Repairing Alignments: Striking the Right Nerve

B. Vázquez-Barreiros[1], S.J. van Zelst[2], J.C.A.M. Buijs[2], M. Lama[1],
M. Mucientes[1]

[1] Centro de Investigación en Tecnoloxías da Informacíon (CiTIUS)
University of Santiago de Compostela, Spain
{borja.vazquez, manuel.lama, manuel.mucientes}@usc.es
[2] Department of Mathematics and Computer Science
Eindhoven University of Technology, The Netherlands
{s.j.v.zelst,j.c.a.m.buijs}@tue.nl

Abstract. Process Mining is concerned with the analysis, understanding and improvement of business processes. One of the most important branches 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 conformance statistics. Alignments map elements of an event log onto activities present in a business process model. Computing alignments is a combinatorial problem and hence, extremely costly. In this paper we show how to compute an alignment for a given process model, using an existing alignment and an existing process model as a basis. We show that we are able to effectively repair the existing alignment by updating those parts that no longer fit the given process model. Moreover, we show that the potential loss of optimality is limited and stays within acceptable bounds.

Keywords: Process Mining, Conformance Checking, Alignments

1 Introduction

Today's information systems store an overwhelming amount of data related to the execution of business processes. *Process mining* [1] is concerned with the analysis, understanding and improvement of business processes based upon such data in the form of *event logs*. Three main branches form the basis of process mining: *process discovery*, *conformance checking* and *process enhancement*. Within process discovery the main goal is to discover a business process model based on an event log. Within conformance checking the main goal is to check whether a given process model conforms to the behavior recorded in an event log. Within process enhancement the main goal is to improve business processes, primarily (though not exhaustively) using the two aforementioned fields.

Alignments [2] have proven to be very effective for the purpose of conformance checking. In essence, an alignment aligns an event log to a process model. Based on such alignment, a variety of analysis techniques can be applied resulting in different statistics describing the relation between the event log and the process model. A plethora of process mining techniques use alignments internally [3–7]. Replay-fitness and precision, two essential process mining quality dimensions [8], are computed on the basis of alignments [3, 4]. In evolutionary process discovery [5, 6], replay-fitness and precision are used to judge the quality of a newly generated process model. The Inductive Visual Miner [7] uses replay-fitness measures to visualize the flow of cases through a given process model.

The sheer complexity of computing alignments has its effects on the techniques that internally use them. Using alignments in combination with realistically sized event logs and process models, typically results in poor run time performance. However, some of the aforementioned techniques share an interesting common property, i.e. the potential use of *similar* process models. For example, within evolutionary process discovery a new generation of process models is created based on slight manipulations of the current generation of process models [9]. The Inductive Visual Miner allows the user to apply filtering techniques, resulting in a new, rather similar, process model for which we need to recompute alignments. Hence, the question arises whether we can use previously computed alignments as a basis for computing of new alignments.

In this paper we propose an *alignment repair method* that, given a process model and an existing alignment on a different process model, computes a new alignment for the given process model. The technique identifies fragments of the existing alignment that do not correspond to the given 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, i.e. the method strikes the right nerve, computation time decreases significantly. Moreover, we show that the loss of optimality is limited and stays within acceptable bounds.

The remainder of this paper is structured as follows. Section 2 explains event logs, process trees and alignments. Section 3 describes the alignment repair method in detail. In Section 4 we present an evaluation of the approach. Section 5 discusses related work and Section 6 concludes the paper.

2 Preliminaries

In this section we introduce the notion of event logs, process trees and alignments.

2.1 Sequences, Bags and Event Logs

We write a bag as $[e_1^{n_1}, e_2^{n_2}, ..., e_k^{n_k}]$ where element e_1 occurs n_1 times, with $n_1 > 0$. As an example, $B_1 = [a^3, b^5]$ denotes a bag consisting of 3 a's, 5 b's and 0 c's. Sequences are written as $\langle e_1, e_2, ..., e_n \rangle$. Sequence concatenation of sequences σ_1 and σ_2 is written as $\sigma_1 \cdot \sigma_2$. As an example consider the concatenation of sequences $\langle a, b \rangle$ and $\langle c, d \rangle$: $\langle a, b \rangle \cdot \langle c, d \rangle = \langle a, b, c, d \rangle$. The set of all possible sequences over some set of elements X is denoted as X^*, e.g. $\langle a, b \rangle \in \{a, b, c\}^*$. Given a set X and an element $e \notin X$, we write X^e as a shorthand for $X \cup \{e\}$.

Table 1: A fragment of an event log loosely based on a fictional loan application process [10], where each individual line corresponds to an event.

Case-id	Activity	Resource	Time-stamp
⋮	⋮	⋮	⋮
3554	*Check application form*	*John*	*2015-10-08T09:45:37+00:00*
3555	*Check application form*	*Lucy*	*2015-10-08T10:12:37+00:00*
3554	*Check credit history*	*Harold*	*2015-10-08T10:14:25+00:00*
3555	*Check credit history*	*Harold*	*2015-10-08T10:31:02+00:00*
3554	*Appraise property*	*Pete*	*2015-10-08T10:45:22+00:00*
3554	*Assess loan risk*	*Harold*	*2015-10-08T10:49:52+00:00*
3555	*Assess loan risk*	*Harold*	*2015-10-08T11:01:51+00:00*
3553	*Return application to client*	*John*	*2015-10-08T11:03:18+00:00*
3556	*Check application form*	*Lucy*	*2015-10-08T11:05:10+00:00*
3555	*Assess eligibility*	*Harry*	*2015-10-08T11:06:22+00:00*
3554	*Assess eligibility*	*Harry*	*2015-10-08T11:33:42+00:00*
3554	*Reject application*	*Harry*	*2015-10-08T11:45:42+00:00*
3557	*Check application form*	*Lucy*	*2015-10-08T13:48:12+00:00*
3555	*Prepare acceptance pack*	*Sue*	*2015-10-08T14:02:22+00:00*
⋮	⋮	⋮	⋮

Event logs often act as a primary input for process mining techniques and describe the actual execution of activities within a business process. In essence, an event log is a bag of sequences that consist of business process events. Consider Table 1 depicting a snapshot of an event log of a fictional loan application handling process. Let us consider all activities related to the case *3554*. First, John *Checks the application form*, after which Harold *checks the applicant's credit history*. Pete *appraises the property* after which Harold performs a *loan risk assessment*. Finally, Harry *assesses the eligibility of the client for the loan* and in the end he decides to *reject the application*. A sequence of events, e.g. the execution of the activities related to case *3554*, is referred to as a *trace*. Thus, from the *control-flow perspective*, i.e. the sequential ordering of *activities* w.r.t. *cases*, case *3554* can be written as ⟨*Check application form*, *Check credit history*, *Appraise property*, *Assess loan risk*, *Assess eligibility*, *Reject application*⟩.

2.2 Process Trees

A process tree [5, 11] is an abstract hierarchical representation of a block-structured workflow net [12]. The leafs of a process tree are labeled with *activities*. The internal nodes are labeled with *operators*, used to specify the relation between their children. Formally, every node within a process tree describes a language, i.e., a set of sequences of activities. The language of the process tree itself is equal to the language of the root node of the process tree. Thus, the labels of the leafs of the process tree form the *alphabet* of a process tree's language. The operators describe how the languages of their children have to be combined.

There are five standard operator types [5] 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 can have an arbitrary number of children in any arbitrary order, except for the sequence and loop operators. For the sequence operator ($\rightarrow$), the number of children can be arbitrary, though the order of the children specifies the order in which they must be evaluated, i.e. from left to right. Loop nodes ($\circlearrowleft$) always have three children: the left child is the *do* part of the loop, the middle child is the *redo* part, and the right child is the *exit* part. We refer to [5, 11] for an exact, formal, language definition of process trees.

Figure 1 shows an example process tree P_1 with all five possible operators. The root node n_1 is labeled with a sequence operator ($\rightarrow$), hence we first evaluate its left-most child, n_2, which is a leaf labeled with activity a. Thereby, every sequence present in the language of P_1 starts with an a-activity. The second child of the root, n_3, is a sub-tree describing the parallel execution ($\wedge$) of activity b, with a non-exclusive choice ($\vee$) between activities c and d. The third child again refers to a single activity, labeled e. Finally a loop ($\circlearrowleft$) will be executed. The *do* part of the loop consists of an exclusive choice ($\times$) between f and g. If we decide to *re-do* the loop, we execute activity h. Executing activity h enforces us to re-execute the exclusive choice between f and g. The *exit* part of the loop is labeled with activity τ. This activity is unobservable, i.e., it is not part of any sequence in the language of P_1.

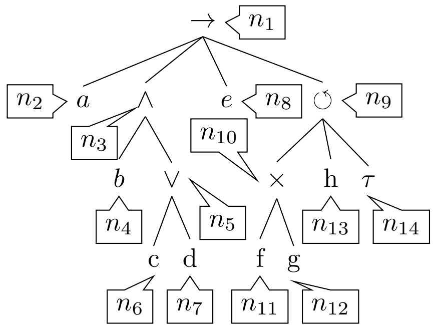


Fig. 1: Process Tree P_1.

Instead of recording the activity labels directly, we first record the sequence of leaf-nodes described by the process tree. As a second step, this sequence is projected on the activities associated with the leaf nodes. As an example consider the sequence of leaf nodes $\langle n_2, n_4, n_6, n_8, n_{12}, n_{14}\rangle$. Projected on the activity labels yields $\langle a, b, c, e, g, \tau\rangle$. The final label τ is, an unobservable label and hence the sequence becomes $\langle a, b, c, e, g\rangle$. Due to the loop operator n_9, the language of P_1 is infinite. Some other example sequences present in P_1's language are: $\langle n_2, n_7, n_4, n_8, n_{11}, n_{14}\rangle \equiv \langle a, d, b, e, f\rangle$, $\langle n_2, n_6, n_4, n_7, n_8, n_{11}, n_{13}, n_{12}, n_{14}\rangle \equiv \langle a, c, b, d, e, f, h, g\rangle$, etc.

2.3 Alignments

Alignments map events present in a trace to activities in a process model, i.e., the leaf nodes of a process tree. Alignments allow us to decide to what degree a given trace fits the language of a process tree. Given a trace σ_L and a sequence of leaf nodes σ_P, an alignment is a partial injective function mapping the elements of σ_L to the elements of σ_P. If the function is total and every non-mapped element in the range has a τ label, the trace σ_L *perfectly aligns* σ_P.

Throughout the paper we will use A to represent the activities of a trace and N to represent the set of leafs of some process tree P. Additionally, we use $\gg$ (i.e. $\gg \notin A$, $\gg \notin N$) to represent the *skip move*. The skip move represents an element from the domain (range) that is not part of the alignment, i.e. not part of the partial injective function. We represent an alignment as a sequence of pairs, combining the events in a trace with leaf nodes of a process tree, i.e., an alignment is represented as an element of $(A^{\gg} \times N^{\gg})^*$. A pair $(a, n) \in (A^{\gg} \times N^{\gg})$ is referred to as a *move*. The premise of an alignment is that: *i)* the elements of the $A^{\gg}$-part respect the ordering of the events within the trace; and *ii)* the elements of the $N^{\gg}$-part, projected onto their activity labels, form an element of the language of the process tree. We distinguish the following moves (a, n): *i)* a synchronous move, if $a \in A$ and $n \in N$ s.t. n's label equals a or $a = \gg$ and $n \in N$ s.t. n's label is τ; *ii)* a model move, if $a = \gg$ and $n \in N$; and *iii)* a log move, if $a \in A$ and $n = \gg$. Other combinations are considered *illegal*. Given a trace σ and a process tree P, we write $\gamma_{(\sigma,P)}$ to denote an alignment of σ and P. We refer to [2] for a formal, Petri-net based definition of alignments.

Consider the trace $\sigma_1 = \langle a, b, c, e, f \rangle$ and the leaf sequence $\langle n_2, n_4, n_6, n_8, n_{11}, n_{14} \rangle$ of process tree P_1 of Figure 1. Clearly $\langle n_2, n_4, n_6, n_8, n_{11}, n_{14} \rangle \equiv \langle a, b, c, e, f \rangle$ and thus if we (trivially) map $\sigma_1(1)$ *onto* n_2, $\sigma_1(2)$ *onto* n_4, *...*, $\sigma_1(5)$ *onto* n_{11} we find a perfect alignment of σ_1 on the process tree.[1]

$$\gamma^1_{(\sigma_1,P_1)} = \frac{A^{\gg}}{N^{\gg}} \begin{array}{||c|c|c|c|c|c|} a & b & c & e & f & \gg \\ \hline n_2 & n_4 & n_6 & n_8 & n_{11} & n_{14} \end{array}$$

Alignment $\gamma^1_{(\sigma_1,P_1)}$ is not the only possible alignment between σ_1 and P_1. It is also possible to map σ_1 to the leaf sequence $\langle n_2, n_4, n_6, n_8, n_{11}, n_{13}, n_{12}, n_{14} \rangle$:

$$\gamma^2_{(\sigma_1,P_1)} = \frac{A^{\gg}}{N^{\gg}} \begin{array}{||c|c|c|c|c|c|c|c|} a & b & c & e & f & \gg & \gg & \gg \\ \hline n_2 & n_4 & n_6 & n_8 & n_{11} & n_{13} & n_{12} & n_{14} \end{array}$$

However, we favor $\gamma^1_{(\sigma_1,P_1)}$ over $\gamma^2_{(\sigma_1,P_1)}$ as it contains less $(\gg, n)$-typed moves.

In the previous example, the trace is an element of the language of P_1. If we consider the trace $\sigma_2 = \langle a, b, c, d, e, f, g \rangle$, this is not the case. Activity f and g can never co-occur in any sequence present in the language of P_1, unless activity h is in between them (due to the loop operator). For σ_2, we are able to construct (amongst others) these alignments:

$$\gamma^1_{(\sigma_2,P_1)} = \frac{A^{\gg}}{N^{\gg}} \begin{array}{||c|c|c|c|c|c|c|c|} a & b & c & d & e & f & g & \gg \\ \hline n_2 & n_4 & n_6 & n_7 & n_8 & n_{11} & \gg & n_{14} \end{array}$$

$$\gamma^2_{(\sigma_2,P_1)} = \frac{A^{\gg}}{N^{\gg}} \begin{array}{||c|c|c|c|c|c|c|c|} a & b & c & d & e & f & g & \gg \\ \hline n_2 & n_4 & n_6 & n_7 & n_8 & \gg & n_{12} & n_{14} \end{array}$$

For alignments $\gamma^1_{(\sigma_2,P_1)}$ and $\gamma^2_{(\sigma_2,P_1)}$, it is less obvious which one is favored over the other one, or, if both alignments are equally favorable. In general, given a trace and a process model, a multitude of alignments exist. We are however interested in an *optimal* alignment. In essence, an optimal alignment is an alignment that minimizes some cost function $\kappa : (A^{\gg} \times N^{\gg})^* \to \mathbb{R}^+$. For a given alignment γ, $\kappa(\gamma)$ often is computed deterministically as each type of move gets a cost assigned. A synchronous move typically has cost 0, whereas any illegal move has cost ∞. Costs of model/log moves are usually problem specific though usually greater

[1] Note that n_{14} is a leaf with a τ label and maps to synchronous move $(\gg, n_{14})$.

than 0. Optimal alignments are computed for ordinary Petri nets with an initial marking and a (collection of) final marking(s), e.g. by using the A^* algorithm [2]. Trivially this applies to workflow nets and, as a consequence, process trees as well. Hence, for the purpose of this paper, we assume the availability of an *oracle* function o that, given a trace σ and a model P, produces an (optimal) alignment.

3 Repairing Alignments

As indicated, some process mining techniques share an interesting property, i.e., alignments need to be computed for multiple relatively *similar* process models. Henceforth, the main research question addressed in this paper is formulated as follows. Given a trace σ_L, a process tree P, a process tree P', and an alignment $\gamma_{(\sigma_L,P)}$, are we able to compute an alignment $\gamma_{(\sigma_L,P')}$ by reusing and repairing $\gamma_{(\sigma_L,P)}$?

3.1 Repairing Alignments: A Concrete Example

We illustrate alignment repair by providing an algorithmic sketch based on a running example. We use process trees P_1 and P_2 (Figure 2) as a running example. Consider trace $\sigma_3 = \langle a, c, d, e, f, g, h, g, f \rangle$ and alignment $\gamma_{(\sigma_3,P_1)}$ of σ_3 on P_1:

$$\gamma_{(\sigma_3,P_1)} = \frac{A^{\gg}}{N^{\gg}} \begin{array}{||c|c|c|c|c|c|c|c|c|c|c|} a & \gg & c & d & e & f & g & h & g & f & \gg \\ \hline n_2 & n_4 & n_6 & n_7 & n_8 & n_{11} & \gg & n_{13} & n_{12} & \gg & n_{14} \end{array}$$

Trace σ_3 is missing a b activity enforced by n_3 ($\wedge$). Moreover, $n_{10}(\times)$, does not allow for executing both the f- and g-activity without an h-activity in between. We are interested in changing the operator type of n_{10} as it yields two moves that include a $\gg$ symbol: $(g, \gg)$ and $(f, \gg)$. This is fixed by changing the operator type of n_{10} to either $\wedge$ or $\vee$. Consider process tree P_2 depicted on the right-hand side of Figure 2, in which the operator type of n_{10} is changed to $\wedge$.

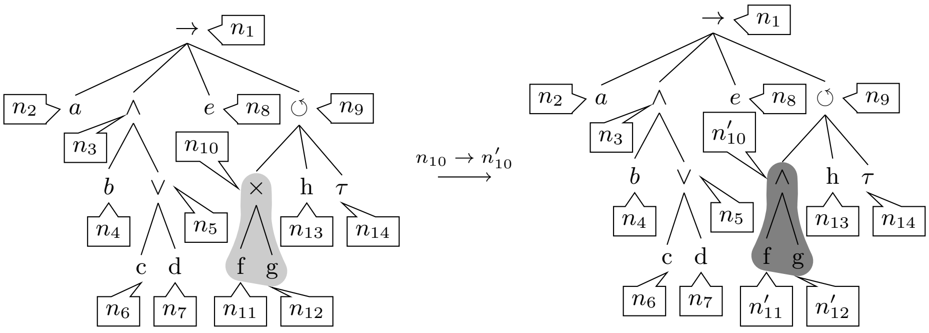


Fig. 2: Modification of node n_{10} in process tree P_1 (left), resulting in process tree P_2 (right). Scope of change S_1 is highlighted in P_1 (light gray), scope S_2 is highlighted in P_2 (dark gray).

For convenience, n_{10} and its children n_{11} and n_{12} are relabeled to n'_{10}, n'_{11} and n'_{12}, respectively. The process to compute $\gamma_{(\sigma_3,P_2)}$ by reusing $\gamma_{(\sigma_3,P_1)}$ consists of three steps: *i)* scope of change detection, *ii)* sub-alignment computation, and *iii)* alignment reassembly.

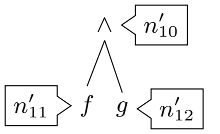
Step 1; Scope of Change Detection The first step in reusing $\gamma_{(\sigma_3,P_1)}$ involves detecting the scope of change of P_1 w.r.t P_2 and vise versa. The scope of change itself is a process tree and is defined by the modified node and its children. For process trees P_1 and P_2 we highlighted scopes of change S_1 and S_2 in light and dark gray in Figure 2. S_1 consists of nodes n_{10}, n_{11} and n_{12} of P_1. S_2 consists of nodes n'_{10}, n'_{11} and n'_{12} of P_2. Using this information, we need to detect what elements of $\gamma_{(\sigma_3,P_1)}$ belong to the scope of change of P_1, i.e. to leaf nodes of S_1. In this step, we linearly walk through all moves of $\gamma_{(\sigma_3,P_1)}$ checking for each $(a, n) \in \gamma_{(\sigma_3,P_1)}$, whether or not it belongs to scope S_1. We take the following (a, n) moves in consideration for scope S_1:

1. If n is a leaf of scope S_1, (a, n) belongs to S_1.
2. If $n =\gg$, and the previous move $(a', n') \in \gamma_{(\sigma_3,P_1)}$ belongs to S_1, then also (a, n) belongs to S_1.

Using the aforementioned rules, we start constructing the new alignment $\gamma_{(\sigma_3,P_2)}$. Every pair $(a, n) \in \gamma_{(\sigma_3,P_1)}$ *not belonging to* S_1 remains untouched and is copied in the exact same position into the new alignment $\gamma_{(\sigma_3,P_2)}$. On the other hand, we skip every move (a, n) *belonging to* S_1, yet for each of such move we remember the exact position in $\gamma_{(\sigma_3,P_1)}$. Moreover, whenever we encounter the first move (a, n) inside scope S_1, we create an intermediary sequence $\sigma_3^1 = \langle a\rangle$ (or $\sigma_3^1 = \epsilon$ if $a =\gg$). For every subsequent move (a', n') belonging to scope S_1 we update σ_3^1 to $\sigma_3^1 \cdot \langle a'\rangle$ (or $\sigma_3^1 \cdot \epsilon = \sigma_3^1$ if $a' =\gg$). However, if during this process we detect a move (a'', n'') belonging to S_1, which indicates a *new execution* of the process tree described by S_1, we create a new trace $\sigma_3^2 = \langle a''\rangle$ (or $\sigma_3^2 = \epsilon$ if $a'' =\gg$). Note that this type of behavior might be present in the alignment due to loop structures in P_1. Let us consider Table 2. The first five elements of $\gamma_{(\sigma_3,P_1)}$ are outside of scope S_1, hence they will be copied directly into $\gamma_{(\sigma_3,P_2)}$. The sixth and seventh element, i.e. (f, n_{11}) and $(g, \gg)$, belong to scope S_1 and thus we remember their positions and create σ_3^1 based on them. The eight element is again outside of scope S_1 and will be directly copied into $\gamma_{(\sigma_3,P_2)}$. The ninth

Table 2: Schematic overview of the first step of the alignment repair algorithm.

$A^{\gg}$	a	$\gg$	c	d	e	f	g	h	g	f	$\gg$
$N^{\gg}$	n_2	n_4	n_6	n_7	n_8	n_{11}	$\gg$	n_{13}	n_{12}	$\gg$	n_{14}
	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\underbrace{\qquad}$		$\downarrow$	$\underbrace{\qquad}$		$\downarrow$
$A^{\gg}$	a	$\gg$	c	d	e	$\sigma_3^1 = \langle f, g\rangle$		h	$\sigma_3^2 = \langle g, f\rangle$		$\gg$
$N^{\gg}$	n_2	n_4	n_6	n_7	n_8			n_{13}			n_{14}



$$\gamma_{(\sigma_3^1,S_2)} = \frac{A^{\gg}}{N^{\gg}} \begin{array}{|c|c|} f & g \\ \hline n'_{11} & n'_{12} \end{array} \qquad \gamma_{(\sigma_3^2,S_2)} = \frac{A^{\gg}}{N^{\gg}} \begin{array}{|c|c|} g & f \\ \hline n'_{12} & n'_{11} \end{array}$$

Fig. 3: Process tree S_2 and the two alignments $\gamma_{(\sigma_3^1,S_2)}$ and $\gamma_{(\sigma_3^2,S_2)}$.

and tenth element, i.e. (g, n_{12}) and $(f, \gg)$, again belong to scope S_1. These two moves indicate a new execution of the process tree described by S_1 and hence, we create a new sequence σ_3^2 out of the two elements. Finally the last element of the alignment is again outside of scope S_1.

Step 2; Alignment Calculation We now constructed a part of the new alignment $\gamma_{(\sigma_3,P_2)}$ together with a set of sequences, i.e. the lower part of Table 2. For each of these sequences, we additionally have a set of pointers referring to the elements of $\gamma_{(\sigma_3,P_1)}$ that generated the sequence. The next step of the repair consists of creating new chunks of alignments for the sub-sequences generated from the elements belonging to S_1. The core idea is that sequences σ_3^1 and σ_3^2 are both referring to behavior related to S_1. However, in P_2, S_2 is the replacement of S_1. Thus in the new alignment, this behavior can no longer be present and needs to be updated in context of S_2. As S_2 itself defines a process tree we use the *oracle* function o to compute two new alignments $\gamma_{(\sigma_3^1,S_2)}$ and $\gamma_{(\sigma_3^1,S_2)}$. The result of computing these alignments, together with S_2, are depicted in Figure 3.

Step 3; Alignment Reassembly The final step of the approach concerns placing back the newly created alignment fragments into the partially finished alignment, i.e. the bottom part of Table 2. Recall that we stored the position of every move in $\gamma_{(\sigma_3,P_1)}$ that belonged to S_1. For each such move (a, n) inside the scope in $\gamma_{(\sigma_3,P_1)}$ with $a \neq \gg$, we know that there is a move (a, n') in either $\gamma_{(\sigma_3^1,S_2)}$ or $\gamma_{(\sigma_3^2,S_2)}$. In our example consider (f, n_{11}) vs. (f, n'_{11}) in $\gamma_{(\sigma_3^1,S_2)}$, $(g, \gg)$ vs. (g, n'_{12}) in $\gamma_{(\sigma_3^1,S_2)}$, (g, n_{12}) vs. (g, n'_{12}) in $\gamma_{(\sigma_3^2,S_2)}$ and $(f, \gg)$ vs. (f, n'_{11}) in $\gamma_{(\sigma_3^2,S_2)}$. These type of moves serve as *anchor points* for placing the new alignment fragments into the partially finished alignment. If an alignment fragment does not contain any anchor point, i.e. caused by replay on an empty sequence, we are still able to place the new alignment back due to the fact that we have a one-to-one mapping between the old alignment moves and the (empty) sequence used in step 2. Once the third step is performed, we obtain an alignment of σ_3 on P_2, constructed by only calculating alignments on S_1 rather than P_1 as a whole. The final step resulting in $\gamma_{(\sigma_3,P_2)}$ is depicted in Table 3.

In the example, intermediary sequences σ_3^1 and σ_3^2 directly correspond to a consecutive block of elements of $\gamma_{(\sigma_3,P_1)}$. Due to parallelism, either by nodes labeled with an $\wedge$ or an $\vee$ operator, this is not necessarily the case, i.e. the subsequences can be related to a set of alignment moves scattered around the original alignment. Note that due to the use of the *anchor moves*, we are still able to put the elements of the newly created alignments into a correct position.

Table 3: Schematic overview of the third step of the alignment repair algorithm.

$A^{\gg}$	a	$\gg$	c	d	e	$\sigma_3^1 = \langle f, g \rangle$		h	$\sigma_3^2 = \langle g, f \rangle$		$\gg$
$N^{\gg}$	n_2	n_4	n_6	n_7	n_8			n_{13}			n_{14}
	↓	↓	↓	↓	↓	↓		↓	↓		↓
$A^{\gg}$	a	$\gg$	c	d	e	f	g	h	g	f	$\gg$
$N^{\gg}$	n_2	n_4	n_6	n_7	n_8	n'_{11}	n'_{12}	n_{13}	n'_{12}	n'_{11}	n_{14}

3.2 Optimality of Repaired Alignments

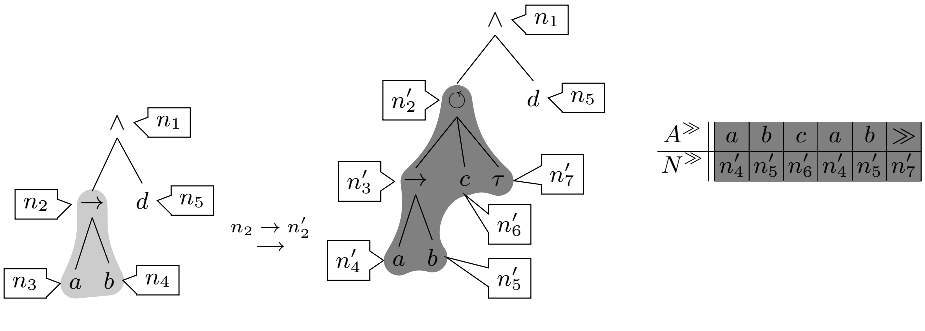
In the previous example, the repaired alignment is optimal. In general, we are not able to guarantee that the resulting repaired alignment is optimal. Consider the change between process trees P_3 and P_4 depicted in Figure 4, trace $\sigma_4 = \langle a, b, c, d, a, b, c, a, b \rangle$ and optimal alignment $\gamma_{(\sigma_4,P_3)}$:

$$\gamma_{(\sigma_4,P_3)} = \begin{array}{c||c|c|c|c|c|c|c|c|c} A^{\gg} & a & b & c & d & a & b & c & a & b \\ \hline N^{\gg} & \gg & \gg & \gg & n_5 & n_3 & n_4 & \gg & \gg & \gg \end{array}$$

Scope S_3 is defined by the sub-tree of P_3 starting from node n_2 whereas scope S_4 is the subtree of P_4 starting at node n'_2. Note that if we apply the algorithm as described in Section 3.1, the light gray colored moves belong to scope S_3. The algorithm will subsequently create the intermediary sequence $\sigma_4^1 = \langle a, b, c, a, b \rangle$ and let *oracle* function o compute the alignment $\gamma_{(\sigma_4^1,S_4)}$ as depicted in Figure 4. Eventually, $\gamma_{(\sigma_4^1,S_4)}$ is combined with the first four moves of $\gamma_{(\sigma_4,P_3)}$, yielding $\gamma_{(\sigma_4,P_4)}$:

$$\gamma_{(\sigma_4,P_4)} = \begin{array}{c||c|c|c|c|c|c|c|c|c|c} A^{\gg} & a & b & c & d & a & b & c & a & b & \gg \\ \hline N^{\gg} & \gg & \gg & \gg & n_5 & n'_4 & n'_5 & n'_6 & n'_4 & n'_5 & n'_7 \end{array}$$

Clearly $\gamma_{(\sigma_4,P_4)}$ is not optimal, i.e. consider $\gamma^*_{(\sigma_4,P_4)}$ on Page 10, which in fact is an optimal alignment of σ_4 on P_4.



$A^{\gg}$	a	b	c	a	b	$\gg$
$N^{\gg}$	n'_4	n'_5	n'_6	n'_4	n'_5	n'_7

Fig. 4: Process Trees P_3 (left), P_4 (right), scope of change S_3 (light gray), scope of change S_4 (dark gray) and alignment $\gamma_{(\sigma_4^1,S_4)}$.

$$\gamma^{*}_{(\sigma_4,P_4)} = \frac{A^{\gg}}{N^{\gg}} \begin{array}{||c|c|c|c|c|c|c|c|c|c|} a & b & c & d & a & b & c & a & b & \gg \\ \hline n'_4 & n'_5 & n'_6 & n_5 & n'_4 & n'_5 & n'_6 & n'_4 & n'_5 & n'_7 \end{array}$$

Unfortunately, we are not able to assign the three log moves at the start of alignment $\gamma_{(\sigma_4,P_3)}$, i.e. $(a,\gg)$, $(b,\gg)$ and $(c,\gg)$ to scope S_3. Hence these moves remain untouched whereas they in fact should be mapped onto elements of the leafs of S_4. Therefore, this leads to a repaired alignment that is not optimal. Nevertheless, in Section 4 we show that the potential loss of optimality is limited and stays within acceptable bounds.

3.3 Feasibility of Repaired Alignments

One of the basic requirements of the presented approach is that, after reusing an existing (optimal) alignment, the repaired alignment itself is an alignment. Due to the rather informal nature of this paper, we provide an intuition on the fact that the repaired alignment is indeed an alignment, rather than a formal proof. Recall that the premise of an alignment is that: *i)* the elements of the $A^{\gg}$-part respect the ordering of the events within the trace and *ii)* the elements of the $N^{\gg}$-part, projected onto their activity labels, form an element of the language of the process tree. Let P and P' denote two process trees and let S and S' denote the scopes of change of P and P' respectively. Moreover let $\sigma \in A^*$ be a trace and let $\gamma_{(\sigma,P)}$ denote an (optimal) alignment of σ on P. Let $\gamma_{(\sigma,P')}$ denote the sequence of $(A^{\gg} \times N'^{\gg})$ moves resulting from a repair of $\gamma_{(\sigma,P)}$ based on P'.

Observe that by using moves (a,n) with $a \neq \gg$ as *anchor points*, we effectively keep all elements of the $A^{\gg}$-part in place w.r.t. each other. The *oracle* function o by definition respects the order of the elements of the $A^{\gg}$-part w.r.t. the generated intermediary subsequences. Thus we enforce that the elements of the $A^{\gg}$-part of $\gamma_{(\sigma,P')}$ respect the ordering of trace σ. Hence $\gamma_{(\sigma,P')}$ fulfills part *i)* of the premise.

The intuition of part *ii)* of the premise is a bit more involved. Let us consider the case in which there is no $\wedge$ or $\vee$ operator on the path from the root of P to the root of S (the root of S may be a $\wedge$ or $\vee$ operator). In this case any valid execution of S always results in a consecutive block of leaf nodes of S in $\gamma_{(\sigma,P)}$. For each of these consecutive blocks we know that at that point in time sub-tree S must have been active. In step two, for each of these consecutive blocks we create a new alignment fragment based on S'. The *oracle* function o guarantees that *ii)* holds for these fragments. We then place the newly created chunks, corresponding to behavior of S', exactly at the points where S was active. If we assume that in this case *ii)* does not hold, this contradicts either the *oracle* function or the fact that $\gamma_{(\sigma,P)}$ was an alignment in the first place, hence *ii)* must hold. In case there is an $\wedge$ or $\vee$ operator on the path from the root of P to the root of S, we know that there might be interference of other parts of the tree w.r.t. S. The intermediate sequences can be potentially build up out of multiple chunks of alignment moves scattered around $\gamma_{(\sigma,P)}$. In this case, S was active throughout the whole span of the first chunk mapping to an intermediate subsequence up until the last chunk mapping to the same intermediate subsequence. Moreover, we know that within the span of S, we can reorder any leaf node of S with any other leaf node not in S, as long as we do not reorder any two leafs of S. The *oracle* function o provides us with the guarantee that *ii)* holds for the new alignment fragments based on S'. Since we use the *anchor points* we know that we might only reposition leafs of S' w.r.t. leafs of P' that are not an element of S'. Leaf nodes of S' are however never shuffled. We know that at any position where we place the new (chunks of) fragments based on S' back, S' has to be active. Thus, also in this case, if we assume that in this *ii)* does not hold, this contradicts either the *oracle* function or the fact that $\gamma_{(\sigma,P)}$ was an alignment.

4 Evaluation

To validate the usefulness of the presented technique, we answer two main questions: *i)* What is the time needed to align a model and a log with the presented technique? and *ii)* How close/far is the repaired alignment from the optimal alignment? In this section we answer these questions by *i)* comparing the time needed for alignment repair with the time expended to compute a new, optimal alignment and *ii)* by measuring the quality of the repaired alignments w.r.t. the new, optimal alignment.

4.1 Experimental Set-Up

Figure 5 shows a schematic overview of the experimental-setup. We generate an initial *random* process tree with a 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 Figure 5), 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: the optimal alignments based on the simulated log (step b in Figure 5) and the repaired alignments reusing the optimal alignments previously calculated on the base model (step c in Figure 5). Finally, we compare both outputs (step d in Figure 5).

Following this process, we created a set of 50 initial *random* trees with arbitrary sizes between 21 and 47 nodes. Thus, we applied the presented technique over

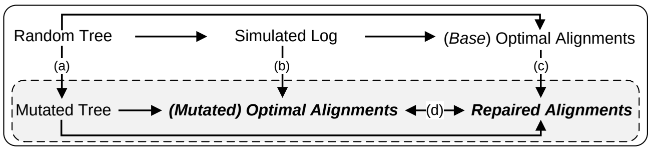


Fig. 5: Process followed during the experimentation.

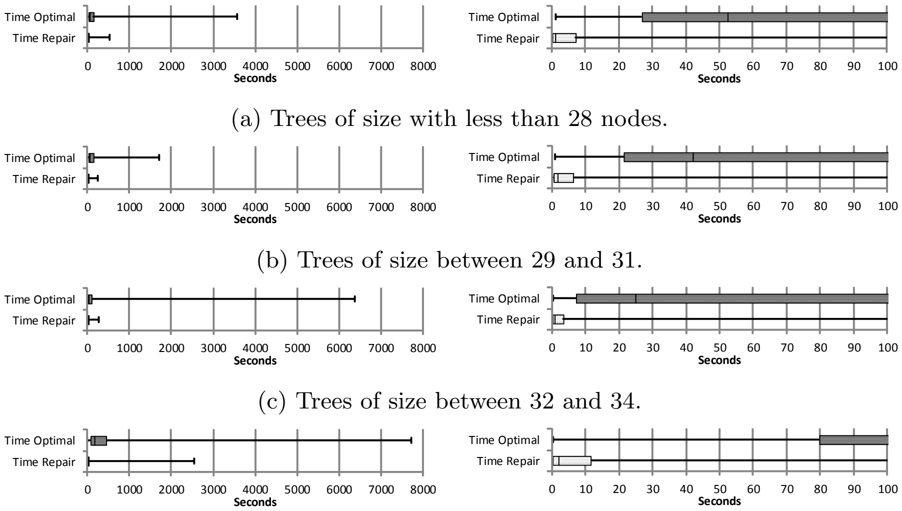
$50 \times 150 \times 2000 \approx 1.5 \cdot 10^6$ alignments. We additionally checked whether the repaired alignment is indeed an alignment, which *was true in all cases.*[1]

4.2 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 nodes (12 trees in total), a bucket with sizes between 29 and 31 nodes (12 trees in total), a bucket with sizes between 32 and 34 nodes (13 trees in total) and a bucket with sizes greater than 35 nodes (13 trees in total).

Figure 6 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 Figure 6 we also show the box plots zoomed into the domain 0-100 seconds.

In general we observe 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. In this case, the time needed for alignment repair is directly related to the size of the scope of change which explains the



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

(b) Trees of size between 29 and 31.

(c) Trees of size between 32 and 34.

(d) Trees with more than 35 nodes.

Fig. 6: 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.

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

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 scope of change is the process tree as a whole. 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.

4.3 Alignment Quality

As explained in Section 3.2, 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 $(a, \gg)$-moves and $(\gg, n)$-moves, such that the a-moves form the trace and the n-moves form a shortest possible valid sequence of leaf nodes. The γ^w alignment represents "the best of the worst" alignment. 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 function $\kappa : (A^{\gg} \times N^{\gg}) \rightarrow \mathbb{R}^+$: $\kappa(a, n) = +\infty$ if a and n's label do not match, $\kappa(a, n) = 5$ if $a \in A$ and $n = \gg$, $\kappa(a, n) = 2$ if $a = \gg$ and $n \in N$, and finally $\kappa(a, n) = 0$ if either $a \in A$ a and n's label match, or, $a = \gg$ and n's label is τ.

Figure 7 shows the 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

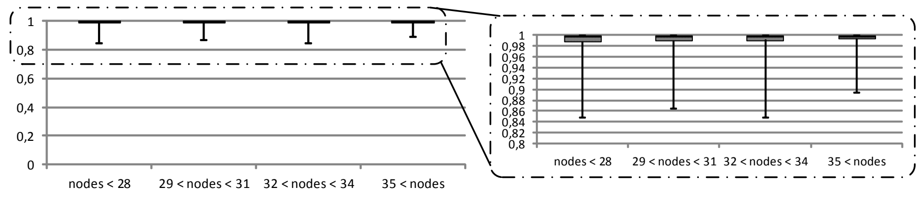


Fig. 7: Normalized *grade* of the repaire alignments.

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 of the scope of change and the potential loss of optimality. If the change is performed close to the root node, more moves of the previous alignment 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 do have optimality.

5 Related Work

Alignments were introduced in [3]. In [2] an alignment computation approach is presented based on the A^* algorithm. The concept presented in this paper, i.e. solving a sub-problem rather than the whole problem, is similar to methods that aim at decomposition of process mining techniques [13–15]. In [15] the authors present a decomposition technique that partitions process models and event logs into smaller parts that can be analyzed independently. A similar approach for data-aware conformance checking problems is presented in [14]. The main difference compared to these works is the fact that the presented technique results in an alignment for the *whole trace* and the *whole process model*, whereas decomposition techniques typically provide solutions for sub-problems, which in aggregated form provide lower bounds rather than a full solution.

6 Conclusion

We presented a novel approach to compute alignments based on an existing alignment, instead of (re)computing the alignment from scratch. The approach has been validated with a set of random trees and event logs. The evaluation shows that our approach always retrieves an alignment in a significantly lower, or equal, time than computing optimal alignments. Furthermore, we show that the potential loss of optimality is limited and stays within acceptable bounds.

We plan to improve and/or extend the approach as follows. 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. Moreover, we plan to develop means to predict optimality, allowing us to decide at which point it would be necessary to compute the optimal alignment instead of reusing and existing one. Based on the achieved results, we plan to apply the presented technique, if applicable, in different process mining domains, e.g. within handling concept drift in stream-based process discovery [16].

Acknowledgments. 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.

Bibliography

[1] Aalst, W.M.P.v.d.: Process Mining - Discovery, Conformance and Enhancement of Business Processes. Springer (2011)

[2] Adriansyah, A.: Aligning Observed and Modeled Behavior. PhD thesis, Eindhoven University of Technology (2014)

[3] Aalst, W.M.P.v.d., Adriansyah, A., Dongen, B.F.v.: Replaying History on Process Models for Conformance Checking and Performance Analysis. Wiley Interdisc. Rew.: Data Mining and Knowledge Discovery **2**(2) (2012) 182–192

[4] Adriansyah, A., Munoz-Gama, J., Carmona, J., Dongen, B.F.v., Aalst, W.M.P.v.d.: Measuring Precision of Modeled Behavior. Inf. Syst. E-Business Management **13**(1) (2015) 37–67

[5] Buijs, J.C.A.M.: Flexible Evolutionary Algorithms for Mining Structured Process Models. PhD thesis, Eindhoven University of Technology (2014)

[6] Vázquez-Barreiros, B., Mucientes, M., Lama, M.: ProDiGen: Mining Complete, Precise and Minimal Structure Process Models with A Genetic Algorithm. Inf. Sci. **294** (2015) 315–333

[7] Leemans, S.J.J., Fahland, D., Aalst, W.M.P.v.d.: Process and Deviation Exploration with Inductive Visual Miner. In: Proceedings of the BPM Demo Sessions 2014, Eindhoven, The Netherlands, September 10, 2014. (2014) 46

[8] Buijs, J.C.A.M., Dongen, B.F.v., Aalst, W.M.P.v.d.: Quality Dimensions in Process Discovery: The Importance of Fitness, Precision, Generalization and Simplicity. Int. J. Cooperative Inf. Syst. **23**(1) (2014)

[9] Eck, M.L.v., Buijs, J.C.A.M., Dongen, B.F.v.: 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. (2014) 291–303

[10] Dumas, M., Rosa, M.L., Mendling, J., Reijers, H.A.: Fundamentals of Business Process Management. Springer (2013)

[11] Leemans, S.J.J., Fahland, D., Aalst, W.M.P.v.d.: Discovering block-structured process models from event logs - A constructive approach. In: Petri Nets 2013, Milan, Italy, June 24-28. Proceedings. (2013) 311–329

[12] Aalst, W.M.P.v.d.: The Application Of Petri Nets to Workflow Management. Journal of Circuits, Systems and Computers **08**(01) (1998) 21–66

[13] Aalst, W.M.P.v.d.: Decomposing Petri Nets for Process Mining: A Generic Approach. Distributed and Parallel Databases **31**(4) (2013) 471–507

[14] Leoni, M.d., Munoz-Gama, J., Carmona, J., Aalst, W.M.P.v.d.: Decomposing Alignment-Based Conformance Checking of Data-Aware Process Models. In: OTM 2014, Amantea, Italy, October 27-31, 2014, Proceedings. (2014) 3–20

[15] Munoz-Gama, J., Carmona, J., Aalst, W.M.P.v.d.: Single-Entry Single-Exit Decomposed Conformance Checking. Inf. Syst. **46** (2014) 102–122

[16] Burattin, A., Sperduti, A., Aalst, W.M.P.v.d.: Heuristics Miners for Streaming Event Data. CoRR **abs/1212.6383** (2012)