. To perform the same operations it took the greedy algorithm from a minimum of 2ms to a maximum of 100ms with an average of 30ms. This means that the greedy algorithm is almost 30 times faster than its A* counterpart. The bulk of the time spent by our method was on computing the edit operations to transform the pre-drift process tree to the post-drift process tree. Although in most cases the A* algorithm finds the optimal solution within a reasonable time, for two process trees with several changes it may be more efficient to use the greedy algorithm. In comparison to these results, the baseline method took on average 510ms to characterize each drift (min = 430ms, max = 690ms).

8 EVALUATION ON REAL-LIFE LOGS

We further evaluated our method on two real-life event logs, one from a ticketing management process and the other from an insurance claim handling process. For the experiments in this section, we used IMfpt for discovering noise-free pre-drift and post-drift process trees, by setting its noise filtering threshold parameter to 10%. We also used the A* algorithm to compute the shortest sequence of edit operations to transform the pre-drift process tree to the post-drift process tree. Furthermore, we considered a relative frequency change of at least 50% as a significant change, where the relative frequency of the fragment is at least 25% in the pre-drift or the post-drift process tree (cf. 6.2).

The first real-life log,[6] is a public log available from the 4TU Data Centrum,[7]. This log contains events from a ticketing management process of the help desk of an Italian software company. There are 21,348 events from 14 activities and 4,580 traces, out of which 226 are distinct. We used the drift detection technique in [24], by initializing its adaptive windows with 1,000 events, and detected 2 drifts in this log. The first drift occurs at the event index 8,757, corresponding to the date July 25th 2011, and the second one occurs at the event index 17,307, corresponding to the date September 11th 2012. We characterized these two drifts by applying our method to the sub-logs extracted from before and after each drift. The transformation of the pre-drift process tree to the post-drift process tree over the first drift is illustrated in Figure 22. For the first drift, our method produced a single statement, reporting on the possibility of skipping the sub-tree marked as "Fragment 1" in Figure 22, Fragment 1 after the occurrence of the drift. We did not have access to a ground truth to validate the obtained results. As an alternative, we analyzed the directly follows graph of the sub-logs from before and after the drift, shown in Figures 23a and 23b, to verify the accuracy of the results. We observed the appearance of a directly follows relation from activity "Assign seriousness" to activity "Resolve ticket" after the drift. This finding aligns with the output of our method for this drift.

For the second drift, our method discovered two changes. The transformation of the pre-drift process tree to the post-drift process tree over the second drift is illustrated in Figure 24. The first discovered change indicates a significant decrease in the relative frequency of the τ-node 8 from 80% to 40%, while the second change indicates a significant increase in the relative frequency of activity "Wait" from 18% to 51%. These two changes are related as activity "Wait" and the τ-node 8 are parented by the same $\times$-node 5. To evaluate the accuracy of these changes we have drawn the directly follows graph of the sub-logs from before and after the drift in Figures 25a and 25b. The pre-drift and post-drift graphs show that the frequencies of the outgoing arcs from activity "Take in charge ticket" to activities "Resolve ticket", "Wait", and "Require upgrade" have changed from 151 (80% of total), 35 (18% of total), and 4 (2% of total) to 66 (40% of total), 82 (51% of total), and 14 (9% of total), respectively, out of which the first two changes are considered as significant. Moreover, in the corresponding process trees to these directly follows graphs, the change in the relative frequency of activity "Resolve ticket" manifests itself by a change in the relative frequency of the τ-node 8. These findings conform to the characterization of this drift by our method.

Our method characterizaed the first and the second drift in 330ms and 350ms, respectively. It is worth noting that since the employed drift detection technique is designed to detect sudden drifts,

[6]https://doi.org/10.4121/uuid:0c60edf1-6f83-4e75-9367-4c63b3e9d5bb
[7]https://data.4tu.nl/repository/

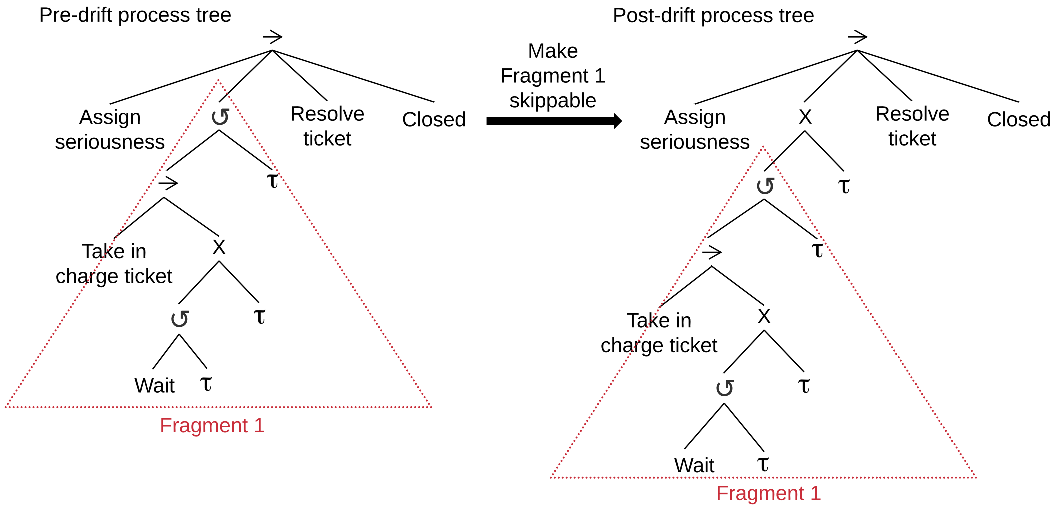
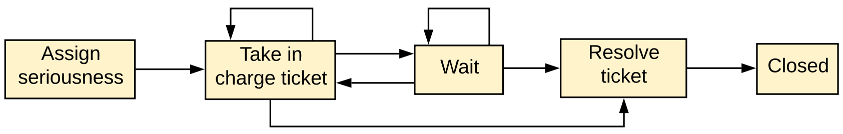
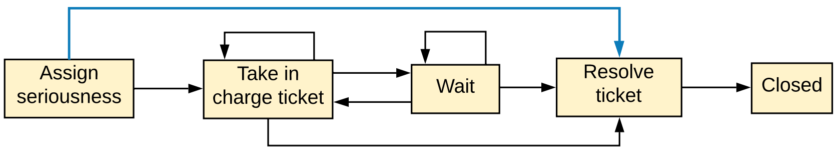


Fig. 22. Transformation of pre-drift process tree to post-drift process tree over the first drift in the ticketing management process.



(a) Directly follows graph before the first drift.



(b) Directly follows graph after the first drift.

Fig. 23. Directly follows graphs of the ticketing management process before and after the first drift.

gradual changes that occurred over the period from the first drift to the second drift, e.g. the insertion of activity "Require upgrade", did not trigger the detection of another drift over this period.

We also applied the baseline method to the discovered drifts in this log, but this method failed to characterize the drifts as it did not report any changes.

Next, we employed our method to characterize drifts in an event log originating from the claims management system of a large Australian insurance company. This private log consists of 61,413 events referring to twelve distinct activities and 16,365 traces, out of which 172 are distinct. It records cases of a windscreen claims handling process over a period of 13 months between 2011 and 2012. Using the drift detection technique in [24] with an adaptive window size initialized to 7,000 events, we detected one drift in this log at the event index 13821, corresponding to the date September 19th, 2011. Next, we used our method to characterize this drift. The transformation of the pre-drift process tree to the post-drift process tree over this drift is illustrated in Figure 26. Our method discovered that Fragment 1 consisting of three sequential activities "Identify Nil Recovery or Settlement Potential",

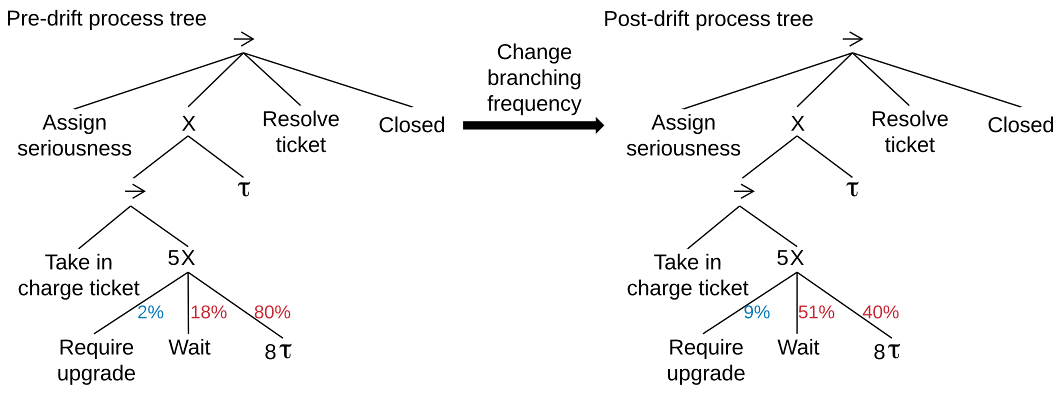
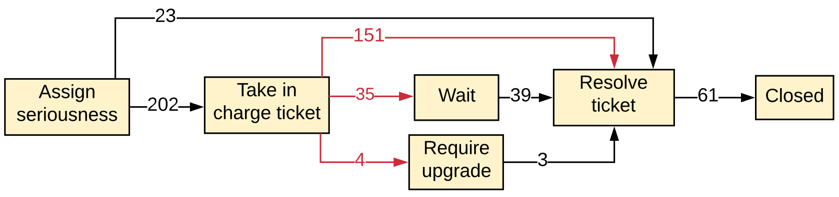
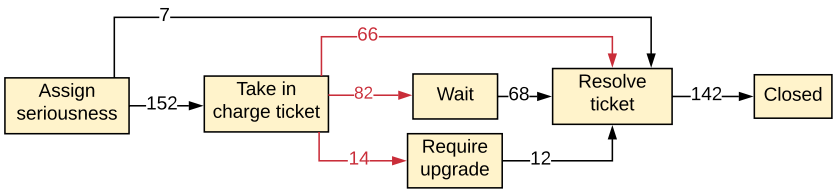


Fig. 24. Transformation of pre-drift process tree to post-drift process tree over the second drift in the ticketing management process.



(a) Directly follows graph before the second drift.



(b) Directly follows graph after the second drift.

Fig. 25. Directly follows graphs of ticketing management process before and after the second drift.

"Review Invoice - Motor Glass", and "Conduct File Review" in the pre-drift process tree is substituted by Fragment 2 consisting of two concurrent activities "Confirm Nil Recovery or Settlement Potential" and "Invoice Paid" in the post-drift process tree. Our method completed the characterization of this drift in 280ms.

We validated these results with a business analyst from the insurance company, who confirmed our findings and explained the reasons underlying the identified changes. Before the drift, by performing activity "Identify Nil Recovery or Settlement Potential" the company tried to claim a fraction of the money paid for every accident case from other insurance companies involved in the accident. However, as performing this task for all cases proved to be costly, they decided to perform it only for cases with certain characteristics, e.g. cases whose cost is below a certain threshold. Thus, they substituted this activity by a new activity, named "Confirm Nil Recovery or Settlement Potential". Moreover, during the same time period they automated invoice payments, by removing the two activities "Review Invoice - Motor Glass" and "Conduct File Review" and introducing a new activity,

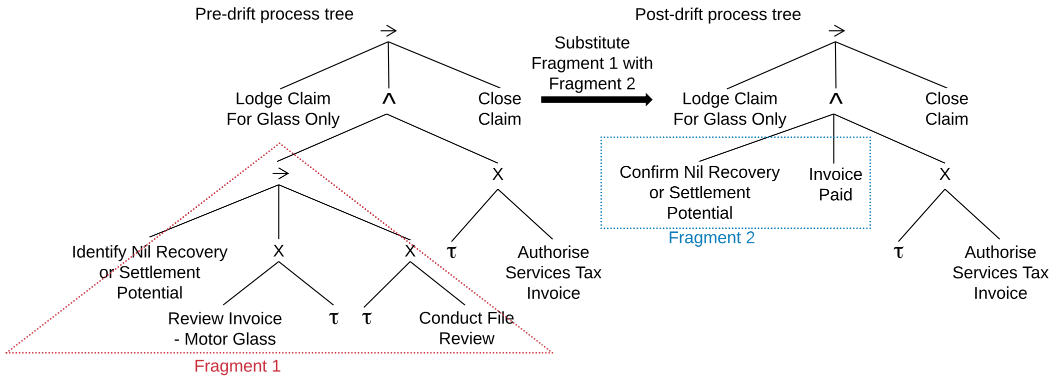

Pre-drift process tree
→
Lodge Claim For Glass Only
∧
Close Claim
→
X
Identify Nil Recovery or Settlement Potential
X
X
τ
Authorise Services Tax Invoice
Review Invoice - Motor Glass
τ
τ
Conduct File Review
Fragment 1
Substitute Fragment 1 with Fragment 2
Post-drift process tree
→
Lodge Claim For Glass Only
∧
Close Claim
Confirm Nil Recovery or Settlement Potential
Invoice Paid
X
Fragment 2
τ
Authorise Services Tax Invoice

Fig. 26. Transformation of pre-drift process tree to post-drift process tree over the drift in the claim handling process.

named "Invoice Paid". These two changes resulted in the substitution of Fragment 1 consisting of three sequential activities "Identify Nil Recovery or Settlement Potential", "Review Invoice - Motor Glass" and "Conduct File Review" in the pre-drift process tree by Fragment 2 consisting of two concurrent activities "Confirm Nil Recovery or Settlement Potential" and "Invoice Paid" in the post-drift process tree.

We also applied the baseline method to the discovered drift in this log. However, this method could only explain the removal of activity "Identify Nil Recovery or Settlement Potential", and failed to discover the other changes.

9 CONCLUSION

In this paper we presented a robust, automated method for characterizing process drifts at the level of fragments, from streams of business process events. We first adapted a state-of-the-art process discovery technique, namely Inductive Miner, to discover block-structured process models (a.k.a. process trees) from event streams. Next, we used this technique to discover two process trees, one from the portion of an event stream just before a given drift, and the other from the portion of stream just after the stream. We then presented a process tree transformation technique that finds a minimum-cost sequence of edit operations to transform a pre-drift process tree to a post-drift process tree. The search for such a sequence is guided by means of process tree mappings, and is supported by two search algorithms, an exhaustive A* algorithm and a fast greedy algorithm, which find the optimal solution or a close approximation thereof. The definition of edit operations and their costs is such that the method is able to characterize changes applied to fragments of any size, from individual activities to larger fragments. Moreover, the hierarchical structure of process trees allows the characterization of complex changes such as overlapping changes as well as nested changes. And as the edit operations are defined based on a well-established set of typical business process change patterns, the identified fragment-level changes can easily be translated into concise natural language statements based on those patterns. The proposed method can also characterize process drifts detected from event logs of complete traces, and can be used on top of any process drift detection technique so long as it is fed with a pre-drift and a post-drift sub-log.

We extensively evaluated our method using both highly variable artificial logs (also in the presence of noise), as well as two real-life logs, and compared the results with a baseline method. The results on the artificial logs show that our method is fast, noise-tolerant, highly accurate and concise in characterizing drifts induced by the application of typical process changes to fragments of different size. When using the greedy algorithm for process tree transformation, the method can scale up to the extent that it can work in real-time. In the experiments with real-life logs, our method could fully characterize the identified drifts. Despite the lack of a ground truth to validate the results in the experiment with the log of the ticketing management process, the results were supported by various observations from the log. For the experiment with the log of the insurance claims management process, a business analyst who works with the process in question confirmed our findings.

Our method outperforms the baseline method when characterizing changes applied to non-singleton fragments, as well as overlapping and nested changes. As expected though, the baseline is better-suited for non-overlapping singleton fragments (individual activities) in the presence of noise, as it uses features on a lower level of abstraction to capture the process behavior, and benefits from a statistically-grounded mechanism for identifying change patterns that best explain a drift. Yet, for this type of fragments, our method still performs at acceptable levels (F-score of 0.8 in most cases).

An avenue for future work is to characterize other classes of drifts beyond sudden drifts, such as gradual and incremental drifts. Another avenue for future work is to provide a visual description of the change patterns identified by our method as a simple and effective way to communicate the characteristics of the drift, as in [7].

ACKNOWLEDGMENTS

We thank Abderrahmane Maaradji for his feedback on early versions of this paper. This research is partly funded by the Australian Research Council (grant DP150103356).

REFERENCES

[1] Rafael Accorsi and Thomas Stocker. 2012. Discovering Workflow Changes with Time-Based Trace Clustering. In *Data-Driven Process Discovery and Analysis*. Springer.

[2] Abel Armas-Cervantes, Paolo Baldan, Marlon Dumas, and Luciano García-Bañuelos. 2014. Behavioral Comparison of Process Models Based on Canonically Reduced Event Structures. In *BPM*. Springer.

[3] Adriano Augusto, Raffaele Conforti, Marlon Dumas, Marcello La Rosa, Fabrizio M Maggi, Andrea Marrella, Massimo Mecella, and Allar Soo. 2018. Automated discovery of process models from event logs: Review and benchmark. *IEEE Transactions on Knowledge and Data Engineering* (2018).

[4] Alfredo Bolt, Massimiliano de Leoni, and Wil MP van der Aalst. 2017. Process variant comparison: Using event logs to detect differences in behavior and business rules. *Information Systems* (2017).

[5] R. P. Jagadeesh Chandra Bose, Wil M. P. van der Aalst, Indre Zliobaite, and Mykola Pechenizkiy. 2014. Dealing with concept drifts in process mining. *IEEE Transactions on NNLS* (2014).

[6] Josep Carmona and Ricard Gavalda. 2012. Online techniques for dealing with concept drift in process mining. In *International Symposium on Intelligent Data Analysis*. Springer.

[7] Abel Armas Cervantes, Nick RTP van Beest, Marcello La Rosa, Marlon Dumas, and Luciano García-Bañuelos. 2017. Interactive and Incremental Business Process Model Repair. In *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*. Springer, 53–74.

[8] Javier de San Pedro and Jordi Cortadella. 2016. Discovering duplicate tasks in transition systems for the simplification of process models. In *International Conference on Business Process Management*. Springer, 108–124.

[9] João Gama, Indrė Žliobaitė, Albert Bifet, Mykola Pechenizkiy, and Abdelhamid Bouchachia. 2014. A survey on concept drift adaptation. *ACM Computing Surveys (CSUR)* (2014).

[10] RP Jagadeesh Chandra Bose. 2012. Process mining in the large: Preprocessing, discovery, and diagnostics. (2012).

[11] Petr Kosina, Joao Gama, and Raquel Sebastiao. 2010. Drift Severity Metric.. In *ECAI*. 1119–1120.

[12] Marcello La Rosa, Marlon Dumas, Reina Uba, and Remco Dijkman. 2013. Business process model merging: An approach to business process consolidation. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 22, 2 (2013), 11.

[13] SJJ Leemans. 2017. *Robust process mining with guarantees*. Ph.D. Dissertation. Ph. D. thesis, Eindhoven University of Technology.

[14] Sander JJ Leemans, Dirk Fahland, and Wil M. P. van der Aalst. 2013. Discovering block-structured process models from event logs-a constructive approach. In *International Conference on Applications and Theory of Petri Nets and Concurrency*. Springer.

[15] Sander J. J. Leemans, Dirk Fahland, and Wil M. P. van der Aalst. 2013. Discovering Block-Structured Process Models from Event Logs - A Constructive Approach. In *Application and Theory of Petri Nets and Concurrency - 34th International Conference, PETRI NETS 2013, Milan, Italy, June 24-28, 2013. Proceedings (Lecture Notes in Computer Science)*, José Manuel Colom and Jörg Desel (Eds.), Vol. 7927. Springer, 311–329. https://doi.org/10.1007/978-3-642-38697-8_17

[16] Sander J. J. Leemans, Dirk Fahland, and Wil M. P. van der Aalst. 2013. Discovering Block-Structured Process Models from Event Logs Containing Infrequent Behaviour. In *Business Process Management Workshops - BPM 2013 International Workshops, Beijing, China, August 26, 2013, Revised Papers (Lecture Notes in Business Information Processing)*, Niels Lohmann, Minseok Song, and Petia Wohed (Eds.), Vol. 171. Springer, 66–78. https://doi.org/10.1007/978-3-319-06257-0_6

[17] Sander J. J. Leemans, Dirk Fahland, and Wil M. P. van der Aalst. 2014. Discovering Block-Structured Process Models from Incomplete Event Logs. In *Application and Theory of Petri Nets and Concurrency - 35th International Conference, PETRI NETS 2014, Tunis, Tunisia, June 23-27, 2014. Proceedings (Lecture Notes in Computer Science)*, Gianfranco Ciardo and Ekkart Kindler (Eds.), Vol. 8489. Springer, 91–110. https://doi.org/10.1007/978-3-319-07734-5_6

[18] Xixi Lu, Dirk Fahland, Frank JHM van den Biggelaar, and Wil MP van der Aalst. 2016. Handling duplicated tasks in process discovery by refining event labels. In *International Conference on Business Process Management*. Springer, 90–107.

[19] Abderrahmane Maaradji, Marlon Dumas, Marcello La Rosa, and Alireza Ostovar. 2015. Fast and Accurate Business Process Drift Detection. In *Proc. of BPM*.

[20] J. Martjushev, R. P. Jagadeesh Chandra Bose, and Wil M. P. van der Aalst. 2015. Change Point Detection and Dealing with Gradual and Multi-order Dynamics in Process Mining. In *BIR*.

[21] Leandro L Minku, Allan P White, and Xin Yao. 2010. The impact of diversity on online ensemble learning in the presence of concept drift. *IEEE Transactions on knowledge and Data Engineering* 22, 5 (2010), 730–742.

[22] H. Nguyen, M. Dumas, M. La Rosa, and A.H.M. ter Hofstede. 2018. Multi-perspective Comparison of Business Process Variants Based on Event Logs. In *In Proceedings of Conceptual Modeling (ER) (Lecture Notes in Computer Science)*, Vol. 11157. Springer, 449–459. https://doi.org/10.1007/978-3-030-00847-5_32

[23] A. Ostovar, A. Maaradji, M. La Rosa, and A.H.M. ter Hofstede. 2017. Characterizing Drift from Event Streams of Business Processes. In *Proc. of CAiSE (LNCS)*. Spriner.

[24] Alireza Ostovar, Abderrahmane Maaradji, Marcello La Rosa, Arthur H. M. ter Hofstede, and Boudewijn F. van Dongen. 2016. Detecting Drift from Event Streams of Unpredictable Business Processes. In *ER*.

[25] Kevin B Pratt and Gleb Tschapek. 2003. Visualizing concept drift. In *Proc. of the ninth ACM SIGKDD international conference on knowledge discovery and data mining*. ACM.

[26] Kuo-Chung Tai. 1979. The tree-to-tree correction problem. *Journal of the ACM (JACM)* 26, 3 (1979), 422–433.

[27] Irene Teinemaa, Marlon Dumas, Marcello La Rosa, and Fabrizio Maria Maggi. 2018. Outcome-oriented predictive process monitoring: review and benchmark. *ACM Transactions on Knowledge Discovery from Data* (2018).

[28] Nick RTP van Beest, Marlon Dumas, Luciano García-Bañuelos, and Marcello La Rosa. 2015. Log delta analysis: Interpretable differencing of business process event logs. In *Proc. of BPM*. Springer.

[29] Wil MP Van der Aalst. 2016. *Process mining: data science in action*. Springer.

[30] Eric Verbeek, Joos C. A. M. Buijs, Boudewijn F. van Dongen, and Wil M. P. van der Aalst. 2010. ProM 6: The Process Mining Toolkit. In *Proceedings of the Business Process Management 2010 Demonstration Track, Hoboken, NJ, USA, September 14-16, 2010 (CEUR Workshop Proceedings)*, Marcello La Rosa (Ed.), Vol. 615. CEUR-WS.org. http://ceur-ws.org/Vol-615/paper13.pdf

[31] Ilya Verenich, Marlon Dumas, Marcello La Rosa, Fabrizio Maggi, and Irene Teinemaa. 2019. Survey and cross-benchmark comparison of remaining time prediction methods in business process monitoring. *ACM Transactions on Intelligent Systems and Technology* (2019).

[32] Geoffrey I. Webb, Roy Hyde, Hong Cao, Hai Long Nguyen, and Francois Petitjean. 2016. Characterizing concept drift. *Data Mining and Knowledge Discovery* (2016). https://doi.org/10.1007/s10618-015-0448-4 arXiv:1511.03816

[33] Barbara Weber, Manfred Reichert, and Stefanie Rinderle-Ma. 2008. Change patterns and change support features–enhancing flexibility in process-aware information systems. *DKE* (2008).

[34] Kaizhong Zhang, Rick Statman, and Dennis Shasha. 1992. On the editing distance between unordered labeled trees. *Information processing letters* 42, 3 (1992), 133–139.

[35] Canbin Zheng, Lijie Wen, and Jianmin Wang. 2017. Detecting Process Concept Drifts from Event Logs. In *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*. Springer, 524–542.

A VALID MAPPING

This appendix provides the formal definition of valid process tree mapping. Before that, we need to define some concepts that are needed in the definition of valid mapping, such as deleted and inserted fragments in a mapping, and various types of nodes in a mapping. These definitions are interspersed with examples to aid their comprehension.

Definition 15 (Deleted fragments in a mapping). *Let M be a mapping between two process trees P and P', and let f be a fragment in P. The fragment f is deleted through M if $\forall_{P[k]\in f}(k,-1)\in M$.* □

Let $S=\{f_1,\ \ldots,\ f_n\}$ be the set of all deleted fragments in M. A fragment $f_i \in S$ is a *maximal deleted fragment* if there is no $f_j(\neq f_i)\in S$ such that f_i is a sub-fragment of f_j.

Definition 16 (Inserted fragments in a mapping). *Let M be a mapping between two process trees P and P', and let f be a fragment in P'. The fragment f is inserted through M if $\forall_{P'[k]\in f}(-1,k)\in M$.* □

Let $S=\{f_1,\ \ldots,\ f_n\}$ be the set of all inserted fragments in M. A fragment $f_i \in S$ is a *maximal inserted fragment* if there is no $f_j(\neq f_i)\in S$ such that f_i is a sub-fragment of f_j.

Example 11. *Figure 27 shows examples of deleted and inserted fragments through a mapping between process trees P and P'. The set of all deleted fragments in this mapping is $S=\{b,\ c,\ \wedge(b,c)\}$, among which Fragment $1=\wedge(b,c)$ is a maximal deleted fragment. The set of all inserted fragments in this mapping is $S'=\{e,\ f,\ \rightarrow(e,f)\}$, among which Fragment $2=\rightarrow(e,f)$ is a maximal inserted fragment.*

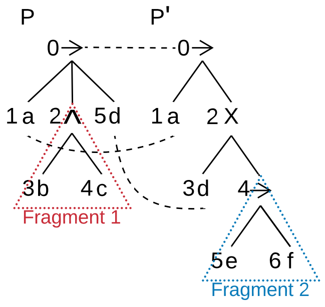


Fig. 27. Examples of deleted and inserted fragments in a mapping between process trees P and P'. Fragment 1 is a maximal deleted fragment, whereas Fragment 2 is a maximal inserted fragment.

Definition 17 (Auxiliary operator nodes in a mapping). *Let M be a mapping between two process trees P and P'. A non-$\circlearrowleft$-operator node $\oplus=P[i]$ (resp., $\oplus=P'[j]$) is an auxiliary operator in M if:*

i) $(i,-1)\in M$ (resp., $(-1,\ j)\in M$)
ii) Exactly one child fragment of $\oplus$ is not a deleted (resp., inserted) fragment in M.

□

An auxiliary operator node $v=P[i]$ in a mapping corresponds to a node deleted by the singularity reduction rule after a fragment deletion edit operation (cf. Definition 8), whereas an auxiliary operator node $v=P'[i]$ in a mapping corresponds to an auxiliary operator node inserted along with a fragment insertion (cf. Definition 9).

Definition 18 (Auxiliary τ-nodes in a mapping). *Let M be a mapping between two process trees P and P'. Also, let $v \in P$ (resp., $v \in P'$) be a τ-node parented by a $\circlearrowleft$-node $u \in P$ (resp., $u \in P'$). v is an* auxiliary τ-node *if v is deleted (resp., inserted) in M while u is not deleted (resp., inserted).* □

An auxiliary τ-node in a mapping corresponds to an auxiliary τ-node inserted (resp., deleted) as a result of deleting (resp., inserting) a child fragment of a $\circlearrowleft$-node by an edit operation D_f (resp., I_f) to keep the number of children of the $\circlearrowleft$-node at 2 (cf. Definitions 8 and 9).

Example 12. *As an example, in the mapping between process trees P and P' in Figure 28, the $\wedge$-node 2 in P' is an auxiliary operator node, inserted along with the insertion of activity 'e', and the τ-node 5 in P is an auxiliary τ-node, deleted as a result of inserting activity 'd'.*

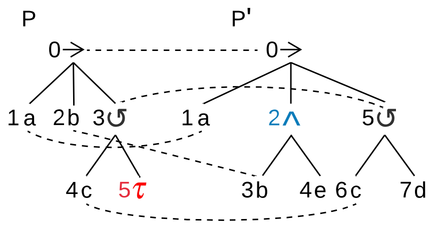


Fig. 28. Sample auxiliary operator node, i.e. the $\wedge$-node 2 in P', and sample auxiliary τ-node, i.e. the τ-node 5 in P, in a mapping between process trees P and P'.

Definition 19 (Trivial operator nodes). *Let M be a mapping between two process trees P and P'. A non-$\circlearrowleft$-operator node $v = P[i]$ (resp., $v = P'[j]$) is a* trivial operator node *in M if v is deleted (resp., inserted), at least two child fragments of v contain some nodes that are not deleted (resp., inserted), and at least one of the following conditions holds for v:*

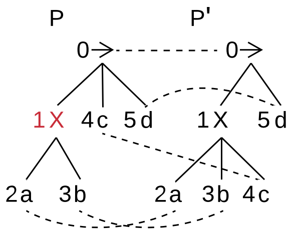
i) There exists an inserted (resp., deleted) operator node v' in P' (resp., P) such that $l(v') = l(v)$, and that all undeleted (resp., uninserted) leaves under v are mapped to leaves under v' and at least one uninserted (resp., undeleted) leaf under v' is not mapped to a node under v. Then, we refer to v as an indirectly-trivial operator node. *Let v' be the deepest node that satisfies this condition, then we refer to v' as* indirect parent *of v.*

ii) Let u be the deepest ancestor of v that satisfies one of the following conditions: • u is mapped to a node u' in P' (resp., P). • u is an indirectly-trivial operator node and a node u' in P' (resp., P) is its indirect parent. • u is an indirect parent for an indirectly-trivial operator node u' in P' (resp., P). such that all undeleted (resp., uninserted) leaves under v are mapped to leaves under u'. Then, one of the following should hold for v and u:
a) $l(v) = l(u)$.
b) $l(v) = l(u')$.

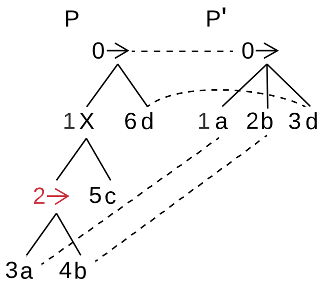
□

A trivial deleted operator node corresponds to an operator node deleted by the associativity reduction rules (cf. Definition 4) after the application of an edit operation. Inversely, a trivial inserted operator node corresponds to an operator node inserted as the root of a sub-fragment of a non-$\circlearrowleft$-operator node as a result of applying an edit operation.

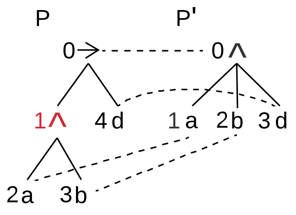
Example 13. *Figure 29 shows examples of trivial operator nodes in mappings. In Figure 29a, the $\times$-node 1 in P is an indirectly-trivial operator node and the $\times$-node 1 in P' is its indirect* *parent (condition i in Definition 19). After substituting the operator of the $\rightarrow$-node in the fragment $\rightarrow(\times(a,b),c)$ with $\times$, resulting in the insertion of the $\times$-node 1 $\in P'$, the $\times$-node 1 $\in P$, i.e. $\times(a,b)$-node, is deleted by the associativity reduction rule $\mathrm{A}_{\times}$. In Figure 29b, the $\rightarrow$-node 2 is a trivial operator node, as after deleting activity 'c', and subsequently the $\times$-node 1 by a singularity reduction rule, the $\rightarrow$-node 2 is deleted by the associativity reduction rule $\mathrm{A}_{\rightarrow}$ (condition iia in Definition 19). In Figure 29c, the $\wedge$-node 1 is a trivial operator node, as after changing the operation of the $\rightarrow$-node 0 to $\wedge$, the $\wedge$-node 1 is deleted by the associativity reduction rule $\mathrm{A}_{\wedge}$ (condition iib in Definition 19).*



(a) $\times$-node 1 in P.



(b) $\rightarrow$-node 2.



(c) $\wedge$-node 1.

Fig. 29. Examples of trivial operator nodes in mappings.

DEFINITION 20 (LOWEST MAPPED ANCESTORS). *Let M be a mapping between two process trees P and P'. The* lowest mapped ancestors *of two nodes $v \in P$ and $v' \in P'$ in M, denoted by $LMAs_M(v,\ v')$, is a pair $(u,\ u')$ of nodes, where $u = P[r]$ and $u' = P'[s]$ are ancestors of v and v', respectively, such that $(r,\ s) \in M$ and there is no pair $(m,\ n)$ in M, where $P[m]$ is an ancestor of v and $P'[n]$ is an ancestor of v', such that $dep(P[m]) > dep(P[r]) \wedge dep(P'[n]) > dep(P'[s])$.* □

DEFINITION 21 (VALID PROCESS TREE MAPPING). *Given two process trees P and P', a valid process tree mapping from P to P' is a mapping M satisfying the following conditions:*

1) *For every subtree $R = P[i](Q_1, Q_2)$ in P (resp., $R = P'[j](Q_1, Q_2)$ in P'), where $P[i]$ (resp., $P'[j]$) is a $\circlearrowleft$-node, and Q_1 and Q_2 are process trees, if $(i, -1) \in M$ (resp., $(-1,\ j) \in M$) then Q_2 should be a deleted fragment (resp., inserted fragment) in M.*
2) *Let $\oplus$ be an operator node in P (resp., P') that is mapped to an operator node in the other tree. If $l(\oplus) \in \{\rightarrow, \wedge\}$ (resp., $l(\oplus) = \circlearrowleft$) then at least two (resp., one) child fragments of $\oplus$ should contain some activity nodes that are not deleted (resp., inserted) in M. If $l(\oplus) = \times$ then at least two child fragments of $\oplus$ should not be deleted (resp., inserted) fragments, and one of which should contain some activity nodes that are not deleted (resp., inserted).*
3) *For every pair $(i,\ j)$ in M such that $t = P[i]$ and $t' = P'[j]$ are two τ-nodes, one of the following conditions should hold:*
   Let $q = P[r]$ and $q' = P'[s]$ be the parents of t and t', respectively.
   a) *There exists a pair $(r,\ s)$ in M.*
   b) *Let $v \in P$ and $v' \in P'$ be the deepest ancestors of t and t', respectively, that satisfy one of the following conditions: • $(v,\ v') = LMAs_M(t,\ t')$ (cf. Definition 20). • v is an indirectly-trivial operator node and v' is its indirect parent (cf. Definition 19). • v' is an indirectly-trivial operator node and v is its indirect parent. Let u be an ancestor of t (resp., t') such that u is on the shortest path from v (resp., v') to q (resp., q') and $l(u) \in \{\rightarrow, \wedge\}$. One of the following conditions should hold for u:*

i) u is an auxiliary operator node in M (cf. Definition 17).
ii) The child fragment of u containing t (resp., t') should at least contain an activity node that is not deleted (resp., inserted) in M.

□

The above conditions are defined to ensure that a mapping satisfies all the conditions of process tree edit operations and sequences thereof, defined in Section 5.1.

As defined for the edit operations $I_{\circlearrowleft}$ and $D_{\circlearrowleft}$, the second child fragment of a $\circlearrowleft$-node which is to be deleted (resp., inserted) is a τ-node (cf. Definitions 8 and 9). That is, to delete a $\circlearrowleft$-node we need to first delete its second child fragment (if $\neq\ \tau$) by a D_f operation. And to insert a $\circlearrowleft$-node with a non-τ second child fragment f we first need to insert the $\circlearrowleft$-node by a $D_{\circlearrowleft}$ operation and subsequently insert f as its second child. This is ensured in M by condition 1, which requires the deletion (resp., insertion) of the second child fragment of a deleted (resp., an inserted) $\circlearrowleft$-node in M.

EXAMPLE 14. *As an example, in the mapping between two process tree P and P in Figure 30 since the $\circlearrowleft$-node $P[1]$ is deleted, its second child fragment, i.e. activity b, is also deleted.*

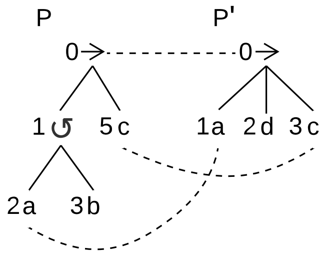


Fig. 30. Sample mapping that satisfies condition 1.

As defined in Definition 11, fragment deletions precede and fragment insertions follow all other operations in a sequence of edit operation. Also, as we explained in Section 5.1, after the application of each edit operation to a process tree we reduce the tree to normal form by applying reduction rules. As a result of the latter some operator nodes may be deleted. Thus, to ensure that an operator node $\oplus$ in P (resp., in P') that is mapped to an operator node in P' (resp., P) cannot be deleted by a reduction rule after (resp., before) the application of all fragment deletions (resp., insertions) we require $\oplus$ to satisfy condition 2.

EXAMPLE 15. *As an example, the invalid mapping between two process tree P and P in Figure 31 does not satisfy condition 2. This is because the $\wedge$-operator node $P[1]$ is mapped to a node in P', while it does not at least have two child fragments that have some undeleted activity nodes. As a result, after deleting activity 'a', the $\wedge$-node $P[1]$ will also be deleted by the singularity reduction rule* S *and hence cannot be mapped to a node in P'.*

As defined in Definition 4, a τ-node may be deleted by one of the τ-reduction rules, $T_{\rightarrow}$ or $T_{\wedge}$. As such, we defined condition 3 to ensure that a τ-node to which one of the τ-reduction rules can be applied is always deleted in a mapping. That is, we only allow a τ-node $t \in P$ to be mapped to a τ-node $t' \in P'$ in M if for which one of the two conditions, 3a or 3b, holds. Condition 3a requires the parents q and q' of t and t', respectively, to be mapped in M. For condition 3b we first define two nodes v and v' as the deepest ancestors of t and t', respectively, that are either mapped in M or one of them is an indirectly-trivial operator node and the other one is its indirect parent. To avoid the

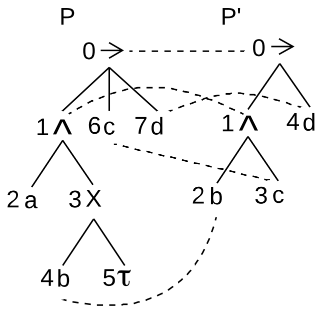


Fig. 31. Sample invalid mapping that does not satisfy condition 2.

deletion of t or t' by one of $T_{\rightarrow}$ or $T_{\wedge}$, we then require each $\rightarrow$- or $\wedge$-node u on the shortest path from q (resp., q') to v (resp., v') to satisfy one of the two conditions, *3bi* or *3bii*.

Example 16. *As an example, consider the mapping between the two process trees P and P' in Figure 32, where the τ-node $P[5]$ is mapped to the τ-node $P'[1]$, and $LMAs_M(P[5], P'[1]) = (P[0], P'[0])$. In this mapping, the $\wedge$-node $P[2]$ satisfies condition 3bi and the $\rightarrow$-node $P[1]$ satisfies condition 3bii.*

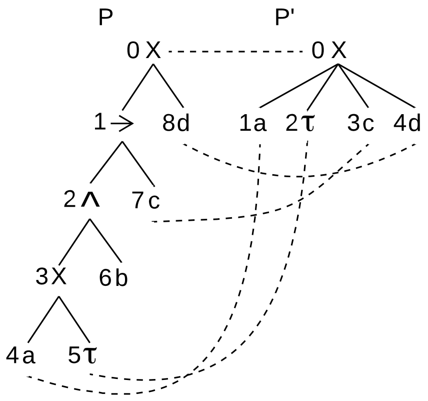


Fig. 32. Sample mapping that satisfies condition 3.

B FUNCTION STRINGIFY

This appendix provides the formal definition of the *stringify* function, which is used in the definition of the compound change patterns in Section 6.2.

Definition 22 (*stringify*). *stringify is a recursive function that converts a fragment to a unique and stable textual representation as follows:*

- *For an activity node or a τ-node v, $stringify(v) = l(v)$.*
- *For a fragment $F = \oplus(F_1, \dots F_n)$ such that $l(\oplus) \in \{\rightarrow, \circlearrowleft\}$, $stringify(F) = l(\oplus)(stringify(F_1) + \dots + stringify(F_n))$.*
- *For a fragment $F = \oplus(F_1, \dots F_n)$ such that $l(\oplus) \in \{\times, \wedge\}$, $stringify(F) = l(\oplus)(stringify(F_i) + \dots + stringify(F_m))$, where $1 \leq i, \dots m \leq n$ and*

$\{stringify(F_i),\ \dots\ stringify(F_m)\}$ is an ordered sequence obtained by arranging the sequence $\{stringify(F_1),\ \dots,\ stringify(F_n)\}$ in ascending alphabetical order.

□

C NOTATION

The notation used in this paper is summarized below.

Notation	**Meaning**
$\mathscr{L}$	Set of activity labels
$V(T)$	Set of nodes in tree T
$E(T)$	Set of edges in tree T
$\|T\|$	Size of tree T, equaling $\|V(T)\|$
$root(T)$	Root node of tree T
$T\langle v\rangle$	Subtree of tree T rooted at node $v \in T$
$Down_T(v)$	Sequence of nodes on the shortest path from $root(T)$ to node $v \in T$
$leaves(v)$	Set of leaves under internal node v
$l(v)$	Label of node v
$dep(v)$	Depth of node $v \in T$, equaling $\|Down_T(v)\|-1$
$dep(T)$	Depth of tree T, equaling the maximum depth of its nodes
$CA(v_1,\dots,v_n)$	Set of common ancestors of nodes $v_1,\dots,v_n$ in tree T, i.e. nodes in $Down_T(v_1)\cap\dots\cap Down_T(v_n)$
$LCA(v_1,\dots,v_n)$	Lowest common ancestor of nodes $v_1,\dots,v_n$ in tree T, i.e. the deepest node in $CA(v_1,\dots,v_n)$
$LCA(T\langle v_1\rangle,\dots,T\langle v_n\rangle)$	Lowest common ancestor of subtrees $T\langle v_1\rangle,\dots,T\langle v_n\rangle$, i.e. $LCA(v_1,\dots,v_n)$
$\times$	Exclusive choice operator
$\wedge$	Concurrency operator
$\rightarrow$	Sequence operator
$\circlearrowleft$	Loop operator
$P=\oplus(P_1,\dots P_n)$	Process tree P rooted at operator node $\oplus$ with subtrees $P_1\dots P_n$
τ-node	Leaf node t in process tree, representing the language with the empty trace, $l(t)\in\{\tau\}$
$C(v)$	Set of activity nodes under operator node $v\in P$, containing the activity nodes in $P\langle v\rangle$
$pre_P(v)$	Pre-order index of node v in process tree P
$P[i]$	Node with the pre-order index of i in P
$Rank(v,\oplus)$	Returns the rank of node v in ordered operator node $\oplus$
S	Singularity reduction rule
$\mathrm{A}_\times$	Associativity reduction rule for $\times$ operator
$\mathrm{A}_\wedge$	Associativity reduction rule for $\wedge$ operator
$\mathrm{A}_\rightarrow$	Associativity reduction rule for $\rightarrow$ operator
$\mathrm{T}_\rightarrow$	τ reduction rule for $\rightarrow$ operator
$\mathrm{T}_\wedge$	τ reduction rule for $\wedge$ operator
$SUB_\oplus$	Operator substitution edit operation
SUB_{ac}	Activity substitution edit operation
D_f	Fragment deletion edit operation
$D_\circlearrowleft$	$\circlearrowleft$-operator deletion edit operation
I_f	Fragment insertion edit operation
$I_\circlearrowleft$	$\circlearrowleft$-operator insertion edit operation
$LMAs_M(v,\ v')$	Lowest mapped ancestors (LMAs) of nodes v and v' in mapping M
$MST(P,\ P')$	Mapping search tree between process trees P and P'
$g^*(v)$	Returns the mapping cost up to node v in a mapping search tree
$h^*(v)$	Returns an estimation of the cost of mapping nodes in P and P' that have not yet been mapped up to node $v\in MST(P,\ P')$
$stringify(F)$	Returns a unique and stable textual representation of fragment F