Business & Information Systems Engineering manuscript No.
(will be inserted by the editor)

Repairing Alignments of Process Models

Sebastiaan J. van Zelst · Joos C.A.M. Buijs · Borja Vázquez-Barreiros · Manuel Lama · Manuel Mucientes

Received: date / Accepted: date

Abstract Process mining represents a collection of data driven techniques that support the analysis, understanding and improvement of business processes. A core branch of process mining is conformance checking, i.e., assessing to what extent a business process model conforms to observed business process execution data. Alignments are the de facto standard instrument to compute such conformance statistics. However, computing alignments is a combinatorial problem and hence extremely costly. At the same time, many process models share a similar structure and/or a great deal of behavior. For collections of such models, computing alignments from scratch is inefficient, since large parts of the alignments are likely to be the same. This paper presents a technique that exploits process model similarity and repairs existing alignments by updating those parts that do not fit a given process model. The technique effectively reduces the size of the combinatorial alignment problem, and hence decreases computation time significantly. Moreover, the potential loss of optimality is limited and stays within acceptable bounds.

Sebastiaan J. van Zelst
Fraunhofer Institute for Applied Information Technology
Fraunhofer Gesellschaft, Sankt Augustin, Germany
E-mail: sebastiaan.van.zelst@fit.fraunhofer.de

Joos C.A.M. Buijs
Department of Mathematics and Computer Science
Eindhoven University of Technology, Eindhoven, The Netherlands
E-mail: j.c.a.m.buijs@tue.nl

Borja Vázquez-Barreiros, Manuel Lama, Manuel Mucientes
Centro Singular de Investigación en Tecnoloxías da Informacíon (CiTIUS)
Universidade de Santiago de Compostela, Santiago de Compostela, Spain
E-mail: {borja.vazquez, manuel.lama, manuel.mucientes}@usc.es

Keywords Process mining, Conformance checking, Alignments, Process trees, Workflow nets

1 Introduction

Process mining (van der Aalst, 2016) has emerged as a means to analyse, understand and improve the behavior of an organization, based on the analysis of event data, i.e., known as *event logs*, stored during the execution of the process. We identify three main process mining areas: *process discovery*, *conformance checking* and *process enhancement*. In *process discovery*, the goal is to discover a process model that accurately describes the behavior recorded in an event log, i.e., a model describing *the real process* followed during process execution. In *conformance checking,* a process model is compared with the recorded behavior of the process to check whether there exist deviations between the model and the observed behavior. In *process enhancement*, a process model is dynamically enriched, with new information about the process based on new analysis of the process model and/or event log, e.g., detecting critical paths, predicting process performance indicators, repairing/simplifying of process models, etc.

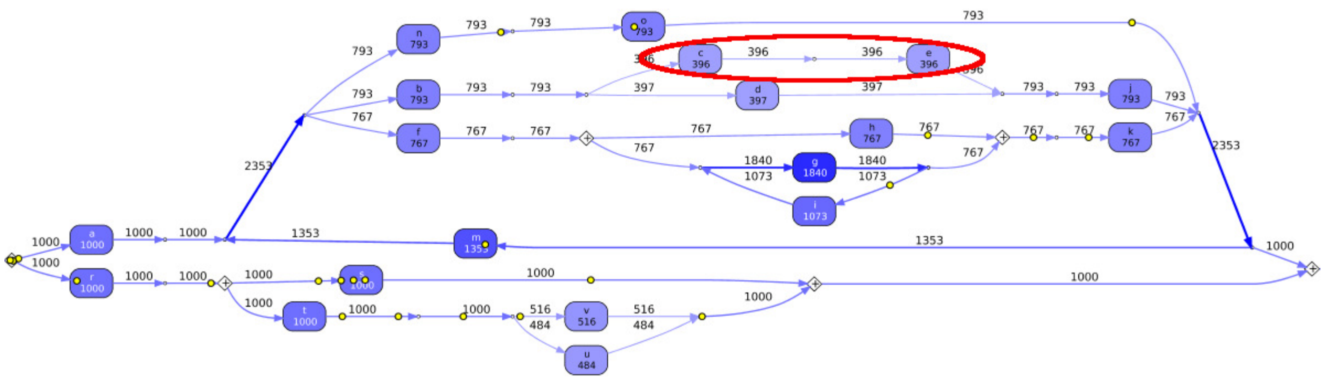
Both in conformance checking and process enhancement techniques, *alignments* (van der Aalst et al., 2012; Adriansyah et al., 2015; van Zelst et al., 2018a) have rapidly developed to a cornerstone technique and are often used heavily. Alignments quantify to what extent a process model and event data conform to each other. In order to do so, an alignment maps the behavior captured in an event log to a process model, relating each observed sequence of events, i.e., each *trace*, to a corresponding execution path of the process model. As an example of their use, consider the development of process mining algorithms such as *evolutionary process discovery algorithms* (Buijs, 2014; Vázquez-Barreiros et al., 2016a), where replay-fitness and precision (calculated on the basis of alignments) are used to evaluate the quality of a newly generated process model; *model repair techniques* (Polyvyanyy et al., 2017; Fahland & van der Aalst, 2015), where alignments are used for detecting the points in which a process model must be repaired such that it is accurately adapted to the observed behavior; or the *Inductive Visual Miner* (Leemans et al., 2014b), which uses alignments to visualize the flow of cases through a given process model.

Computing an alignment is an NP-hard problem. Several techniques have been proposed for alignment computation based on shortest-path search or optimization algorithms that look for *optimal alignments*, i.e., alignments with a minimal deviation cost (Adriansyah et al., 2011, 2013; Alizadeh et al., 2014; de Leoni et al., 2012; de Leoni & van der Aalst, 2013; van Dongen, 2018; de Leoni & Marrella, 2017; van Zelst et al., 2018a; Carmona et al., 2018). However, using these techniques in combination with realistically sized event logs and process models typically results in poor runtime performance. As a solution, some authors propose to decompose the process model into sub-models before applying search-based or optimization algorithms (Song et al., 2017; van der Aalst, 2013; Munoz-Gama et al., 2014). However, these decomposition techniques provide solutions for sub-problems, which in aggregated form *provide lower bounds*, i.e., underestimations of the true alignment costs.

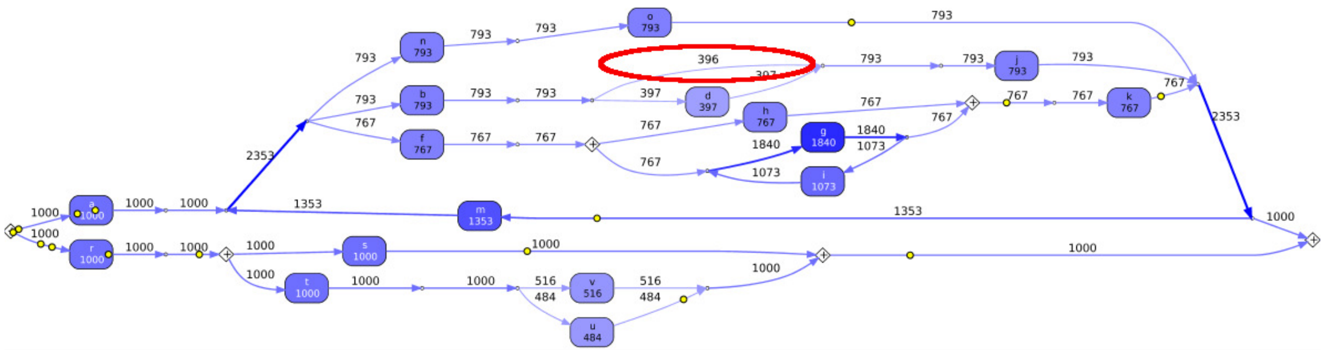
The previously mentioned process mining techniques compute alignments from scratch for new process models. However, in a variety of cases, these models are similar to one another. Relevant examples of such situation are:

- *Evolutionary process discovery.* This kind of algorithms lead to good results, discovering high quality process models, even in the presence of noise (van Eck et al., 2014; Vázquez-Barreiros et al., 2016a). In evolutionary process discovery there exists an initial population of process models that evolves over a number of iterations in which a new generation of process models is created by introducing slight modifications (crossover and mutation of the current generation of process models). In order to decide which process models are ruled out between two iterations, each one of them needs to be evaluated based on replay-fitness and/or precision, and therefore in each iteration there are a high number of evaluations. It is clear that this evaluation should be as efficient as possible to make evolutionary process discovery applicable to medium-large size event logs.
- *Visualizing trace executions.* The Inductive Visual Miner has a graphical interface that allows users to visualize a simulation of the execution of the traces (Leemans et al., 2014b). This simulation is based on alignments, as it highlights model paths related to trace executions. Furthermore, the graphical interface allows users to interactively filter noise. Such filtering often results in a similar process model compared to the current model. Consider Fig. 1, which shows the result of the Inductive Visual Miner twice, using a slightly different filtering setting. The only difference between the models is the absence of two activities highlighted by circles. Therefore, increasing the efficiency of alignment computation is a critical point for this algorithm in order to improve the user experience by changing thresholds and simulating trace runs. Observe that, a technique that allows us to repair alignments, can in principle be exploited in all interactive visualizations of alignments on process models.
- *Scenario Based Prediction.* Observe that, using alignments as a basis, i.e., explaining the event data in terms of a model, we are able to compute performance metrics on top of a given process model as well. In case a business owner aims to assess the expected impact of a certain change in his/her process, he/she usually changes small parts of the model, e.g., changing a parallel operator to a sequence operator, etc. Again in such a case, the models being compared are very similar to one-another.

Hence, the question arises whether we can use previously computed alignments as a basis for computing new alignments of similar process models, and thus potentially reduce alignment computation time. Therefore, in this paper, we propose an *alignment repair method* that computes alignments by repairing parts of existing alignments. The technique identifies fragments of the existing alignment that do not correspond to the process model and replaces them with new alignment fragments that do correspond. Because the method only focuses on those alignment fragments that do not fit, computation time decreases significantly. Moreover, we show that the loss of optimality is limited and stays within acceptable bounds. The proposed method is only applicable to sound process models, since the internal representation of the process models considered in this paper is based on *process trees*. We do so, since process trees allow us to represent sound models through a hierarchical structure in blocks, enabling a more efficient comparison between different models and, therefore, the location of those parts that have effectively change in relation to a similar model. Observe that, this feature prevents the applica-



(a) Process model with case visualization *before filtering*.



(b) Process model with case visualization *after filtering*.

Fig. 1: Application of filtering in the Inductive Visual Miner (Leemans et al., 2014b,a).

tion of our algorithm in unstructured processes, which are usually represented through non-sound models.

The main contributions of this paper are:

– The development of a novel and efficient method that computes alignments by *reusing* existing alignments for different, though similar, process models. The proposed method consists of three phases: *scope of change detection*, where the alignment part corresponding to the sub-model of the process model that has changed is identified; *realignment*, where the alignments related to the changes of the process model are computed; and *alignment reassembly*, where the alignments computed in the previous step are assembled as part of the original alignment. This method is specially interesting for complex, but similar, process models and when the size of traces is large.
– A validation of the method which shows that it retrieves alignments in a significantly lower, worst-case equal, time when compared to computing optimal alignments from scratch.

The remainder of this paper is structured as follows. Section 2 discusses related work. In Section 3, we present background concepts such as process trees, event data and alignments. In Section 4, we present our proposed alignment repair technique. In Section 5, we prove the correctness of our approach. In Section 6, we present an evaluation of the approach, whereas Section 7 concludes the paper.

2 Related Work

A broad overview of work in the field of process mining is outside the scope of this paper, hence we refer to (van der Aalst, 2016). Here, we primarily focus on related work in conformance checking.

Early work in conformance checking focuses on *token-based replay techniques* (Rozinat & van der Aalst, 2008). In token-based replay, markings and firing sequences of *Petri nets* (Murata, 1989) are used to computing conformance statistics. The techniques simulate traces through the model and produce, and keep track of, missing tokens in order to be able to fire transitions that are not enabled. The main disadvantage of token-based replay techniques is the fact that produced tokens are potentially used to enable future transitions, allowing for behavior that originally could not be performed within the model.

Alignments were introduced in (van der Aalst et al., 2012). The main challenge of alignments is their computation, which is an NP-hard problem. To deal with this issue two kind of approaches have been proposed: search-based techniques, which look for the alignment with minimum cost, and decomposition-based techniques, which decompose models into sub-models before applying search-based algorithms. We briefly review these approaches.

In (Adriansyah et al., 2011) the authors convert the alignment computation problem to a shortest path problem, based on the marking-based reachability graph of the Workflow net. Moreover, the authors propose the use of the A^*-algorithm (Hart et al., 1968), i.e., an algorithm that exploits a heuristic distance function to find a path with minimum cost in a weighted graph. In (Adriansyah et al., 2013) the authors improve the efficiency of the A^* approach of (Adriansyah et al., 2011) by defining a heuristic function based on the solution of the marking equation of the Workflow net through Integer Linear Programming (ILP). In (van Zelst et al., 2018a), a large scale experimental evaluation of the different parameters of the aforementioned A^* approach is presented. In (van Dongen, 2018) an alternative, A^*-inspired, search strategy is presented that exploits an extended version of the aforementioned marking equation. In (Alizadeh et al., 2014) the authors propose an alternative cost function based on information extracted from past process executions. The cost of an alignment depends on the move type and the activity involved in the move though, differently from (Adriansyah et al., 2013), it also depends on the position in which the move is inserted.

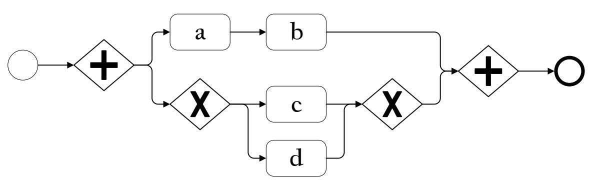
In (Song et al., 2017), the authors propose to analyse the structural and behavioral features of process models to reduce the search space by *(1)* decomposing the process model in a set of independent sub-models where a trace follows only one of the sub-models and *(2)* by simulating the execution of each trace in the sub-model to which it belongs to. Taking this into account, the authors present an algorithm based on effective heuristics relying on the trace to reduce the search space for computing the optimal alignment. Simple heuristics are considered for models with both iterative and alternative routing.

All the previous approaches calculate alignments solely based on the control-flow perspective. In (de Leoni & van der Aalst, 2013) the authors present a method for alignment calculation taking all perspectives into account: control-flow, data, time and resources. The first step of the proposal finds the control-flow alignment through A^* based on (Adriansyah et al., 2011). Then, an ILP problem is constructed to obtain an optimal alignment which also considers other perspectives of the process.

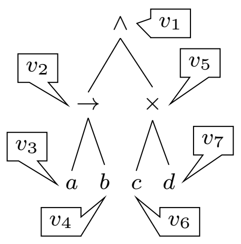
A different problem is conformance checking in declarative models. A declarative model lists constraints that specify the forbidden behavior, as opposed to imperative models, such as Workflow nets, which only describe allowed behavior. In (de Leoni et al., 2012) the authors propose calculation of alignments using A^* for declarative models. As the authors point out, the application of A^* for declarative models is more challenging than for procedural models, as the set of admissible behavior is far larger. Thus, the method implements a search space reduction based on the equivalence of partial alignments. Moreover, the approach provides metrics to measure the degree of conformance of single activities and constraints.

Decomposition techniques allow to approach conformance checking from another perspective (van der Aalst, 2013; Munoz-Gama et al., 2014). For instance, in (van der Aalst, 2012), the authors present an approach to decompose a model into net fragments which correspond to minimal passages. A passage is formed by two sets of nodes of a process model where the outputs of the first set are all inputs of the nodes in the second set, and the inputs of the nodes of the second set are all outputs of the nodes in the first set. Given this decomposition, it is possible to calculate the conformance in a distributed way. In (Fahland & van der Aalst, 2012, 2015), the authors propose a methodology to repair a process model through alignments. Based on alignment information, they decompose the log into several sub-logs that do not fit the original model. Finally, for each sub-log, a sub-process is derived and added to the original model in the appropriate location. In (de Leoni et al., 2014), the authors present a proposal for decomposing large data-aware conformance checking problems into smaller problems that can be solved more efficiently. The approach uses the Single-Entry Single Exit (SESE) decomposition (Munoz-Gama et al., 2014) to split the data-aware process model into smaller model fragments. These fragments are created by selecting a particular set of SESEs in the Refined Process Structure Tree (RPST) (Vanhatalo et al., 2009). To check the conformance of each fragment, the authors used the technique presented in (de Leoni & van der Aalst, 2013).

The main difference of this work compared to related work is the fact that the technique presented in this paper results in an alignment for the *whole trace* and the *whole process model* reusing previously computed alignments.



(a) The process model in BPMN notation.



(b) The process model visualized as a process tree (which we refer to in the remainder as PT_1).

Fig. 2: Two process models describing the parallel execution of a sequence of activities a and b, together with a choice between activities c and d.

3 Background

In this section, we present background material used throughout the remainder of this paper. We focus on process trees as a modelling formalism as well as the notion of alignments.

3.1 Process Trees

In this paper, we focus on hierarchical process models, i.e., *process trees* (Buijs, 2014; Leemans et al., 2013), which are known to be *sound by design*. A process tree is a compact tree-like *representation of a Workflow net* (van der Aalst, 1998). Process trees allow us to represent sound process models through a hierarchical structure in structured blocks, which makes the comparison between two different models relatively efficient. Consider Fig. 2, in which we present a simple process model in both BPMN notation and its corresponding process tree visualization.

The models describe that activities a and b need to be executed in sequence, i.e., first activity a, then activity b. Moreover, either activity c or activity d is executed. This can be done concurrently with executing the sequence of activities a and b. The leafs of a process tree always represent (possibly unobservable by means of τ-labels) *activities*, whereas internal vertices always represent *operators* used to specify the relation between their children. Each vertex within a process tree defines a process tree itself.

In this paper we consider five standard operator types, similar to the work of (Buijs, 2014), defined for process trees: the sequential operator ($\rightarrow$), the parallel execution operator ($\wedge$), the exclusive choice operator ($\times$), the non-exclusive choice operator ($\vee$) and the repeated execution (loop) operator ($\circlearrowleft$). Operators have an arbitrary number of children in arbitrary order, except for the sequence and loop operators. The sequence operator has an arbitrary number of children, though the order of the children specifies the order in which they must be evaluated, i.e., from left to right. Loop operators always have three children. The left child is the *do-child* of the loop and is always executed, the middle child is the *redo-child* and is optional, the right child is the *exit-child* and is also always executed. Whenever the redo-child is executed, it has to be followed by the do-child. Whenever the exit-child is executed the operator terminates. For example given a simple process tree $\circlearrowleft(a, b, c)$, example behavioral sequences described by the tree are $\langle a, c\rangle$, $\langle a, b, a, c\rangle$, $\langle a, b, a, b, a, c\rangle$, etc. Furthermore, example behavioral sequences described by the process tree depicted in Fig. 2, are: $\langle a, b, c\rangle$, $\langle a, b, d\rangle$, $\langle c, a, b\rangle$, $\langle a, d, b\rangle$, etc.

3.2 Event Data and Alignments

Modern information systems track the execution of business processes within a company. These systems store the execution of business activities in context of a *case*, i.e., an instance of the underlying process. Such data is often stored in the form of an *event log*. An event log records the *actual execution* of activities within a business process. Consider Table 1 depicting a snapshot of an event log of a loan application process.

The actual execution of a business process activity is referred to as an *event*, which is unique. A sequence of events is referred to as a *trace*. In the context of this paper we are merely interested in the sequential ordering of the business process activities recorded in traces, i.e., the *control-flow perspective*. Observe that, when adopting the control-flow perspective, we obtain the trace of activities ⟨Check application form, Check credit history, ..., Reject application⟩ for the process instance identified by case-id 3554.

Alignments (van der Aalst et al., 2012; Adriansyah, 2014) allow us to explain observed behavior, during the execution of a process, in terms of a given process model. Alignments map the observed business process

Table 1: Event log fragment based on a simple fictional loan application process (Dumas et al., 2018).

Case-id	**Activity**	**Resource**	**Time-stamp**
⋮	⋮	⋮	⋮
3554	*Check application form*	*John*	*2015-10-08T09:45:37*
3555	*Check application form*	*Lucy*	*2015-10-08T10:12:37*
3554	*Check credit history*	*Harold*	*2015-10-08T10:14:25*
3555	*Check credit history*	*Harold*	*2015-10-08T10:31:02*
3554	*Appraise property*	*Pete*	*2015-10-08T10:45:22*
3554	*Assess loan risk*	*Harold*	*2015-10-08T10:49:52*
3555	*Assess loan risk*	*Harold*	*2015-10-08T11:01:51*
3556	*Check application form*	*Lucy*	*2015-10-08T11:05:10*
3555	*Assess eligibility*	*Harry*	*2015-10-08T11:06:22*
3554	*Assess eligibility*	*Harry*	*2015-10-08T11:33:42*
3554	*Reject application*	*Harry*	*2015-10-08T11:45:42*
3557	*Check application form*	*Lucy*	*2015-10-08T13:48:12*
3555	*Prepare acceptance pack*	*Sue*	*2015-10-08T14:02:22*
⋮	⋮	⋮	⋮

$-$	$-$	a	b	$-$	$-$	c	$-$	$-$	d	e
v_1^s	v_2^s	v_3	v_4	v_2^e	v_5^s	v_6	v_5^e	v_1^e	$-$	$-$

(a) Alignment γ_1

$-$	$-$	$-$	a	b	$-$	c	d	$-$	$-$	e
v_1^s	v_5^s	v_2^s	v_3	v_4	v_2^e	$-$	v_7	v_5^e	v_1^e	$-$

(b) Alignment γ_2

$-$	$-$	$-$	$-$	a	b	$-$	c	d	$-$	$-$	e
v_1^s	v_5^s	v_2^s	v_3	$-$	v_4	v_2^e	$-$	v_7	v_5^e	v_1^e	$-$

(c) Alignment γ_3

Fig. 3: Three possible ways to align $\sigma_1 = \langle a, b, c, d, e \rangle$ to PT_1.

events to the activities in a process model. Such an individual mapping is referred to as a *move*. We observe three types of moves, i.e., *synchronous moves*, mapping observed behavior onto activities described by a process tree, *model moves*, referring to behavior in the process tree that is not observed in the data, and *log moves*, indicating that we are not able to map observed behavior onto an element of the process tree.

As an example, consider Fig. 3, in which we depict three possible alignments of the trace $\langle a, b, c, d, e \rangle$ and the process tree depicted in Fig. 2b. The first move of Fig. 3a, i.e., $(-, v_1^s)$, refers to enabling/starting the root vertex of the tree, i.e., vertex v_1.[1] Since v_1 is an internal vertex, we are not able to observe it, hence, $(-, v_1^s)$ *always represents a model move.* We use the $-$ symbol to indicate that we are not able to construct a mapping. Similarly, the second move of the alignment, i.e., $(-, v_2^s)$, is a model move, referring to enabling/starting internal vertex v_2. The third move represents a synchronous move on activity a, which is mapped to the execution of vertex v_3, which indeed has label a. Similarly, the fourth move represents a synchronous move on activity b. After this, we observe move $(-, v_2^e)$, indicating that the execution of the subtree formed by vertex v_2 has ended. The last two moves of Fig. 3a are log moves, i.e., we are not able to map d onto the execution of vertex v_7, because it is in a choice construct with vertex v_6 of which we chose to map observed activity c on. Furthermore, since label e is not present in the model, it is guaranteed to always show up as a log move.

A sequence of moves, i.e., such as presented in Fig. 3a, is an alignment, if the "top part", when excluding the $-$ symbols, equals the input trace. Secondly, the "bottom part", again when excluding the $-$ symbols, needs to correspond to a feasible execution of the process tree. Observe that, indeed, the sequence of moves depicted in Fig. 3a, is an alignment. Note that, for a given trace, several different alignments exist. Consider Fig. 3b, in which we show an alternative alignment of trace $\langle a, b, c, d, e \rangle$ and process tree PT_1. W.r.t. Fig. 3a, vertex v_5 is started prior to vertex v_2. Observe that this is allowed due to the fact that vertex v_1 describes a parallel operator. Moreover, the alignment synchronises on activity d, rather than activity c.

Observe the alignment in Fig. 3c, in which we describe a model move on vertex v_3 and a log move on activity a. Furthermore, observe that this is again a proper alignment of trace $\langle a, b, c, d, e \rangle$ and process tree PT_1. However, this is a less desirable alignment compared to the alignments presented in Fig. 3a and Fig. 3b, i.e., since it is possible to synchronize on a. For alignments γ_1 and γ_2 it is less obvious which one is favoured over the other one or if both alignments are equally favourable. Thus, we need a means to grade/score alignments in terms of their quality. Therefore, we typically use a *cost-function*, defined on top of the different types of possible moves, which allows us to find the most desir-

[1]We use v^s and v^e to represent the *start*, respectively *end* of an internal vertex of a process tree.

able alignment (also referred to as *optimal alignment*). Usually we adopt the following cost function (known as the standard cost function):

– *synchronous moves/internal model moves/invisible leaf model moves*: cost 0.[2]
– *log moves/visible leaf model moves*: cost 1.

Observe that, using the cost function as presented, the cost of the alignments in Fig. 3a and Fig. 3b is 2 (two log moves), whereas the cost for the alignment in Fig. 3c is 4 (three log moves and one leaf-based model move). The problem of computing an optimal alignment can be translated to a shortest path problem. In (Adriansyah, 2014) a solution to this shortest path problem, for the purpose of arbitrary Petri nets, is presented by applying the A^* algorithm (Hart et al., 1968), i.e., an algorithm that exploits a heuristic distance function to find a path with minimum cost in a weighted graph. As this solution method trivially applies to process trees, in the context of this paper, we assume that we are able to compute an optimal alignment for arbitrarily given trace and process tree.

4 Repairing Alignments

Several process discovery techniques build on top of alignments and use process trees as a process modelling formalism. These techniques compute alignments for a given (set of) process model(s) and subsequently (re)compute alignments for very similar process models. Moreover, the fact that these techniques use process trees as a process model formalism, as opposed to arbitrary Workflow nets, allows us to efficiently pinpoint the similarity between two given process models. We therefore propose a method that allows us to repair readily available alignments of a given trace and process model, for newly obtained, preferably similar, process trees.

In the remainder of this section, we describe the proposed repair algorithm. In this context, we assume that we are given a trace σ, a process tree PT and an alignment γ of the trace and the process tree. Moreover, we assume that we are given an alternative process tree PT' which is the result of changing a sub-tree of PT with some alternative sub-tree. The proposed alignment repair technique exploits the process models' similarity and produces an alignment γ' for trace σ and process tree PT'. A global overview of the approach is presented in Fig. 4.

The approach consists of three main stages:

[2]If a leaf vertex v has label τ, it is unobservable by definition, which always leads to model move $(-, v)$.

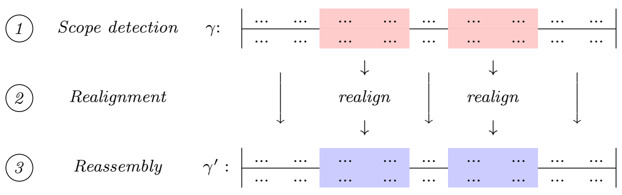


Fig. 4: Schematic overview of the repair approach.

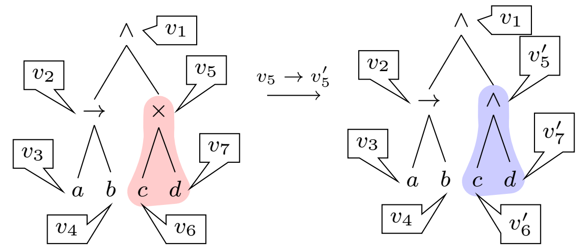


Fig. 5: Modification of sub-tree PT_1 into PT_2 by replacing v_5.

1. *Scope of change detection.* In this step we identify moves in the existing alignment that correspond to behavior of the changed sub-tree. In particular, we identify what label-based-moves, i.e., log and/or synchronous moves, are likely to become/stay synchronous moves based on the new sub-tree.
2. *Realignment.* In this step we compute new alignment fragments based on the labelled moves identified in the previous step and the new sub-tree.
3. *Alignment reassembly.* In this step we replace the moves related to the changed sub-tree in the original alignment by their corresponding new alignment fragments obtained in the previous step to form the new, repaired, alignment.

In the upcoming subsections we describe each step in more detail. Prior to this, we present a running example that we use throughout this section to clarify each step.

Running Example We use the modification of process tree PT_1 into PT_2, shown in Fig. 5, as a running example. We change vertex v_5, which is a $\times$ operator, into vertex v_5', which is a $\wedge$ operator. The new nodes generated by the change are v_5', v_6' and v_7'. Note that vertices v_6' and v_7' have the same label as vertices v_6 and v_7. The change enforces us to always fire both branches corresponding to leaf nodes v_6' and v_7' concurrently. Reconsider trace $\sigma = \langle a, b, c, d, e\rangle$. We reuse the optimal alignment γ_1 for the sequence and process tree PT_1 presented in Fig. 3a, to compute a new alignment of $\langle a, b, c, d, e\rangle$ and PT_2.

$-$	$-$	a	b	$-$	$-$	c	$-$	d	e	$-$
v_1^s	v_2^s	v_3	v_4	v_2^e	v_5^s	v_6	v_5^e	$-$	$-$	v_1^e

Fig. 6: Identification of the moves that trivially belong to the scope of v_5.

...	$-$	d	$-$	e	...
...	t_{15}	$-$	v_5^e	$-$	...

Fig. 7: In any alignment, we are able to swap log and model moves, without jeopardizing the alignment, e.g., swapping $(-, v_5^e)$ and $(d, -)$ in the context of Fig. 3a.

4.1 Scope of Change Detection

The first step in reusing γ_1, involves detecting what moves in γ_1 refer to the changed sub-tree, i.e., the sub-tree defined by v_5. We refer to the collection of these moves as *the scope of change* of v_5. We do so by collecting all moves in the alignment that directly relate to the changed subtree, combined with adjacent log moves. In particular, for these adjacent log moves, only model moves are allowed to be in-between the moves related to the changed subtree and the log moves themselves.

Consider that a naive way to construct the scope of change is to only include moves of the form (x, v) within γ_1 s.t. $v \in \{v_5^s, v_5^e, v_6, v_7\}$, i.e., both synchronous and model moves, as part of the scope of change. In Fig. 6, these type of moves are highlighted in terms of γ_1. However, if we only use such trivial moves, we obtain sub-optimal results. The second step of the approach concerns computing a new alignment based on the activities present in the scope of change. Since in this case the only activity present in the scope of change is c, we compute an alignment of sequence $\langle c \rangle$ and the new sub-tree defined by v_5'. Observe that such an alignment contains a synchronous move (c, v_6') and a model move $(-, v_7')$, i.e., a model move on the vertex labelled with activity d. However, in alignment γ_1, the move next to $(-, v_5^e)$ is a log move on d, i.e., $(d, -)$. If we assign the log move to the scope as well, we end up with sequence $\langle c, d \rangle$. In such case both vertices v_6' and v_7', after aligning $\langle c, d \rangle$ with the sub-tree defined by v_5', relate to synchronous moves, i.e., (c, v_6') and (d, v_7'). Thus, it is beneficial to include adjacent log moves within the scope of change.

Let m^s and m^e denote the *moves* related to the unique start- and end transition of the changed sub-tree, i.e., $(-, v_5^s)$ and $(-, v_5^e)$ in case of the running example. Consider log moves in-between m^s and m^e. We know that within that position of the alignment, behavior of the subtree is allowed. If we assign log moves in-between m^s and m^e to the scope of change and in step 2 use their labels to compute a new alignment based on the new sub-tree, these moves either stay log moves or become synchronous. Thus, the overall contribution of these log moves to the alignment cost can only decrease, which is desirable. We therefore deduce that any log move in-between m^s and m^e is eligible to be part of the scope.

However, our previous example shows that log moves that are not in-between m^s and m^e are also interesting to use within the scope, i.e., $(d, -)$ in case of alignment γ_1. Observe that when swapping a log- and a model move within an alignment, none of the two requirements as presented in Section 3.2 is violated, i.e., the activity sequence (top part) still describes the trace, and the behavioral sequence (bottom part) is still a feasible execution of the process tree. Hence, trivially, we deduce that we are able to swap log-moves and model moves in any alignment. Thus, in the context of alignment γ_1, if we swap the moves $(-, v_5^e)$ and $(d, -)$ (cf. Fig. 7), the newly obtained sequence of moves is still an (optimal) alignment.

By applying such a swap, move $(d, -)$ is positioned in-between the moves related to the unique start- and end transition and thus eligible for inclusion in the scope. Obviously, we are able to apply the same trick for move $(e, -)$. However, in general, we are not able to swap all possible moves, i.e., we are not able to swap:

1. Log moves with log moves, as we have to respect the order of the events in the trace.
2. Model moves with model moves, as the process model demands a specific execution ordering.[3]
3. Synchronous moves with any other type of move, i.e., synchronous moves, log moves or model moves.

For example, we are not allowed to swap (c, v_6) with $(-, v_5^e)$. Based on the previous observation, we observe that any log move m_l that occurs after move m^e s.t. there are only model moves in-between m^e and m_l can be swapped such that it precedes m^e. Moreover, an other log move m_l' that occurs after m_l, and, due to swapping of m_l now only has model moves in-between m^e and itself can subsequently be swapped such that it precedes m^e. As an example consider moves $(d, -)$ and $(e, -)$, i.e., after swapping $(d, -)$ with $(-, v_5^e)$ we are subsequently able to swap $(e, -)$ and $(-, v_5^e)$. Symmetrically, this also holds for moves m_l that precede move m^s, i.e., we are also able to swap these move in-between m^s and m^e.

Thus, given aforementioned move m^s and corresponding move m^e at position i, respectively j in some

[3]Due to parallelism, in some cases we are allowed to swap model moves with other model moves or synchronous moves, as the process model allows several execution orderings. This does however not hold in the general case.

$-$	$-$	a	b	$-$	$-$	c	$-$	d	e	$-$
v_1^s	v_2^s	v_3	v_4	v_2^e	v_5^s	v_6	v_5^e	$-$	$-$	v_1^e

Fig. 8: Final result of scope of change detection.

$-$	c	d	$-$	e
$v_5'^s$	v_6'	v_7'	$v_5'^e$	$-$

Fig. 9: Alignment of $\langle c, d, e\rangle$ on the new sub-tree formed by v_5'.

alignment γ, the following moves belong to the scope of change:

1. Model/synchronous moves at position i' s.t. $i < i' < j$ that relate to the changed sub-tree.
2. Any log move at position $i < i' < j$.
3. Any log move at position $i' < i$ s.t. there is no synchronous move at position i'' with $i' < i'' < i$.
4. Any log move at position $i' > j$ s.t. there is no synchronous move at position i'' with $j < i'' < i'$.

In Fig. 8, we illustrate the final result of scope identification for γ_1.

4.2 Alignment Recalculation

In this section, we describe step 2 of the approach, i.e., alignment recalculation, which is trivial. We obtain the log moves and the synchronous moves part of the scope of change and we project these moves onto their label values. Subsequently we simply compute a new alignment for the generated subsequence of behavior. In case of our running example, this results in the alignment depicted in Fig. 9. Subsequently, the main challenge concerns placing the moves of the new alignment back into the old alignment at adequate positions.

4.3 Alignment Reassembly

In this section, we describe the final step of the approach, in which we replace the scope of change by parts of the newly obtained alignment. When the scope of change is not within a parallel construct, such reassembly is trivial, i.e., we simply paste the new fragment starting at the same position as the scope of change. However, in case the scope of change resides in a parallel block, i.e., one of its ancestors in the tree is an $\wedge$- or an $\vee$-operator, it is likely that the moves of the scope of change are interleaving with moves outside of the scope. Hence, when replacing the scope of change with the newly obtained alignment fragment, we need to ensure that each move of the new alignment fragment is placed on the right position, i.e., in order not to break the overall alignment.

We replace the scope of change by the newly computed alignment fragment, on the basis of pointers. We store a pointer for each move m in the scope of change that relates to an activity observed in the trace, and, the first move in the scope of change that relates to behavior in the subtree, e.g., v_5^s in case of our running example. We do so, as we are able to relate moves in the newly obtained alignment fragment back to these moves in the scope of change. For each move in the scope of change, the pointer structure is constructed as follows:

1. If it is the first model/synchronous move related to the changed subtree, e.g., $(-, v_5^s)$ in the context of the running example, we store a pointer to the closest preceding move, i.e., $(-, v_2^e)$ in the context of our example.
2. If it is a log/synchronous move, e.g., (c, v_6) and $(d, -)$ in the context of the running example, we store a pointer to the closest preceding log/synchronous move. For example, for (c, v_6), we store a pointer to (b, v_4).

Consider the upper alignments of Fig. 10 and Fig. 11 respectively, in which we visualize the aforementioned pointer structure in the context of the running example. We use double-headed arrows to represent such pointers.

When replacing the scope of change by the new alignment fragment, we walk through the new alignment fragment step-by-step. For each move we encounter, we check whether there exists a pointer stored in the corresponding move in the scope of change. For example, in Fig. 10, the first move of the new alignment fragment is $(-, v_5'^s)$. Clearly, this move relates to the first model/synchronous move in the scope of change, i.e., $(-, v_5^s)$. Based on the pointer stored for $(-, v_5^s)$, i.e., pointing to $(-, v_2^e)$, we start inserting the newly obtained alignment fragment in the original alignment. We subsequently inspect the next move in the newly obtained alignment fragment. In case this is a model move, it does not have a corresponding counter part in the scope of change, and we append it to the previously inserted move. However, if this either a synchronous or a log move, there exits a corresponding pointer in the scope of change. For example, in Fig. 10, the second move in the new alignment fragment is (c, v_6'), for which its corresponding move in the scope of change has a pointer to move (b, v_4). Hence, we need to make sure that when placing (c, v_6') into the alignment, it is the next synchronous/log move after (b, v_4). Observe that, in Fig. 10, this is indeed the case, i.e., (c, v_6') is the first log/synchronous move occurring after (b, v_4), hence, we

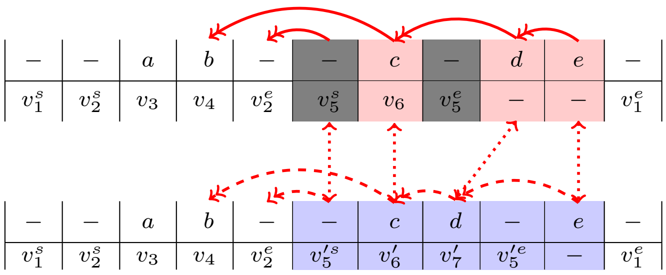

$-$ $-$ a b $-$ $-$ c $-$ d e $-$
v_1^s v_2^s v_3 v_4 v_2^e v_5^s v_6 v_5^e $-$ $-$ v_1^e
$-$ $-$ a b $-$ $-$ c d $-$ e $-$
v_1^s v_2^s v_3 v_4 v_2^e $v_5'^s$ v_6' v_7' $v_5'^e$ $-$ v_1^e

Fig. 10: Repositioning of the new alignment fragment in the existing alignment, in case there is no interference with parallel behavior. Since there is no interleaving between the scope of change and other parts of the model, we are able insert the new alignment fragment as a consecutive block.

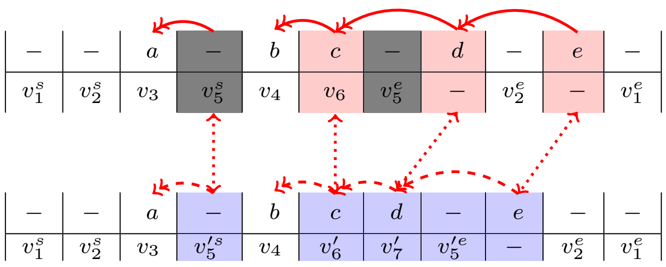

$-$ $-$ a $-$ b c $-$ d $-$ e $-$
v_1^s v_2^s v_3 v_5^s v_4 v_6 v_5^e $-$ v_2^e $-$ v_1^e
$-$ $-$ a $-$ b c d $-$ e $-$ $-$
v_1^s v_2^s v_3 $v_5'^s$ v_4 v_6' v_7' $v_5'^e$ $-$ v_2^e v_1^e

Fig. 11: Repositioning of the new alignment fragment in the existing alignment, in case there is interference with parallel behavior. After pasting the first move of the new alignment fragment, we need to skip move (b, v_4) and paste (c, v_5') directly after it.

do not need to shift the insertion point and can proceed to the next move. For the next move in the newly obtained alignment fragment, we repeat the procedure.

In Fig. 10, the scope of change is a consecutive block of moves. As a result, we are able to insert the newly obtained alignment fragment as a consecutive block as well. However, as indicated, this is not always the case. Consider Fig. 11, in which we present an alternative alignment of trace $\langle a, b, c, d, e\rangle$ and PT_1. In this case, move $(-, v_5^s)$ occurs prior to move (b, v_4). Furthermore, move $(-, v_2^e)$ occurs in-between moves $(d, -)$ and $(e, -)$. When inserting the new alignment fragment, we start with its first move, i.e., $(-, v_5'^s)$, which we, on the basis of the stored corresponding pointer, position directly after (a, v_3). The next move in the fragment is (c, v_6'). As the corresponding move (c, v_6) occurs after move (b, v_4), we start inserting from there, rather than directly after $(-, v_5'^s)$. All subsequent moves are in the right position and are therefore inserted in a consecutive manner.

Note that, the procedure described, i.e., consisting of scope detection, realignment and reassembly, works for *every described execution* of the changed subtree. In case the changed subtree is in a loop structure, i.e., on the path from the root of the process tree to the root of the changed subtree there occurs an $\circlearrowleft$ operator, it is potentially executed multiple times. Hence, we executed the aforementioned procedure for each individual execution of the subtree.

5 Correctness and Optimality

In the examples used in Section 4, the repaired alignments are in fact alignments, i.e., they respect the requirements laid out for alignments in Section 3.2. Moreover, they are optimal. In this section we show the correctness of the general approach, i.e., that a repaired alignment is always an alignment. Moreover we show, by means of a counter example, that we are not able to guarantee optimality.

5.1 Correctness

The basic correctness requirement of the presented approach is that, after reusing an existing (optimal) alignment, the repaired alignment itself is an alignment. To prove that a repaired sequence of moves γ' is an alignment, we need to prove that the two basic requirements presented in Section 3.2 hold for γ'. In this section, we show that his indeed holds.

Consider the first requirement, i.e., projection of the moves onto activities yields the trace. Observe that the repair method inserts alignment fragments back into the original alignment based on *pointers*. Observe that, due to the use of the pointers, a move is never placed at a relative *earlier* position, i.e., if the insertion index is too small, we use the pointers to shift it to the correct position, e.g., as exemplified in Fig. 11. Thus, the only problem that potentially jeopardizes the property, is a label-based move m_l that is placed relatively *too far back*, i.e., there appears (at least) one label-based move m_l' in-between m_l and m_l's actual preceding event in the trace. However, this only happens if we shift the pointer too far, which in turn only happens if two label-based moves are swapped by the underlying alignment algorithm. This contradicts that the underlying alignment algorithm guarantees to return alignments. Thus, the moves are always placed back in correct order.

For the second requirement, we need to show that projection on the model-part of the alignment is in the newly created process tree's language. Let m^s denote the first move of the scope of change, that relates to starting behavior of the changed subtree, e.g., $(-, v_5^s)$ in Fig. 10 and Fig. 11, i.e., the first non-log move of the scope of change. Furthermore, let m' be the closest non-log move preceding m^s, i.e., relating to execution of some other behavior in the tree, e.g., $(-, v_2^e)$ in Fig. 10 and (a, v_3) in Fig. 11 respectively. Symmetrically we define m^e as the final move of the scope of change relating to the behavior in the changed subtree, and we let m'' denote the first non-log move succeeding m^e, e.g., $(-, v_5^e)$ and $(-, v_1^e)$ in Fig. 10.

Since move m' and m'' do not relate to the scope of change, they remain present in the resulting alignment. Furthermore, all the moves within the scope of change that relate to behavior in the changed subtree, occur in-between moves m' and m''. Due to using the explicit pointer related to the start of the changed subtree, the first move in the new alignment related to behavior of the newly inserted subtree, occurs directly after m'. Furthermore, it is impossible to insert some moves of the new alignment, related to behavior of the new subtree, after m''. Observe that this is the case, because we only shift the insertion of the alignment fragment due to the existence of a pointer on the basis of a log/synchronous move. Assume that such a pointer exists to a move m_p that occurs after m''. Move m_p can only be a log move, if there is no synchronous move in-between m'' and m_p. However, in that case, m_p itself is part of the scope of change, which contradicts the possibility of the existence of a pointer to m_p. If m_p is a synchronous move, we have assigned log moves occurring after the synchronous move to the scope of change, which is not allowed, i.e., the scope of change stops when we observe the first synchronous move occurring after m^e. Hence,

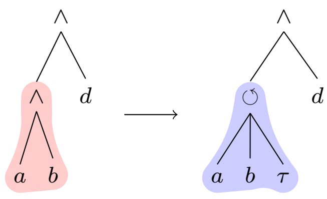


Fig. 12: Example change of a process tree from a concurrent operator to a loop operator.

we are guaranteed that the newly generated alignment fragment is reinserted in-between m' and m''.

Since the original alignment is a proper alignment, we know that the behavior of the changed subtree is allowed to occur in-between the moves m' and m''. Hence, by construction of process trees, the behavior of the newly generated subtree is also allowed to occur at that position. In case there exists, due to parallelism, interleaving of moves outside of the changed subtree in-between m' and m'', we are allowed to arbitrary shuffle that interleaving behavior (subject to not shuffling label-based moves). Hence, any interleaving occurring after inserting the newly generate alignment fragment relates to the existence of parallelism and is allowed as well.

5.2 Optimality

In this section, we show that we are not able to guarantee optimality of the proposed approach. We show this by means of a simple counter example, which also shows that optimality is partially depending on the form of the original alignment.

Consider the simple process tree in Fig. 12. Assume we align the trace $\langle a, b, a, d, b, a, b\rangle$ on the left process tree in Fig. 12. Observe that a possible optimal alignment of $\langle a, b, a, d, b, a, b\rangle$ and the left process tree of Fig. 12, is constructed by making the first three events log moves, making the d event the first synchronous move, the subsequent b event a log move again, and the final two events, i.e., $\langle a, b\rangle$ synchronous. Additionally we require that, in the underlying alignment, the start of sub-tree $\wedge(a, b)$ occurs after the synchronous move on the d event.

We now change the process tree and obtain the process tree depicted in the right-hand side of Fig. 12. When we apply the proposed repair algorithm, the log moves prior to the d-event, i.e., the first three events $\langle a, b, a\rangle$ are not incorporated in the scope of change. These moves therefore stay log moves. However, the given trace perfectly fits the new process model in Fig. 12. This shows that the proposed technique is not able to guarantee optimality of the resulting alignments.

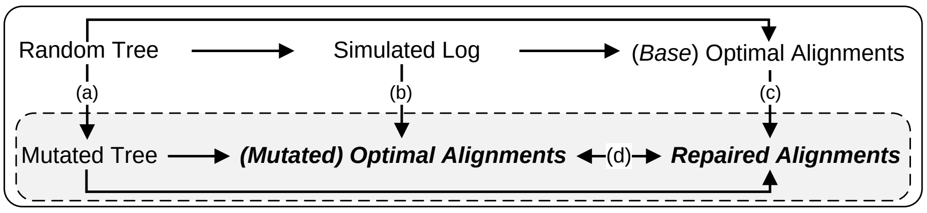

Random Tree
Simulated Log
(*Base*) Optimal Alignments
(a)
(b)
(c)
Mutated Tree
(Mutated) Optimal Alignments
(d)
Repaired Alignments

Fig. 13: Process followed during the experimentation.

6 Evaluation

To evaluate the proposed technique, we answer two main questions: *(1)* What is the time needed to align a model and a log with the presented technique? and *(2)* How close/far is the repaired alignment from the optimal alignment? In this section we answer these questions by comparing the time needed for alignment repair with the time expended to compute a new, optimal alignment and by measuring the quality of the repaired alignments w.r.t. the new, optimal alignment. Finally, we investigate the actual impact of the proposed approach on evolutionary process discovery using a real event log.

Implementation Part of the experimental results shown in this section are based on experiments performed for (Vázquez-Barreiros et al., 2016b). Moreover, the newly added experiments for the purpose of this paper are based on the code-base of (Vázquez-Barreiros et al., 2016b)[4]. In the code-base, the number of log moves that are adopted in the scope are only those log moves that directly border a synchronous/model move that belongs to the changed sub-tree. Moreover, also pointers are stored if there are model moves in-between two scope moves. Thus, as opposed to the more generic approach presented in this paper, within the code some log moves may be left out of the scope. This has an expected negative impact on the alignment optimality of the implementation, i.e., we expect it to be equal or slightly worse w.r.t. the general approach.

6.1 Experimental Set-Up

In Fig. 13 we depict a schematic overview of the experimental setup. We generate an initial *random* process tree of random size. Based on this model, we simulate a non-fitting event log, i.e., the event log contains noise, consisting of 2000 traces. We then calculate the optimal alignments of all traces in the event log w.r.t. the initial model. As a second step, we perform a set of random changes on the base model (step a in Fig. 13), generating a total of 150 *different* mutated process trees. We enforce that every mutated model is unique. The possible changes applied over the base model are: randomly adding a new node, randomly removing a node and randomly changing a node of the tree. Then, we calculate two different types of alignments for each mutated tree: optimal alignments based on the simulated log (step b in Fig. 13) and repaired alignments reusing the optimal alignments previously calculated on the base model (step c in Fig. 13). Finally, we compare both outputs (step d in Fig. 13).

Following this process, we created a set of 50 initial *random* trees with arbitrary sizes between 21 and 47 vertices. Thus, we applied the presented technique over $50 \times 150 \times 2000 \approx 1.5 \cdot 10^6$ alignments[5].

6.2 Running Time

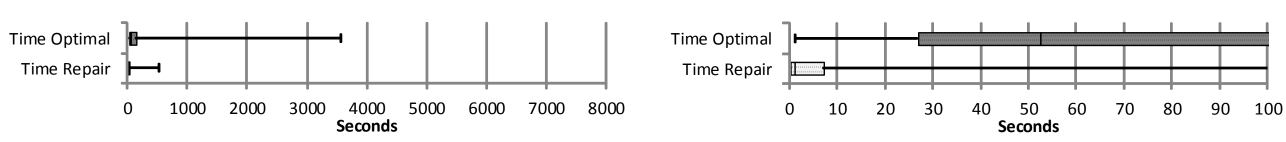
As the time needed to compute alignments varies significantly between runs, we grouped the results of the experiments based on the size of the *initial random process trees*. We created a bucket with initial trees of sizes between 21 and 28 vertices (12 trees in total), a bucket with sizes between 29 and 31 vertices (12 trees in total), a bucket with sizes between 32 and 34 vertices (13 trees in total) and a bucket with sizes greater than 35 vertices (13 trees in total).

Fig. 14 shows the time comparison, using box plots, for each bucket of experiments. Due to the high dispersion of the data, on the right-hand side of Fig. 14 we also show the box plots zoomed into the domain 0-100 seconds.

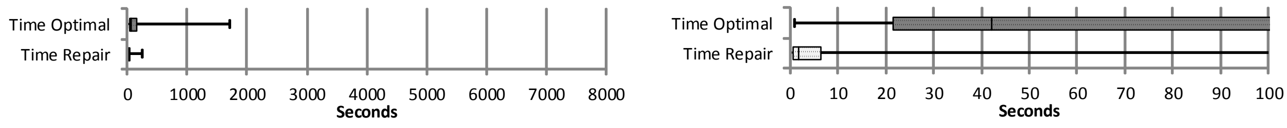
Consider results shown in Fig. 14a. When inspecting the time needed for computing optimal alignments, i.e., *Time Optimal*, we observe that in the middle 50% of the runs (Q2,Q3) it roughly took between 25 and 145 seconds to align an event log and a model. The fastest 25% of the experiments (Q1, left whisker) took less than 30 seconds, whereas the slowest 25% of the experiments

[4]`https://svn.win.tue.nl/repos/prom/Packages/EvolutionaryTreeMiner/Branches/BorjaImp/experiments/`

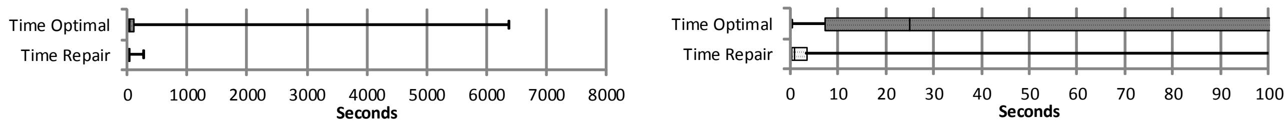
[5]All results can be found at `https://svn.win.tue.nl/repos/prom/Packages/EvolutionaryTreeMiner/Branches/BorjaImp/experiments/bpmds2016/`.



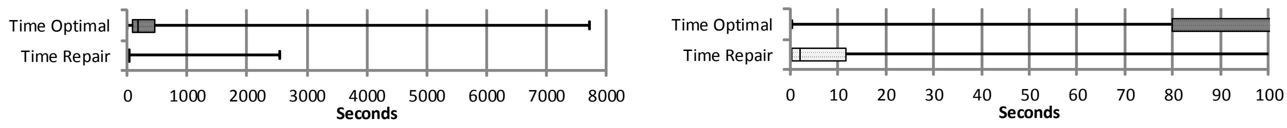
(a) Trees of size with less than 28 vertices.



(b) Trees of size between 29 and 31.



(c) Trees of size between 32 and 34.



(d) Trees with more than 35 vertices.

Fig. 14: Box plots showing the time needed to repair an alignment versus computing the optimal alignments for each bucket of experiments. The right-hand side shows the results zoomed into the domain 0-100 seconds.

(Q4, right whisker) took more than 150 seconds. Thus, in the 75% of the experiments it took more than 30 seconds to align a log and a model and only in the remaining 25% less than 30 seconds. On the other hand, for alignment repair, i.e., *Time Repair*, the middle 50% of the experiments (Q2, Q3) roughly took between one and seven seconds to align an event log and a model. In the fastest 25% of the experiments it took less than a second whereas in then the slowest 25% of the experiments computation time took more than seven seconds. If we compare both techniques, aligning a log and a tree with the presented technique took less than seven seconds in the 75% of the cases, whereas for computing the optimal alignments, only in the 25% of the experiments this took less than 30 seconds. The same pattern is visible in the other results presented in Fig. 14.

In general we observe that there is *no overlap* in the second and third quartiles of computing alignments based on the repair method versus computing an optimal alignment from scratch. This implies that in nearly all cases, the time needed to align a model and an event log by applying alignment repair outperforms computing a new optimal alignment.

The time needed for alignment repair seems directly related to the size of the changed sub-tree, which explains the rather high range of the right whiskers in the box plots for alignment repair. Clearly, if the change is performed in the root node of a process tree, the time needed to apply the presented approach will be roughly equal to the time needed to compute the optimal alignment as there is no room to repair the old alignment. Thus, we conclude that using the presented technique, guarantees a lower, or, in worst case equal, running time compared with computing the optimal alignments between an event log and a process tree from scratch.

6.3 Alignment Quality

As explained in Section 5.2, alignment repair does not guarantee optimality. It is not straightforward to assess how well the repaired alignment scores in terms of optimality. To judge the *rank* of the repaired alignment, i.e., how many other alignments are closer to the optimal alignment, we need to traverse all possible alignments of a trace and a process tree. This is rather involved from a run-time complexity point and hence hard to incorporate within the experiments.

We propose a *grade* measure, that grades the repaired alignment, based on the relative distance of the alignment w.r.t. the optimal alignment. To compute the distance, we first compute the cost of the optimal alignment γ^*. Additionally, we create an alignment γ^w, consisting of only $(a, -)$-moves and $(-, v)$-moves, such that the log moves form the trace and the model moves

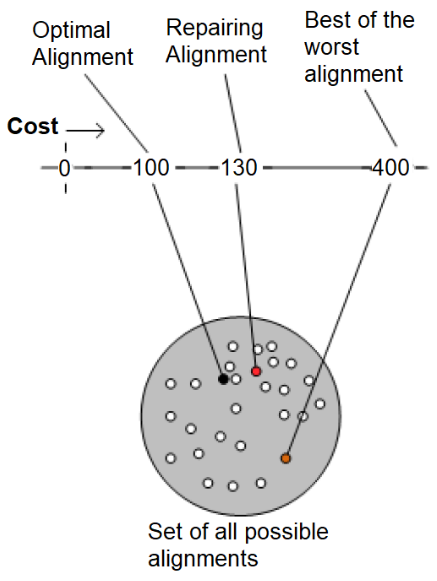


Fig. 15: Conceptual example of alignment grading.

form a shortest possible firing sequence of the process tree. Alignment γ^w represents *the best of the worst* alignments, i.e., a longer firing sequence is potentially possible though yields a worse alignment score. Finally, we calculate the cost of the repaired alignment γ^r. Based on the difference between the cost of γ^* and γ^w we compute the relative cost of γ^r. Let c^*, c^w and c^r denote the costs for γ^*, γ^w and γ^r. We grade the cost of γ^r as follows: $grade(\gamma^r) = 1 - \frac{c^r - c^*}{c^w - c^*}$. Clearly, $0 \leq grade(\gamma^r) \leq 1$. We used the following cost for move m: $z(m) = 5$ if m is a log move, $z(m) = 2$ if m is a model move and $z(m) = 0$ if m is synchronous. With these costs the movements in the model are more probable than the movements in the log, which is a reasonable assumption for alignments computation for models generated by process discovery algorithms. Consider Fig. 15 which schematically depicts the concept of alignment grading.

Fig. 16 shows box plots for the computed average grades of the repaired alignments. As the figure shows, we always have a grade above 0.84, and in the top 75% of all experiments is above 0.98. Thus, when the repaired alignments are not optimal, the difference with the optimal alignments is minimal. Hence, the loss of optimality is limited and stays within acceptable bounds.

Again, there is a close relation between the size the changed sub-tree and the potential loss of optimality. If the change is performed close to the root node, more log moves will belong to the scope of change. Consequently, the probability of retrieving an optimal alignment is higher. If the root of the point of change is the root node, we obviously do guarantee optimality.

6.4 Incorporation in the Evolutionary Tree Miner

In the previous sections we evaluated both runtime and the alignment quality. In this section the practical effects of the application of alignment repair are evaluated by running the Evolutionary Tree Miner (ETM) process discovery algorithm (Buijs, 2014). The ETM is applied on the real-life 2015 BPI Challenge (van Dongen, 2015) event log, which is filtered to contain those 30 activities that cover 50% of all events. This results in an event log with 1,199 cases and 26,208 events, implying that a trace contains 22 events on average.

Since the ETM can produce variable results, e.g. when it starts off with a particularly good or bad set of process trees, we ran the ETM 30 times. During each run the ETM created 200 generations of 20 process trees, of which 2 where kept in the elite, i.e. transferred between generations. This means that in each run of the ETM 3,602 process trees were generated and evaluated.

Analyzing the results show that the repaired alignment was calculated for 16.45%(±2.16%) of the process trees, i.e. one out of six process trees is repaired. Further analysis into the fraction of process trees repaired over the generations results in the graph of Fig. 17. The graph shows that the fraction of repaired trees per generation fluctuates (even after averaging over the 30 runs). The fluctuation is also partly caused by the population size of 20 trees per generation. The graph also clearly shows that in the first generations few trees are repaired. Overall there seems to be a slight trend towards a higher fraction of trees being repaired in later generations.

For the process trees where a repaired alignment was calculated, also a new optimal alignment was calculated for comparison. The results are shown in Table 2 where the average values of each run are averaged again. The results show that both the calculated cost and the resulting replay-fitness are not significantly different between the repaired and full alignment variants. The repaired alignments on average reports only a slightly worse replay-fitness compared to the a new optimal calculation. The average replay-fitness values are rather low, but this is typical for the behavior of the ETM in early runs. The complexity of alignment computation is measurable in the number of states, i.e., vertices in the marking-based reachability graph, it visits. When we consider the number of states visited by the alignment algorithm however, we see that the repaired version requires significantly less states (roughly a factor 2,000) to compute the final result.

These results confirm that the performance gains, as demonstrated by the significant drop in number of states required by the alignment algorithm, outweigh the decrease in accuracy, which is insignificant.

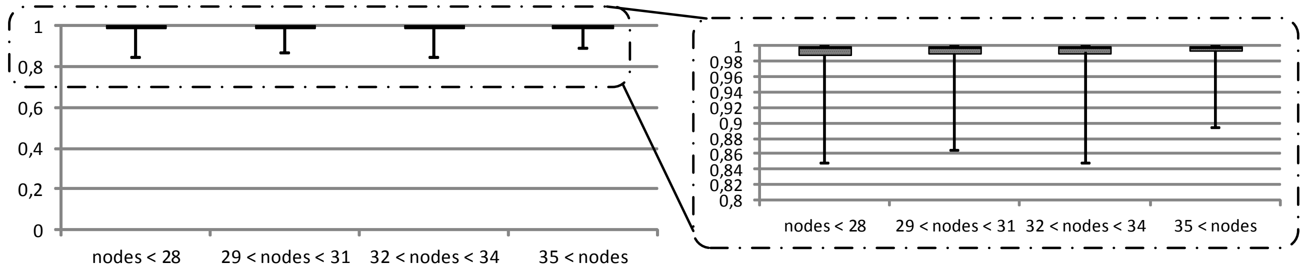


Fig. 16: Normalized *grade* of the repaired alignments.

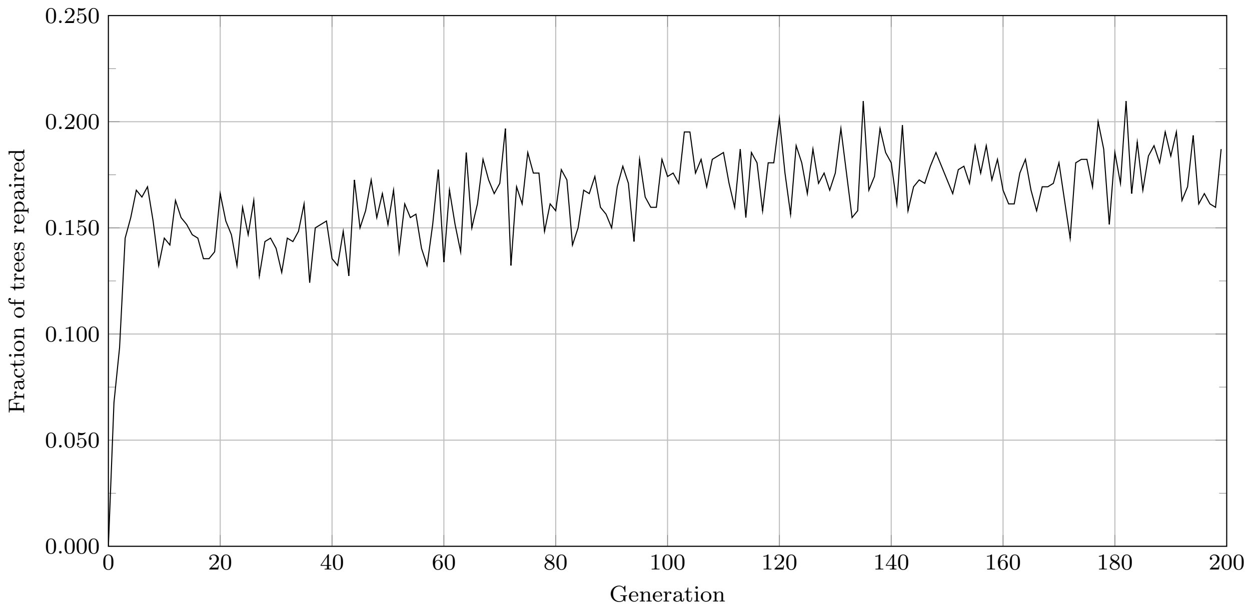


Fig. 17: Fraction of repaired trees per generation (averaged over 30 runs).

Table 2: Experimental results of ETM.

	Cost		*Replay-Fitness*		*States*	
	Repair	*Optimal*	*Repair*	*Optimal*	*Repair*	*Optimal*
Average	121,578.060	120,993.249	0.11938	0.12359	5.760	10,846.832
Std. Dev.	1,913.806	1,909.323	0.02084	0.02121	1.243	1,880.271

7 Conclusion

We presented a novel approach to compute alignments based on an existing alignment, instead of (re)computing the alignment from scratch. The approach needs a process model and an existing alignment in order to compute a new alignment for a similar process model. The technique extends and generalizes the technique presented in earlier work (Vázquez-Barreiros et al., 2016b).

We have shown that the technique guarantees to return sequences of moves which are in fact proper alignments. The evaluation shows that our approach always retrieves an alignment in a significantly lower, or worst-case equal, time than computing optimal alignments. Furthermore, we show that the potential loss of optimality is limited and stays within acceptable bounds. The approach has been validated with a set of random trees and event logs, resulting in more than 10^6 alignments. Furthermore, we show that the potential loss of optimality is limited and stays within acceptable bounds. Additionally we have integrated the approach within the Evolutionary Tree Miner (Buijs, 2014). Using the integration together with a real event log, we have shown the applicability of the approach in practice. Moreover the ETM-based experiments confirm that applying alignment repair reduces the complexity of computing alignments significantly.

Future Work The current approach only focuses on the changed sub-tree and not on its surroundings and/or the nature of the root of the changed sub-tree. Depending on the type of operators in the tree, it might be possible to extend or shrink the scope of change, allowing to reduce the loss of optimality. Hence, we plan to more explicitly the process model into account when computing the scope. Moreover, we plan to develop means to predict optimality, allowing us to decide at which point it is necessary to compute an optimal alignment instead of reusing an existing one.

The speedup obtained by using alignment repair is crucial for certain areas, e.g., stream-based process mining (Burattin et al., 2014, 2015; Hassani et al., 2015; van Zelst et al., 2018b, 2017), where it is necessary to keep the model up to date based on a real-time stream of events. New streams might lead to modifications of the discovered process model (concept drift (Ostovar et al., 2016)), resulting in new process models which are not so different from the previous model. This typically happens for gradual and incremental concept drifts that are related to changes in the structure of the process model. Reusing the previous alignments potentially allows us to update conformance checking statistics in significantly less time compared to recomputing all the optimal alignments. Therefore, we plan to assess challenges and the effectiveness of the presented technique in stream-based process mining.

Acknowledgements This research was supported by the Spanish Ministry of Economy and Competitiveness (grant TIN2014-56633-C3-1-R, co-funded by the European Regional Development Fund - FEDER program), the Galician Ministry of Education under the projects EM2014/012, CN2012/151, GRC2014/030, and the DELIBIDA research program supported by NWO.

The authors would like to thank Wil M.P. van der Aalst, Boudewijn F. van Dongen and XiXi Lu for their valuable feedback and suggestions.

References

van der Aalst, Wil M. P. 1998. The Application of Petri Nets to Workflow Management. Journal of Circuits, Systems, and Computers, 8(1):21–66.

van der Aalst, Wil M. P. 2012. Decomposing Process Mining Problems Using Passages. In Application and Theory of Petri Nets - 33rd International Conference, PETRI NETS 2012, Hamburg, Germany, June 25-29, 2012. Proceedings, pages 72–91.

van der Aalst, Wil M. P. 2013. Decomposing Petri Nets for Process Mining: A Generic Approach. Distributed and Parallel Databases, 31(4):471–507.

van der Aalst, Wil M. P. 2016. Process Mining - Data Science in Action, Second Edition. Springer.

van der Aalst, Wil M. P., Arya Adriansyah, & Boudewijn F. van Dongen 2012. Replaying History on Process Models for Conformance Checking and Performance Analysis. Wiley Interdisc. Rew.: Data Mining and Knowledge Discovery, 2(2):182–192.

Adriansyah, Arya 2014. Aligning Observed and Modeled Behavior. PhD thesis, Eindhoven University of Technology, Department of Mathematics and Computer Science.

Adriansyah, Arya, Boudewijn F. van Dongen, & Wil M. P. van der Aalst 2011. Conformance Checking Using Cost-Based Fitness Analysis. In Proceedings of the 2011 IEEE 15th International Enterprise Distributed Object Computing Conference, EDOC '11, pages 55–64, Washington, DC, USA. IEEE Computer Society.

Adriansyah, Arya, Boudewijn F. van Dongen, & Wil M. P. van der Aalst 2013. Memory-Efficient Alignment of Observed and Modeled Behavior. Technical report, BPM Center Report.

Adriansyah, Arya, Jorge Munoz-Gama, Josep Carmona, Boudewijn F. van Dongen, & Wil M. P. van der Aalst 2015. Measuring Precision of Modeled Behavior. Inf. Syst. E-Business Management, 13(1):37–67.

Alizadeh, Mahdi, Massimiliano de Leoni, & Nicola Zannone 2014. History-based Construction of Log-Process Alignments for Conformance Checking: Discovering What Really Went Wrong. In Accorsi, Rafael, Paolo Ceravolo, & Barbara Russo (eds), Proceedings of the 4th International Symposium on Data-driven Process Discovery and Analysis (SIMPDA 2014), Milan, Italy, November 19-21, 2014, volume 1293 of *CEUR Workshop Proceedings*, pages 1–15. CEUR-WS.org.

Buijs, Joos C. A. M. 2014. Flexible Evolutionary Algorithms for Mining Structured Process Models. PhD thesis, Eindhoven University of Technology.

Burattin, Andrea, Marta Cimitile, Fabrizio M. Maggi, & Alessandro Sperduti 2015. Online Discovery of Declarative Process Models from Event Streams. IEEE Trans. Services Computing, 8(6):833–846.

Burattin, Andrea, Alessandro Sperduti, & Wil M. P. van der Aalst 2014. Control-flow Discovery from Event Streams. In Proceedings of the IEEE Congress on Evolutionary Computation, CEC 2014, Beijing, China, July 6-11, 2014, pages 2420–2427. IEEE.

Carmona, Josep, Boudewijn F. van Dongen, Andreas Solti, & Matthias Weidlich 2018. Conformance Checking - Relating Processes and Models. Springer.

van Dongen, Boudewijn F. 2015. BPI Challenge 2015. 4TU.Centre for Research Data. Dataset.

van Dongen, Boudewijn F. 2018. Efficiently Computing Alignments - Using the Extended Marking Equation. In Business Process Management - 16th International Conference, BPM 2018, Sydney, NSW, Australia, September 9-14, 2018, Proceedings, pages 197–214.

Dumas, Marlon, Marcello La Rosa, Jan Mendling, & Hajo A. Reijers 2018. Fundamentals of Business Process Management, Second Edition. Springer.

van Eck, Maikel L., Joos C. A. M. Buijs, & Boudewijn F. van Dongen 2014. Genetic Process Mining: Alignment-Based Process Model Mutation. In Business Process Management Workshops - BPM 2014 International Workshops, Eindhoven, The Netherlands, September 7-8, 2014, Revised Papers, pages 291–303.

Fahland, Dirk, & Wil M. P. van der Aalst 2012. Repairing Process Models to Reflect Reality. In Business Process Management - 10th International Conference, BPM 2012, Tallinn, Estonia, September 3-6, 2012. Proceedings, pages 229–245.

Fahland, Dirk, & Wil M. P. van der Aalst 2015. Model Repair - Aligning Process Models to Reality. Inf. Syst., 47:220–243.

Hart, Peter E., Nils J. Nilsson, & Bertram Raphael 1968. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. IEEE Trans. Systems Science and Cybernetics, 4(2):100–107.

Hassani, Marwan, Sergio Siccha, Florian Richter, & Thomas Seidl 2015. Efficient Process Discovery From Event Streams Using Sequential Pattern Mining. In IEEE Symposium Series on Computational Intelligence, SSCI 2015, Cape Town, South Africa, December 7-10, 2015, pages 1366–1373. IEEE.

Leemans, Sander J. J., Dirk Fahland, & Wil M. P. van der Aalst 2013. Discovering Block-Structured Process Models from Event Logs - A Constructive Approach. In Colom, J. M., & J. Desel (eds), Application and Theory of Petri Nets and Concurrency - 34th International Conference, PETRI NETS 2013, Milan, Italy, June 24-28, 2013. Proceedings, volume 7927 of *Lecture Notes in Computer Science*, pages 311–329. Springer.

Leemans, Sander J. J., Dirk Fahland, & Wil M. P. van der Aalst 2014a. Exploring Processes and Deviations. In Business Process Management Workshops - BPM 2014 International Workshops, Eindhoven, The Netherlands, September 7-8, 2014, Revised Papers, pages 304–316.

Leemans, Sander J. J., Dirk Fahland, & Wil M. P. van der Aalst 2014b. Process and Deviation Exploration with Inductive Visual Miner. In Proceedings of the BPM Demo Sessions 2014 Co-located with the 12th International Conference on Business Process Management (BPM 2014), Eindhoven, The Netherlands, September 10, 2014., page 46.

de Leoni, Massimiliano, & Wil M. P. van der Aalst 2013. Aligning Event Logs and Process Models for Multi-perspective Conformance Checking: An Approach Based on Integer Linear Programming. In Business Process Management - 11th International Conference, BPM 2013, Beijing, China, August 26-30, 2013. Proceedings, pages 113–129.

de Leoni, Massimiliano, Fabrizio Maria Maggi, & Wil M. P. van der Aalst 2012. Aligning Event Logs and Declarative Process Models for Conformance Checking. In Business Process Management - 10th International Conference, BPM 2012, Tallinn, Estonia, September 3-6, 2012. Proceedings, pages 82–97.

de Leoni, Massimiliano, & Andrea Marrella 2017. Aligning Real Process Executions and Prescriptive Process Models through Automated Planning. Expert Syst. Appl., 82:162–183.

de Leoni, Massimiliano, Jorge Munoz-Gama, Josep Carmona, & Wil M. P. van der Aalst 2014. Decomposing Alignment-Based Conformance Checking of Data-Aware Process Models. In On the Move to Meaningful Internet Systems: OTM 2014 Conferences - Confederated International Conferences: CoopIS, and ODBASE 2014, Amantea, Italy, October 27-31, 2014, Proceedings, pages 3–20.

Munoz-Gama, Jorge, Josep Carmona, & Wil M. P. van der Aalst 2014. Single-Entry Single-Exit Decomposed Conformance Checking. Inf. Syst., 46:102–122.

Murata, Tadao 1989. Petri nets: Properties, Analysis and Applications. Proceedings of the IEEE, 77(4):541–580.

Ostovar, Alireza, Abderrahmane Maaradji, Marcello La Rosa, Arthur H. M. ter Hofstede, & Boudewijn F. van Dongen 2016. Detecting Drift from Event Streams of Unpredictable Business Processes. In Proceedings of the 35th International Conference on Conceptual Modeling ER'16, volume 9974 of *Lecture Notes in Computer Science*, pages 330–346. Springer.

Polyvyanyy, Artem, Wil M. P. van der Aalst, Arthur H. M. ter Hofstede, & Moe Thandar Wynn 2017. Impact-Driven Process Model Repair. ACM Trans. Softw. Eng. Methodol., 25(4):28:1–28:60.

Rozinat, Anne, & Wil M. P. van der Aalst 2008. Conformance Checking of Processes Based on Monitoring Real Behavior. Inf. Syst., 33(1):64–95.

Song, Wei, Xiaoxu Xia, Hans-Arno Jacobsen, Pengcheng Zhang, & Hao Hu 2017. Efficient Alignment Between Event Logs and Process Models. IEEE Trans. Services Computing, 10(1):136–149.

Vanhatalo, Jussi, Hagen Völzer, & Jana Koehler 2009. The Refined Process Structure Tree. Data Knowl. Eng., 68(9):793–818.

Vázquez-Barreiros, Borja, Manuel Mucientes, & Manuel Lama 2016a. Enhancing discovered processes with duplicate tasks. Inf. Sci., 373:369–387.

Vázquez-Barreiros, Borja, Sebastiaan J. van Zelst, Joos C. A. M. Buijs, Manuel Lama, & Manuel Mucientes 2016b. Repairing Alignments: Striking the Right Nerve. In Enterprise, Business-Process and Information Systems Modeling - 17th International Conference, BPMDS 2016, 21st International Conference, EMMSAD 2016, Held at CAiSE 2016, Ljubljana, Slovenia, June 13-14, 2016, Proceedings, pages 266–281.

van Zelst, Sebastiaan J., Alfredo Bolt, & Boudewijn F. van Dongen 2018a. Computing Alignments of Event Data and Process Models. T. Petri Nets and Other Models of Concurrency, 13:1–26.

van Zelst, Sebastiaan J., Alfredo Bolt, Marwan Hassani, Boudewijn F. van Dongen, & Wil M. P. van der Aalst 2017. Online Conformance Checking: Relating Event Streams to Process Models using Prefix-Alignments. International Journal of Data Science and Analytics.

van Zelst, Sebastiaan J., Boudewijn F. van Dongen, & Wil M. P. van der Aalst 2018b. Event Stream-Based Process Discovery Using Abstract Representations. Knowl. Inf. Syst., 54(2):407–435.