Beyond Log and Model Moves in Conformance Checking: Discovering Process-Level Deviation Patterns

Michael Grohs[1][0000−0003−2658−8992], Han van der Aa[2][0000−0002−4200−4937], and Jana-Rebecca Rehse[1][0000−0001−5707−6944]

[1] University of Mannheim, Germany
{michael.grohs,rehse}@uni-mannheim.de
[2] University of Vienna, Austria
han.van.der.aa@univie.ac.at

Abstract. Process managers apply conformance checking techniques to identify deviations between the desired and the actual execution of a process. From a process-level perspective, these deviations often involve multiple interrelated events, for example if activities are executed in the wrong order or are unnecessarily repeated. However, state-of-the-art conformance checking techniques do not reveal these process-level deviations, instead identifying only event-level deviations in the form of inserted or skipped events. To address this shortcoming, this paper presents an approach that discovers process-level deviations from event-level insights provided by alignment-based conformance checking techniques. These deviations are discovered as instantiations of five commonly used patterns of non-conformance: inserted, skipped, repeated, replaced, and swapped. The approach is designed to choose patterns according to a user's preferences and contextualize them within parallelism and choices in the process model. Our evaluation shows that it reliably detects process-level deviations, thus providing process managers with more comprehensive information on process conformance.

Keywords: Process Mining · Conformance Checking · Deviation Patterns.

1 Introduction

Ensuring the compliance of business processes is paramount to an organization's success [9]. To manage this compliance, organizations commonly capture their desired processes in form of normative process models [11]. If process executions deviate from this desired behavior, detailed insights into those deviations will help process managers to prevent future non-compliance. To obtain these insights in an automated way, they can, for example, apply conformance checking [9], which detects deviations between recorded process executions and desired behavior, as captured in a process model.

Deviations occur in traces when the executed activities or their order differ from what is prescribed in the process model. From a process-level perspective, these deviations often involve multiple interrelated events, e.g., if activities are executed in the wrong order or are unnecessarily repeated. However, even state-of-the-art conformance checking techniques do not reveal these process-level deviations. Instead, they identify

deviations in a trace on event level, e.g., as log moves (inserted events) and model moves (skipped events) in alignments [10]. To illustrate this, consider a loan application process in which an approved loan is submitted for payment twice, rather than once. Then, an alignment would show a log move on *submit application*. However, this does not reveal that the activity was erroneously repeated, causing a double payout of the loan. This process-level deviation relates to multiple events (the first correct and second erroneous execution of *submit application*) and, thus, remains implicit. For this insight, a manager has to consult the entire alignment (not just the log and model moves) and the process model, a manual task that is complex for long traces with many deviations.

To address this shortcoming, this paper presents an approach that discovers process-level deviations from event-level deviations. Our goal is to provide process managers with the process-level information they need to fully understand the process-level deviations that occurred in traces, instead of the mere event-level information they get from alignments. For that, we rely on a set of five patterns commonly used to characterize deviations on the process level [15]: *inserted*, *skipped*, *repeated*, *replaced*, and *swapped* activities (or sequences thereof). Our approach makes use of the fact that instantiations of these patterns can be identified by aggregating event-level deviations, as found by alignment techniques [10]. From a given alignment, we extract all potential process-level deviations and choose the optimal interpretation of the deviations by means of a linear program. The approach chooses patterns according to a user's preferences and contextualizes them within parallelism and choices in the process model. Our evaluation shows that it accurately detects process-level deviations in nine labeled datasets.

The paper is structured as follows: After the preliminaries (Sect. 2), Sect. 3 illustrates the deviation discovery problem. Our approach is introduced in Sect. 4 and evaluated in Sect. 5. We discuss related work in Sect. 6, before concluding the paper Sect. 7.

2 Preliminaries

Event log. We adopt an event model that builds upon a set of activities $\mathcal{A}$. An event e, recorded by an information system, is assumed to be related to the execution of one of these activities. $\mathcal{E}$ denotes the universe of all events. For an event e, the function $act(e) : \mathcal{E} \rightarrow \mathcal{A}$ returns the activity to which e belongs. A single execution of a process, called a trace, is modeled as a sequence of events $t \in \mathcal{E}^*$, such that no event can occur in more than one trace. An event log is a set of traces, $L \subseteq 2^{\mathcal{E}^*}$. Two distinct traces that indicate the same sequence of activity executions are of the same (trace) variant.

Process model. A process model captures execution dependencies between the activities of a process. Our work can be applied to any process model M^A (independent of its notation) that defines a set of valid execution sequences over a set of activities $A \subseteq \mathcal{A}$, where each sequence should lead the process from an initial to a final state.

Alignments. An alignment σ between a trace t and a process model M^A is a sequence of moves that connects the activities (of the events) in t to an execution sequence of M^A [9]. Fig. 1 shows an example, with one move per column.[3] Each move is a pair (e_i, a_i) with $e_i \in \mathcal{E} \cup \{\gg\}$ and $a_i \in A \cup \{\gg\}$, where the

t	A	B	$\gg$	D
M^A	A	B	C	$\gg$

Fig. 1: An exemplary alignment σ

[3] Note that in this paper, any alignment visualization places the trace above the model sequence.

empty symbol $\gg$ denotes the lack of a counterpart. A move (e_i, a_i) is a synchronous move if $act(e_i) = a_i$ (e.g., (A, A) and (B, B)), a model move if $e_i = \gg$ and $a_i \in A$ (e.g., $(\gg, C)$), or a log move if $e_i \in t$ and $a_i = \gg$ (e.g., $(D, \gg)$). Model moves signal an activity missing from the trace, whereas log moves signal a superfluous activity.

A cost function $cf : (e_i, a_i) \to \mathbb{R}_0^+$ assigns non-negative cost to each move (e_i, a_i) in an alignment σ, with varying costs for different moves. Given t, M^A, and cf, σ is optimal if no other possible alignment has a lower cost than σ. The standard cost function penalizes log and model moves equally (i.e., $cf(e_i, \gg) = cf(\gg, a_i) = 1$) [9], but other cost functions allow the user to assign severity to certain moves [2] or define milestones in traces [7], thus influencing which alignment is considered optimal.

3 Problem Illustration

To illustrate the problem of discovering process-level deviations, consider the process model M_1^A for a loan application process in Fig. 2. This BPMN2.0 model consists of ten activities, from creating an application (CA) to its eventual acceptance (AA) or rejection (RA). Also consider a trace $t_1 = \langle$OR, AR, AR, CI, CC, PC, FA$\rangle$.

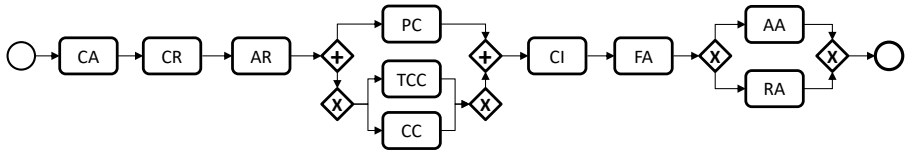

Fig. 2: Exemplary process model M_1^A

Aligning t_1 with M_1^A reveals multiple event-level deviations, shown in Fig. 3a. The model moves on CA, CR, CI, and AA show that these activities are missing from t_1; the log moves on OR, AR, and CI show that these activities are executed unnecessarily or at a wrong position. Although these moves indicate where the trace deviates from the model, they are not connected to each other. As a result, the following underlying issues of t_1 (which we refer to as process-level deviations) are not explicitly revealed:

- Instead of creating an application (CA) and a corresponding request (CR), the case starts by opening a previously rejected application (OR).
- The request is repeatedly assigned (AR), rather than just once.
- Interest is calculated (CI) before conducting the personal (PC) and credit checks (CC), rather than after.
- At the end of the process, no decision is taken, since the application is neither accepted (AA) nor rejected (RA).

The problem is that the insights provided by an alignment are far removed from these issues. Therefore, this paper introduces an approach that can discover such process-level deviations in alignments to provide managers with the information they need to fully understand deviating process behavior. However, this task faces two challenges:

≫	≫	OR	AR	AR	CI	CC	PC	≫	FA	≫
CA	CR	≫	AR	≫	≫	CC	PC	CI	FA	AA

(a) Alignment σ_1 of t_1 and M_1^A

≫	OR	≫	AR	AR	CI	CC	PC	≫	FA	≫
CA	≫	CR	≫	AR	≫	CC	PC	CI	FA	RA

(b) Altern. alignment σ_2 of t_1 and M_1^A

Fig. 3: Two optimal alignments of t_1 and M_1^A under the standard cost function

≫	≫	OR	AR	AR	CI	CC	PC	≫	FA	≫
CA	CR	≫	AR	≫	≫	CC	PC	CI	FA	AA
OR replaces CA, CR			AR repeated		CC, PC swapped with CI				AA or RA skipped	

(a) Interpretation 1

≫	≫	OR	AR	AR	CI	CC	PC	≫	FA	≫
CA	CR	≫	AR	≫	≫	CC	PC	CI	FA	AA
CA skipped		OR replaces CR		AR, CI inserted				CI skipped		AA or RA skipped

(b) Interpretation 2

Fig. 4: Two mappings of alignment moves in σ_1 to process-level deviations

Challenge 1: Mapping alignment moves to process-level deviations. There can be different ways to map the moves of a given alignment to process-level deviations [10]. For example, Fig. 4 shows two so-called *interpretations* of the moves in σ_1. Among other differences, Interpretation 1 shows a single process-level deviation at the start of the case, replacing CA and CR by OR, whereas Interpretation 2 separates this into two distinct issues: skipping CA, before replacing CR with OR. In light of this, our approach has three beneficial properties when determining which interpretation to return:

P.1 It only discovers process-level deviations that are instantiations of established deviation patterns and known to be practical and useful for managers, i.e., *inserted*, *skipped*, *repeated*, *replaced*, and *swapped* [15].

P.2 It identifies process-level deviations that are maximal, thus returning fewer, but larger deviations where possible. For example, our approach would return Interpretation 1 from Fig. 4, since this recognizes that the trace starts with one deviation, i.e., OR replaced both CA and CR, instead of two separate, smaller ones.

P.3 It is customizable, allowing users to select and prioritize patterns according to their preferences. For example, in an auditing context, it was found that the *replace* pattern was too abstract to provide value [15]. In this case, our approach can be configured to avoid interpretations that contain such replacements.

Challenge 2: Dealing with non-deterministic alignment search. Often, multiple optimal alignments can exist for the same trace, differing in the order of the contained moves [9, p. 143] or even in the moves themselves [9, p. 161]. The algorithms (and their implementations) used to search for alignments are, to some degree, non-deterministic, meaning that it is typically unclear which of the optimal alignments they return. Since the choice for a particular alignment can affect the process-level deviations derived from it, our approach has two properties to limit the impact of non-determinism on its results:

P.4 It nudges its alignment search in a way that ensures that so-called *independent moves* appear in a specific order in the obtained optimal alignment. Moves are independent if they can be arbitrarily ordered without affecting the validity or optimality of an alignment. Common examples are model moves on parallel activities or sequences of log and model moves, which can be interleaved arbitrarily [9, p.143]. We exploit this notion of independence to obtain alignments that, where possible, contain synchronous moves first, followed by model and then log moves, which simplifies the detection of process-level deviations. To illustrate this, consider alignments σ_1 and σ_2 from Fig. 3. In σ_1, it is easier to recognize that $\langle$CA, CR$\rangle$ was replaced

by OR, since it groups the two log moves together, whereas they are separated in σ_2. Similarly, σ_1 clearly reveals that activity AR was unnecessarily repeated, since it places the correct execution of AR before its undesired repetition, whereas σ_2 uses the reverse order.

P.5 It contextualizes process-level deviations in light of choices and parallelism in a process model. This way, the results are not affected by non-determinism with respect to arbitrary selection of exclusive moves, or arbitrary orders of parallel model and synchronous moves. For example, σ_1 and σ_2 differ in their final moves, where in σ_1, the request was not *accepted*, but in σ_2, it was not *rejected*. Our approach considers the execution semantics of M_1^A to recognize that the choice between these two moves is arbitrary and should, thus, not affect the results. Therefore, no matter which move appears in the alignment, it will reveal the true issue: The entire choice at the end of the process was skipped. Similarly, for PC and CC, which are concurrent in M_1^A, we will discover the same process-level deviation involving these activities, independent of their order in a trace or its alignment.

In the following section, we describe an approach that has these properties to discover process-level deviations from event-level deviations.

4 Approach

This section presents our approach for the discovery of process-level deviations. Given a process model, a trace, and a cost function, it consists of four steps (cf. Fig. 5):

(1) Establish an optimal alignment for the trace, which involves nudging the alignment search towards a specific order over move types (Sect. 4.1).
(2) Extract possible process-level deviations in the alignment as instantiations of deviation patterns, covering all potential mappings of moves to deviations (Sect. 4.2).
(3) Determine an interpretation of a trace's event-level deviations, defined as an optimal mapping of all moves to a subset of the possible process-level deviations (Sect. 4.3).
(4) Contextualize process-level deviations in light of choices and parallelism in the process model (Sect. 4.4).

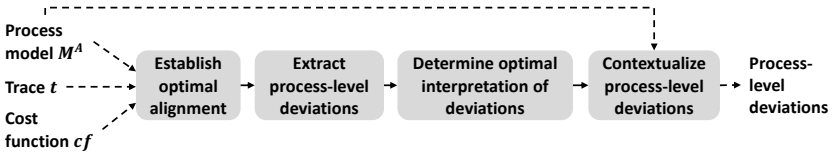

Fig. 5: Overview of our approach for process-level deviation pattern discovery

4.1 Establish Optimal Alignment

This first step establishes an optimal alignment σ for trace t and process model M^A under cost function cf. To obtain P.4, we strive to obtain alignments in which independent moves are ordered by type, with synchronous moves first, followed by model and then log moves. This will later simplify the detection of repetitions and replacements.

The idea of re-ordering moves in a cost-optimal alignment, while preserving all other characteristics, is well-established [9, p. 143] and can be achieved in several ways,

e.g., by adjusting the search algorithm for alignments, post-processing obtained alignments, or adjusting the cost function used to find them. We opt for the latter strategy, involving minute adjustments to the cost function cf, since it makes our approach independent of a specific alignment search algorithm or implementation. To obtain the desired move order (synchronous, model, log), we increase the costs for synchronous moves and decrease the costs of log moves by a small factor ϵ per step in the alignment. This nudges the alignment search to choose synchronous moves as early and log moves as late as possible (all else being equal). We adapt the cost function as follows:

$$cf'(e_i, a_i) = \begin{cases} cf(e_i, a_i) + i \times \epsilon, & \text{if } act(e_i) = a_i \text{ (synchronous moves)} \\ max(cf(e_i, a_i) - i \times \epsilon,\ 0) & \text{if } a_i = \gg \text{ (log moves)}[4] \\ cf(e_i, a_i), & \text{otherwise (model moves)} \end{cases}$$

We set ϵ so that $\epsilon \times |t|$ is much smaller than any difference between the costs of moves in cf. This ensures that an alignment that is optimal under cf' is also optimal under cf.

To illustrate this, consider the log and synchronous move on AR in Fig. 3. Under the standard cost function, the moves are independent and can be placed at positions 4 and 5 in the alignment, in either order. However, under the adapted cost function cf', these move orders receive different costs:

- $\langle(\text{AR}_4, \text{AR}_4), (\text{AR}_5, \gg_5)\rangle$, costs $(0 + 4 \times \epsilon) + (1 - 5 \times \epsilon) = 1 - \epsilon$.
- $\langle(\text{AR}_4, \gg_4), (\text{AR}_5, \text{AR}_5)\rangle$, costs $(1 - 4 \times \epsilon) + (0 + 5 \times \epsilon) = 1 + \epsilon$.

Using cf' would thus lead to the first order (costing $1-\epsilon$), thereby achieving the goal of having the synchronous move ahead of the log move in an obtained optimal alignment.

4.2 Extract Process-Level Deviations

Based on the optimal alignment σ obtained for trace t, this step extracts a set of possible process-level deviations $\mathcal{PD}$ for t. This is achieved by aggregating the log and model moves in σ into instantiations of specific deviation patterns (property P.1). We first describe these patterns, before presenting our instantiation procedure.

Deviation patterns. Our approach expresses process-level deviations in terms of an established set of five deviation patterns [15], referred to as *inserted*, *skipped*, *repeated*, *replaced*, and *swapped*.[5] As illustrated in Fig. 6, these patterns are defined as combinations of alignment moves on *process fragments* (one or multiple consecutive activities in the trace or model). *Inserted* and *skipped* are simple patterns consisting of one move type, whereas the other three are more complex, combining different move types:

- The *inserted* pattern denotes that a trace contains a fragment that should not have occurred at that point in the trace, recognizable as one or more consecutive log moves.
- The *skipped* pattern denotes that a trace misses a fragment that should have occurred at a given part in the trace, recognizable as one or more consecutive model moves.
- The *repeated* pattern denotes the undesired re-execution of a process fragment that has been previously executed in accordance with the model (e.g., activity “AR" in Fig. 6 should have been performed once, rather than twice).

[4] We avoid log moves with negative costs, which could cause issues in the alignment search. However, this only affects the uncommon case where some log moves have cost of 0 in cf.

[5] Note that the original source [15] also proposes a *loop* pattern, but in the context of alignments, this is simply a specific version of the *repeated* pattern.

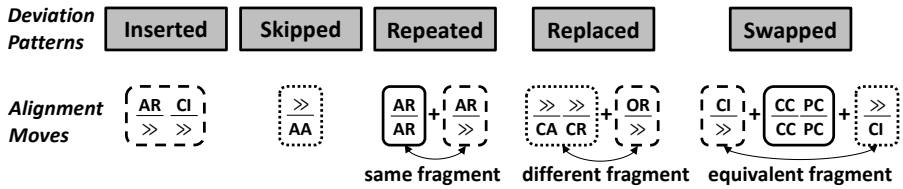

Fig. 6: Illustration of deviation patterns based on the running example

- The *replaced* pattern captures that a fragment of a trace should not have been executed, but rather replaced a different required fragment (e.g., activity "OR" was performed, but fragment "CA,CR" is required according to the process model).
- The *swapped* pattern captures that two fragments were performed in the wrong order in a trace (e.g., "CI" was performed before "CC,PC", rather than vice versa).

Pattern instantiation. Our approach applies Alg. 1 to discover all instantiations of the deviation patterns in an alignment σ. The algorithm iterates over the moves in σ, checking for each move $\sigma[i]$ if it is part of instantiations of the five pattern types. Note that these checks are, purposefully, not exclusive, meaning that the same move can be part of multiple different process-level deviations in the result set $\mathcal{PD}$.

Inserted and skipped. Identifying insertions and skips is straightforward since these simply correspond to (sequences of) log or model moves, respectively. Given a non-synchronous move $\sigma[i]$, Alg. 1 thus creates an *Insertion* (or *Skip*) deviation for each sequence of one or more consecutive log (or model) moves (lines 4–6).

Repeated. Thanks to the move order in σ (see Sect. 4.1), repetitions consist of synchronous moves followed by log moves on the same activities. Alg. 1 checks whether a move $\sigma[i]$ is a log move (i.e., an unnecessary step) and whether a synchronous move on the activity of $\sigma[i]$ previously occurred in σ (line 7). If both hold, we instantiate a repetition (line 8), which is extended if consecutive activities are repeated (lines 9–12).

Replaced. Thanks to the move order in σ, replacements consist of (a sequence of) model moves, directly followed by (a sequence of) log moves. Given a model move $\sigma[i]$, Alg. 1 thus first gets a sequence seq_{model} of consecutive model moves starting from $\sigma[i]$ (line 14), and a sequence seq_{log} of log moves starting right afterwards. If seq_{log} is not empty (i.e., line 17), a replacement has been discovered.

Swapped. A swap consists of three distinct move sequences. The first sequence seq_1 contains moves on activities either performed too early (log moves) or too late (model moves). It is followed by a sequence seq_2 of synchronous moves that should have happened before or after the moves in seq_1, depending on the move type. The last sequence seq_3 is the inverse of seq_1 (i.e., moves of opposite type on the same activities in seq_1), representing the counterpart for the activities performed too early or too late. For example, in Fig. 6, $seq_1 = (CI, \gg)$, $seq_2 = \langle (CC, CC), (PC, PC) \rangle$, and $seq_3 = (\gg, CI)$.

To identify swaps, Alg. 1 collects seq_1, seq_2 and a candidate sequence seq'_3, starting from move $\sigma[i]$ (lines 20–23). When comparing seq'_3 to seq_1, two aspects need to be considered. First, seq'_3 does not have to contain the inverse moves of seq_1 but can also contain *behaviorally equivalent* ones. This can happen in the presence of choices, e.g., $(\gg, RA)$ is behaviorally equivalent to the inverse of $(AA, \gg)$, and parallelism, e.g., $\langle (\gg, PC), (\gg, CC) \rangle$ is behaviorally equivalent to the inverse of $\langle (CC, \gg), (PC, \gg) \rangle$.

Algorithm 1 Process-Level Deviation Extraction

Input Alignment σ, process model M^A
Output Set of process-level deviations $\mathcal{PD}$

1: $\mathcal{PD} \leftarrow \emptyset$ ▷Initialize result set
2: **for** $i = 0$ to $|\sigma| - 1$ **do** ▷Loop over moves in alignment σ
▷ Detect insertions and skips:
3: **if** $\mathsf{type}(\sigma[i]) \neq$ sync **then** ▷Check if $\sigma[i]$ is a log or model move
4: $seq \leftarrow \mathsf{getConsecMovesOfType}(i, \mathsf{type}(\sigma[i]))$ ▷Get sequence of consecutive moves
5: **for** $j = 0$ to $|seq| - 1$ **do**
▷ Add deviation for sub-sequence of length $j + 1$ according to move type
6: $\mathcal{PD}$.add(new $\mathsf{Insertion/Skip}(seq[0 : j])$)
▷ Detect repetitions:
▷ Check if $\sigma[i]$ is a log move and was preceded by a corresponding synchronous move
7: **if** $\mathsf{type}(\sigma[i]) =$ log $\wedge$ $\mathsf{syncMoveBefore}(i, \mathsf{act}(\sigma[i]))$ **then**
8: $seq \leftarrow \langle \sigma[i] \rangle$ ▷Initialize sequence of repeated steps with move $\sigma[i]$
9: **for** $j = i + 1$ to $|\sigma| - 1$ **do** ▷Loop over consecutive moves
▷ Check if consecutive move $\sigma[j]$ is also a repetition
10: **if** $\mathsf{type}(\sigma[j]) =$ log $\wedge$ $\mathsf{syncMoveBefore}(i, \mathsf{act}(\sigma[j]))$ **then**
11: seq.append($\sigma[j]$) ▷Expand the sequence of repeated steps
12: **else**: **break** ▷Otherwise stop checking consecutive moves
13: $\mathcal{PD}$.add(new $\mathsf{Repetition}(seq)$) ▷Add process-level deviation
▷ Detect replacements:
14: **if** $\mathsf{type}(\sigma[i]) =$ model **then** ▷$\sigma[i]$ is model move?
▷ Get consecutive model moves starting from i
15: $seq_{model} \leftarrow \mathsf{getConsecMovesOfType}(i, \text{model})$
▷ Get all log moves directly after seq_{model} (if any)
16: $seq_{log} \leftarrow \mathsf{getConsecMovesOfType}(i + |seq_{model}|, \text{log})$
17: **if** $seq_{log} \neq \langle\rangle$ **then**
18: $\mathcal{PD}$.add(new $\mathsf{Replacement}(seq_{model}, seq_{log})$) ▷Add process-level deviation
▷ Detect swaps:
19: **if** $\mathsf{type}(\sigma[i]) \neq$ sync **then**
▷ Get all moves directly after $\sigma[i]$ that are of the same type
20: $seq_1 \leftarrow \mathsf{getConsecMovesOfType}(i, \mathsf{type}(\sigma[i]))$
▷ Get all synchronous moves directly after seq_1 (if any)
21: $seq_2 \leftarrow \mathsf{getConsecMovesOfType}(i + |seq_1|, \text{sync})$
▷ Get all moves directly after seq_2 of the opposite type as seq_1 (if any)
22: **if** $\mathsf{type}(\sigma[i]) =$ log **then** $type_3 \leftarrow$ model **else** $type_3 \leftarrow$ log
23: $seq'_3 \leftarrow \mathsf{getConsecMovesOfType}(i + |seq_1| + |seq_2|, type_3)$
▷ Keep only the part of seq'_3 that is the inverse of seq_1 (if any)
24: $seq_3 \leftarrow \mathsf{findEquivSeq}(seq'_3, seq_1, M^A)$
25: **if** $seq_3 \neq \langle\rangle$ **then**
26: $\mathcal{PD}$.add(new $\mathsf{Swap}(seq_1, seq_2, seq_3)$) ▷Add process-level deviation
27: **return** $\mathcal{PD}$

Second, seq'_3 may contain moves that are not relevant for the swap, but relate to other deviations. Therefore, function `findEquivSeq` looks for a sequence seq_3 that is a projection of seq'_3, consisting of the inverse moves of seq_1 (or a behavioral equivalent of them, given M^A). If such a projection exists, a swap is discovered (line 26). We provide the details of this function in our project's repository, linked in Sect. 5.

Algorithm 1 ensures that every potential instantiation of a pattern in σ is identified. However, $\mathcal{PD}$ will also include non-maximal subsets of instantiations, as well as different process-level deviations related to the same moves. The next step sorts this out by identifying an optimal subset of $\mathcal{PD}$.

4.3 Determine Optimal Interpretation of Deviations

Since there can be different ways to map event-level deviations to process-level deviations, we now select a subset $PD_{opt} \subseteq \mathcal{PD}$ that interprets σ in an optimal manner, i.e., it uses maximal process-level deviations (P.2) and respects the customizable user preferences (P.3). To determine PD_{opt}, we define an optimization problem by means of a linear program (LP) [22]. LPs can be used to determine an optimal solution to a mathematical model by optimizing a linear objective function for a set of decision variables, subject to linear constraints. In our case, we solve a special type of LP, called binary program (BP), in which all decision variables are binary:

Input Parameters

$\mathcal{ED}$	Set of all event-level deviations, i.e., all log and model moves in σ
$\mathcal{PD}$	Set of extracted process-level deviations in the trace
$\rho_{ed,pd}$	Parameter capturing if $ed \in \mathcal{ED}$ is part of $pd \in \mathcal{PD}$, with $\rho_{ed,pd} = 1$ if true, else 0
α_{pd}	Penalty of deviation pd, based on its pattern type; default: $\alpha_{pd} = 1.0$ if swapped, 1.1 if replaced, 1.2 if repeated, 1.3 if inserted/skipped

Decision Variable

Y_{pd}	Decision whether $pd \in \mathcal{PD}$ is included in PD_{opt}, with $Y_{pd} = 1$ if true, else 0

Objective Function

$$\text{Minimize} \sum_{pd \in \mathcal{PD}} Y_{pd} \times \alpha_{pd} \quad \text{subject to} \sum_{pd \in \mathcal{PD}} Y_{pd} \times \rho_{ed,pd} = 1 \qquad \forall\, ed \in \mathcal{ED}$$

Our BP takes as input the sets of event-level ($\mathcal{ED}$, consisting of all log and model moves in σ) and process-level deviations ($\mathcal{PD}$), the parameters $\rho_{ed,pd}$, capturing whether an $ed \in \mathcal{ED}$ is part of $pd \in \mathcal{PD}$, and a penalty α_{pd} for each pd, based on its pattern type. It has one (binary) decision variable Y_{pd} per process-level deviation $pd \in \mathcal{PD}$, where $Y_{pd} = 1$ indicates that deviation pd is part of the optimal interpretation PD_{opt}. The resulting interpretation PD_{opt} fulfils P.2 and P.3 as follows:

P.2 PD_{opt} contains maximal process-level deviations because the objective function minimizes the sum of the decision variables such that PD_{opt} contains the largest process-level deviations. PD_{opt} is guaranteed to cover each event-level deviation $ed \in \mathcal{ED}$ exactly once due to the constraint $\sum_{pd \in \mathcal{PD}} Y_{pd} \times \rho_{ed,pd} = 1$, $\forall ed \in \mathcal{ED}$.

P.3 The customizable user preferences are reflected by α_{pd}, which assigns a priority to certain pattern types: The lower the penalty for a type, the higher its preference. If the assigned penalties differ slightly per pattern type (as is the case in the default setting), this also ensures consistency across similar inputs, where multiple interpretations with different pattern types would minimize the objective function.

Multiple readily-available solvers can be applied to solve the program. They all vary the decision variables in their allowed value range to minimize the objective function, guaranteeing an optimal solution. We use the CPLEX solver [16]. Applied to our running example, the BP aggregates the event-level deviations into four process-level deviations, yielding the outcome previously illustrated in Fig. 4a.

4.4 Contextualize Process-Level Deviations

In this final step, the elements of the optimal interpretation PD_{opt} are contextualized within choices and parallelism in the process model to reflect generalized process-level insights, achieving property P.5. Concretely, we first extract all block structures from a process model M^A, identifying those indicating choices (XOR-blocks) and parallelism (AND-blocks). We use these blocks to indicate when process-level deviations in PD_{opt} apply to an entire choice or parallel construct, rather than just individual paths through them. Finally, the deviations in PD_{opt} are verbalized and presented to the user.

Extracting block structures. A block structure consists of multiple paths (i.e., branches) through a part of a process model that all start in one and end in another activity (in graph-based process models, these are *single-exit-single-entry* fragments) [17]. We focus on XOR-blocks, in which branches are mutually exclusive (e.g., the choice between AA and RA in Fig. 2), and AND-blocks, in which branches can be executed in an arbitrary order (e.g., the parallel construct between PC and the choice of TCC and CC).

To obtain block structures for a process model M^A, we use an existing PM4Py function [6], yielding sets $\mathcal{X}$ of extracted XOR-blocks and $\mathcal{C}$ of AND-blocks. We represent these block structures in the form of process trees [17], to compactly capture them. These process trees are mathematical trees in which leaves are labeled with activities $a \in A$ and internal nodes are labeled with operators $o \in \phi$, where the set of operators $\phi = \{\rightarrow, \times, \wedge, \circlearrowleft\}$ indicate sequence, choice, parallel, or loop behavior, respectively. For our running example, we obtain $\mathcal{X} = \{\times$(AA,RA), $\times$(TCC,CC)$\}$ and $\mathcal{C} = \{\wedge$(PC,$\times$(TCC,CC))$\}$, with the second XOR-block nested within the AND-block.

Indicating missing XOR-blocks. So far, *skipped* and *replaced* deviations capture a missing sequence of activities in a trace. However, if the missing sequence is a branch in an XOR-block, this specific branch was arbitrarily chosen during the alignment search. Thus, we want to generalize that this entire XOR-block is missing, rather than just one branch of it. For instance, rather than capturing that *Accept Application* (AA) is missing, we indicate that the choice between rejecting or accepting an application was missing, i.e., that the block $\times$(AA,RA) was skipped. We achieve this by checking if the sequence of model moves in $pd \in PD_{opt}$ (if it is a *skip* or *replacement*) corresponds to a valid sequence of an XOR-block in set $\mathcal{X}$. If so, we refine pd with that information.

Indicating deviating AND-blocks. We also want to recognize when different traces contain the same process-level deviation involving concurrent activities in an AND-block, independent of the order of these activities in the trace. We achieve this by checking if a fragment of deviation $pd \in PD_{opt}$, consisting of moves of the same type, corresponds to an execution sequence of an AND-block in $\mathcal{C}$. If so, we refine pd with that information. In this way, we would, for instance, be able to capture that swapping *Credit Check, Personal Check* (CC,PC) with the activity *Calculate Interest* (CI), corresponds to the same deviation as a swap of (PC,CC) with CI in another trace.

Output of process-level deviations. Our approach yields a set of contextualized process-level deviations PD_{opt} per trace. When presenting the results to a user, we verbalize PD_{opt} using a standard template for each pattern in order to make it easier to interpret. In our running example, our approach then yields the output in Fig. 7.

Finally, we can aggregate the process-level deviations of all traces in an event log L, resulting in a multi-set of identified issues and their respective frequencies.

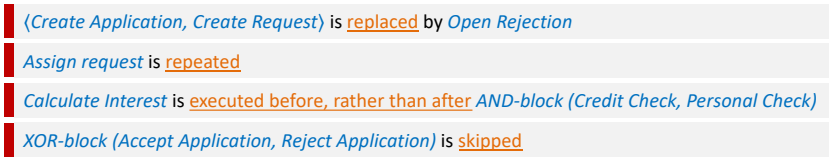

Fig. 7: Exemplary output for our running example

5 Evaluation

We implemented our approach in a Python prototype[6], using the PM4Py library [6]. Based on that, we evaluate its capability to re-discover process-level deviations in labeled data (Sect. 5.1) and illustrate its potential practical value by showing that it can find process-level deviations in real-life scenarios (Sect. 5.2). Its computational complexity is dominated by the alignment computation in step 1, which can be exponential in the size of the trace and the model. In practice, however, our complete approach took less 50 seconds for each log-model pair in this section, as shown in our repository.

5.1 Re-discovering Process-Level Deviations in Labeled Datasets

This section describes experiments conducted to assess whether our approach can detect known process-level deviations in event logs, where each trace is individually labeled by independent researchers with the process-level deviations it contains.

Data. We evaluate our approach on nine synthetic, labeled event logs as illustrated in Tab. 1. The first log [15] was created for the (manual) identification of deviation patterns in traces; traces can contain multiple process-level deviations. The remaining eight logs [19] were created for anomaly detection. Based on eight process models, the authors simulated event logs with control-flow anomalies to evaluate whether these can be detected without a process model. Each trace contains at most one anomaly. The anomalies are grouped into the categories *inserted*, *skipped*, *rework* (corresponds to *repeated*), *early*, and *late* (both correspond to *swapped*). These logs do not contain replacements. Although created for a different purpose, we can use these labelled traces to check if we can discover process-level deviations w.r.t. the provided process models.

Baselines. To illustrate the importance of the different parts of our approach, we compare its performance to versions of our approach that skip certain steps:

(a) *Omit Step 1*: This baseline does not use the adjusted alignment cost function to order moves in the alignment. Instead, it uses the standard cost function, leading to less-deterministic move orders in obtained alignments.
(b) *Omit Step 2 & 3*: This baseline does not aim to establish any process-level deviations but just returns the most basic interpretation, achieved by simply mapping the log and model moves to *inserted* and *skipped* patterns.
(c) *Omit Step 4*: This baseline does not aim to contextualize deviations within exclusive and concurrent behavior. Instead, it skips the last step of the approach and returns the non-contextualized process-level deviations.

[6] https://dx.doi.org/10.6084/m9.figshare.25942474

Table 1: Descriptive statistics for used event logs

	[15]	[19]							
		Gigantic	Huge	Large	Medium	P2P	Paper	Small	Wide
Activities in Model	9	154	109	85	65	27	27	41	68
Events in Log	152	29,829	43,210	57,524	31,991	43,193	56,814	43,437	31,910
Deviating Variants	18	671	796	1115	662	583	647	653	559

Benchmark. We compare our approach to existing work by García-Bañuelos et. al. [12], which also detects deviation patterns in event data. In contrast to our work, their approach targets an entire event log at once, rather than detecting patterns per trace. Still, we can use it as a benchmark for our approach by applying it to (logs consisting of) one trace at a time. When used per trace, their approach returns patterns that can be directly mapped to the patterns that we use. Next to that, it detects additional patterns that are only visible when considering all traces at once, e.g., related to concurrency in the event log. We provide details on the adaptations necessary to make their approach work for individual traces, as well as further conceptual differences in our repository.

Metrics. We measure the capability to discover process-level deviations using precision and recall (Eq. 1). If a trace contains a certain process-level deviation, its classification is a true positive (TP) if both pattern and fragment of the deviation are identified correctly and a false negative (FN) otherwise. The logs from [19] do not label fragments, so they are assessed based on pattern only. Conversely, a trace without a certain process-level deviation is a true negative (TN) if that deviation is not detected in it and a false positive (FP) otherwise. A classification that does not properly contextualize a deviation in an XOR- or AND-block is counted as half-correct, increasing both FN and TP by 0.5.

$$Prec. = \frac{TP}{TP + FP} \qquad Rec. = \frac{TP}{TP + FN} \tag{1}$$

Results. Table 2 displays precision and recall for the nine event logs and compares the performance of our full approach to the baselines and the benchmark. Values for the eight event logs of [19] are averages, information per log can be found in our repository.

Our full approach has perfect recall and precision for all deviation patterns in all event logs. Thus, we were able to correctly discover all process-level deviations accord-

Table 2: Precision and recall for used event logs

Log	**Pattern**	**Support**	**Full Approach**		**Omit Step 1**		**Omit Step 2 & 3**		**Omit Step 4**		**[12]**	
			Prec.	**Rec.**	**Prec.**	**Rec.**	**Prec.**	**Rec.**	**Prec.**	**Rec.**	**Prec.**	**Rec.**
[15]	inserted	6	1.00	1.00	0.60	1.00	0.24	1.00	1.00	1.00	0.06	0.83
	skipped	5	1.00	1.00	1.00	1.00	0.42	1.00	1.00	0.90	0.23	1.00
	repeated	12	1.00	1.00	1.00	0.67	0.00	0.00	1.00	1.00	0.27	0.55
	replace	3	1.00	1.00	1.00	1.00	0.00	0.00	1.00	1.00	0.27	1.00
	swap	4	1.00	1.00	1.00	1.00	0.00	0.00	1.00	1.00	0.27	1.00
[19]	inserted	1919	1.00	1.00	0.87	1.00	0.48	1.00	1.00	1.00	0.42	1.00
	skipped	692	1.00	1.00	1.00	1.00	0.42	1.00	1.00	0.64	0.38	0.99
	repeated	1163	1.00	1.00	1.00	0.75	0.00	0.00	1.00	1.00	0.43	1.00
	swap	1912	1.00	1.00	1.00	1.00	0.00	0.00	1.00	1.00	0.64	0.39

ing to the provided labels. However, the performance of the baselines and the benchmark indicate that only our full approach addresses this task in a satisfactory way.
Omit Step 1. If we do not enforce a certain order of alignment moves, we obtain lower recall values for *repetitions*, meaning that fewer are correctly discovered. Instead, these deviations are interpreted as *insertions*, shown by lower precision values for that pattern. Due to the nature of this task, the results for this baseline are highly non-deterministic, so another execution of this baseline might obtain different values.
Omit Step 2 & 3. If we do not establish any process-level deviations, we obtain lower precision values for the *inserted* and *skipped* patterns, which means that they are detected too frequently, whereas the three other patterns are not detected at all.
Omit Step 4. If we do not contextualize process-level deviations within XOR- and AND-blocks, we obtain lower recall values for *skipped* patterns, showing that they are not correctly identified. This effect does not occur for the *inserted* and *repeated* patterns as they are simulated as additional, model-external activities, which are not part of an XOR- or AND-block. Further, it does not affect *swapped* patterns as no complete AND-blocks are swapped. It is also not visible for replacements because those in [15] do not need to be contextualized and there are no replacements in [19].
Benchmark [12]. The benchmark approach performs worse than our approach, especially with respect to precision. This is because it does not identify maximal deviations, instead returning separate deviations for each element of a deviating process fragment, leading to many false positives. To the same effect, the returned deviations are not mutually exclusive, meaning that the approach returns e.g., a replacement of A by B, an insertion of B, and a skip of A instead of just the replacement. Further, the approach does not uncover all process-level deviations, manifesting in worse recall. This happens if the approach returns only one of many deviations in a trace and if it mixes up different pattern types and, e.g., returns insertions as repetitions. In addition, some swaps are missed completely and replacements are wrongly discovered in the logs from [19].

Performance between the logs differs due to their individual characteristics. For example, the logs of [19] have relatively many *inserted* and *skipped* patterns, meaning that omitting Steps 2 and 3 has a less drastic impact than for [15]. Further, the log from [15] contains many repetitions in form of loops, which are not correctly recognized when omitting Step 1, indicated by lower recall. The benchmark performs worse for [15] as it is less able to detect the more complex process-level deviations in it.

5.2 Illustration of Process-Level Deviation Discovery in Real-Life Application

We next demonstrate the potential practical value of our approach by showing that it reveals process-level deviations in a well-known real-life event log. For that, consider the loan application process in Fig. 8. It is a part of the BPI Challenge 2012 process, where all activities start with A_ (BPIC 12A). After submission, the approval process moves through four steps, with the option to be terminated by either A_DECLINED and A_CANCELLED at any stage. If, after the four steps, the loan is approved, the two parallel activities A_REGISTERED and A_ACTIVATED will follow. Although this is a rather simple process, it serves as a suitable example for our approach as it clearly illustrates how event-level deviations can be set into process-level context.

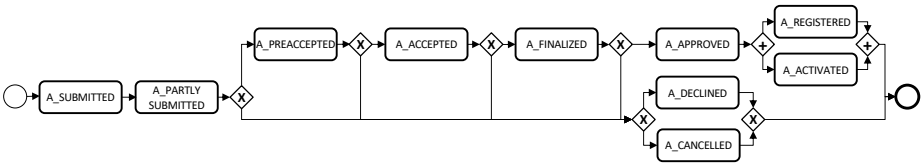

Fig. 8: Process model for BPI Challenge 2012 (only A_ Activities)

For this process, the BPIC 2012 event log[7] contains 13,087 execution traces in 17 variants. In the seven variants that deviate from the model, our approach discovers four distinct process-level deviations, shown in Tab. 3. Among these deviations, the first three are very similar: They involve a *swap* of A_APPROVED with either A_REGISTERED, A_ACTIVATED, or both. For the two variants where both activities are swapped, we identify the same process-level deviation, since they are contained in an AND-block. The fourth process-level deviation occurs in three variants, which are terminated prematurely without executing either A_CANCELLED or A_DECLINED. Our approach discovers the *skipping* of the XOR-block in all three variants.

Table 3: Discovered process-level deviations for deviating variants in BPIC 12A

Process-Level Deviation	**Deviating Variant**	**Count**
APP swapped with REG	⟨SUB,PAR,PRE,ACC,FIN,REG,APP,ACT⟩	532
APP swapped with ACT	⟨SUB,PAR,PRE,ACC,FIN,ACT,APP,REG⟩	322
APP swapped with ∧(REG, ACT)	⟨SUB,PAR,PRE,ACC,FIN,ACT,REG,APP⟩	154
	⟨SUB,PAR,PRE,ACC,FIN,REG,ACT,APP⟩	183
x(CAN, DEC) skipped	⟨SUB,PAR,PRE,ACC,FIN⟩	327
	⟨SUB,PAR,PRE⟩	69
	⟨SUB,PAR,PRE,ACC⟩	3

SUB = A_SUBMITTED; PAR = A_PARTLY SUBMITTED; PRE = A_PREACCEPTED;
ACC = A_ACCEPTED; FIN = A_FINALIZED; REG = A_REGISTERED;
APP = A_APPROVED; ACT = A_ACTIVATED; CAN = A_CANCELLED; DEC = A_DECLINED

6 Related Work

The adjustment of alignment cost functions as well as the discovery of high-level deviation patterns has been addressed by previous works. Deviation patterns have also been used in research on model repair.

Adjustment of alignment cost function. Alignment cost functions can be adjusted to obtain alignments with specific characteristics. For example, the cost function can be defined based on historic data to calculate alignments closest to previous conforming traces [3]. It can also incorporate the severity of (context-aware) deviations, including process-level deviations that are known to exist [1, 2]. Alternatively, the cost function can be adjusted to maximize the number of synchronous moves and introduce unskippable events in alignments [7] or to increase computational efficiency [8]. Similarly,

[7] https://doi.org/10.4121/UUID:3926DB30-F712-4394-AEBC-75976070E91F

Sect. 4.1 describes how we adapt the cost function provided as input. However, our adaptations do not change any characteristics of the resulting alignment, except for the order of independent moves. Hence, our approach can be used in conjunction with any cost function, including those in the cited papers.

Deviation pattern discovery. In a framework to categorize deviation patterns [10], the authors stress that process-level deviations can be discovered from skipped and inserted activities. Although an algorithmic idea for that is sketched, no approach is provided.

Alternatively, two existing approaches discover process-level deviations on the log level instead of the trace level. The first approach conducts a conformance check based on log-level event structures [12]. It detects process-level deviations in entire event logs but can, in principle, be used for trace-level feedback, which is equivalent to our five pattern types. In Sect. 5.1, we use it as a benchmark by applying it to each trace individually. Given the different scope, additional patterns are found if considering the entire event log at once. The approach does not consider deviations on process fragments in the five pattern types used by us and it does not account for the same deviation multiple times in one trace. Returned deviations are not mutually exclusive. The second approach discovers sets of highly correlated event-level deviations, which are assumed to constitute a process-level deviation [13]. Such a statistical approach might miss rare process-level deviations, which can have major impact on the organization. Also, as no predefined patterns are used, the user is required to interpret the deviation.

If process-level deviations are previously known and expected to occur, they can be included in a process model. A so-called "break-the-glass" alignment assigns specific costs to these deviations [2] and allows for their detection in process executions. Alternatively, these deviations can also be detected by means of token-replay [5]. However, these approaches are limited to known and explicitly modeled process-level deviations.

Finally, some approaches classify event-level deviations as either skipped, swapped, or duplicated [21, 24]. However, the discovery is restricted to these three pattern types. Also, event-level deviations are not aggregated, but rather each alignment move is classified. Similarly, other approaches classify data-aware alignment moves as "deviation patterns", defined as a combination of deviations in control-flow and data perspective [18]. Thus, they do not discover process-level deviations. Further, an approach to workaround detection classifies activities as repeated, substituted, interchanged, bypassed, or added [23]. Although these are similar to our deviation patterns, there is a conceptual difference between (typically intentional) workarounds and (usually unintended) deviations. Also, only individual activities are classified by means of machine learning, thus not ensuring correct classification and not setting deviations into context.

Model repair. In research on model repair, some approaches base their repairs on deviation patterns. For example, given a process model, one approach detects high-level anomalous behavior in an event log and includes it in a repaired version of the model [14]. However, this anomalous behavior does not use predefined patterns but rather is defined as often co-occurring low-level deviations. Another approach considers the same types of patterns as [12] and allows users to incrementally include them in the model [4]. The identification of these patterns is equivalent to our benchmark. Last, one approach repairs model by assuming the existence of skips and repetitions [20].

7 Conclusion

In this paper, we propose an approach that discovers process-level deviations from event-level insights provided by alignments. Relying on a set of five commonly used patterns, this four-step approach accounts for user's preferences and contextualizes deviations within choices and parallelism in process models. Our evaluation shows that all process-level deviations are discovered in labelled datasets, addressing the two main challenges of this tasks. Also, we reveal process-level deviations in a real-life event log.

Our approach is subject to a few limitations. First, we use trace alignments as the basis for our approach, which are not initially contextualized in the control-flow constructs of a process model. However, since alignments provide a symmetric view of how a trace fits to a process model and can be considered state-of-art for trace level feedback [9], they provide a solid foundation to discover process-level deviations in traces. Therefore, in contrast to the benchmark we compared against [12], our approach requires an additional contextualization step. Second, although we have reduced the non-determinism of both the alignment computation (by adjusting the cost function) and the BP solution (by assigning slightly different penalties for pattern types), we cannot eliminate it completely and therefore cannot prove that our approach always behaves fully deterministically. Still, we consider this risk to be low since this situation did not occur in any of our experiments. Third, we define an optimal interpretation of an alignment based on based on the assumption that maximal process-level deviations are preferred to non-maximal ones. This optimality assumption might not hold in every setting, for example in strongly deviating traces. However, we consider it to be reasonable as it is in line with current research findings [15]. Last, in very rare edges cases, our adjusted cost function cf' will influence not only the move order but also the move choice (while maintaining optimality under cf). For example, consider a process model with a loop on activities X and Y and a trace that only executes X. Under cf', a log move on X will be favored over a model move on Y as we make the log move less expensive. However, since model move (i.e., skipping half of the loop) and log move (i.e., inserting half of the loop) reflect the same insight, this does not reduce the practicality of our approach.

The output of our approach can be used in the future to provide deeper insights into process conformance. To show its practical value, we aim to assess the informativeness of the approach by conducting a user study and evaluating whether the output is useful for managers. Further, we want to research if we can further characterize deviating behavior and, e.g., detect early terminations of traces. In the same context, we aim to study how similarities between process-level deviations and identify dependencies between different deviation patterns such as links (e.g., if A is skipped, Z is also skipped).

References

1. Acitelli, G., Angelini, M., Bonomi, S., Maggi, F.M., Marrella, A., Palma, A.: Context-aware trace alignment with automated planning. In: ICPM. pp. 104–111 (2022)
2. Adriansyah, A., Van Dongen, B.F., Zannone, N.: Controlling break-the-glass through alignment. In: ICSC. pp. 606–611. IEEE (2013)
3. Alizadeh, M., de Leoni, M., Zannone, N.: History-based construction of alignments for conformance checking: Formalization and implementation. In: SIMPDA. pp. 58–78 (2015)

4. Armas Cervantes, A., van Beest, N.R.T.P., La Rosa, M., Dumas, M., García-Bañuelos, L.: Interactive and incremental business process model repair. In: OTM. pp. 53–74. Springer (2017)
5. Banescu, S., Petković, M., Zannone, N.: Measuring privacy compliance using fitness metrics. In: BPM. pp. 114–119. Springer (2012)
6. Berti, A., van Zelst, S.J., van der Aalst, W.: Process mining for python (pm4py): Bridging the gap between process-and data science. ICPM Demos (2019)
7. Bloemen, V., van Zelst, S., van der Aalst, W., van Dongen, B., van de Pol, J.: Aligning observed and modelled behaviour by maximizing synchronous moves and using milestones. Inf Syst **103**, 101456 (2022)
8. Boltenhagen, M., Chatain, T., Carmona, J.: A discounted cost function for fast alignments of business processes. In: BPM. pp. 252–269 (2021)
9. Carmona, J., van Dongen, B., Solti, A., Weidlich, M.: Conformance Checking - Relating Processes and Models. Springer (2018)
10. Depaire, B., Swinnen, J., Jans, M., Vanhoof, K.: A process deviation analysis framework. In: BPM Workshops. pp. 701–706 (2013)
11. Dunzer, S., Stierle, M., Matzner, M., Baier, S.: Conformance checking: A state-of-the-art literature review. In: S-BPM ONE. p. 1–10. ACM (2019)
12. García-Bañuelos, L., Van Beest, N., Dumas, M., La Rosa, M., Mertens, W.: Complete and interpretable conformance checking of business processes. Trans Softw Eng **44**(3), 262–290 (2017)
13. Genga, L., Alizadeh, M., Potena, D., Diamantini, C., Zannone, N.: Discovering anomalous frequent patterns from partially ordered event logs. J Intell Inf Syst **51**, 257–300 (2018)
14. Genga, L., Rossi, F., Diamantini, C., Storti, E., Potena, D.: Model repair supported by frequent anomalous local instance graphs. Inf Syst **122**, 102349 (2024)
15. Hosseinpour, M., Jans, M.: Auditors' categorization of process deviations. J Inf Sys **38**(1), 67–89 (2024)
16. IBM: Cplex optimization studio. Version **12** (1987-2018)
17. Leemans, S.J., Fahland, D., van der Aalst, W.: Discovering block-structured process models from event logs-a constructive approach. In: PETRI NETS. pp. 311–329 (2013)
18. Mozafari Mehr, A., de Carvalho, R., van Dongen, B.: Explainable conformance checking: Understanding patterns of anomalous behavior. Eng Appl Artif Intell **126**, 106827 (2023)
19. Nolle, T., Luettgen, S., Seeliger, A., Mühlhäuser, M.: Binet: Multi-perspective business process anomaly classification. Inf Syst **103**, 101458 (2019)
20. Polyvyanyy, A., Aalst, W.M.V.D., Hofstede, A.H.T., Wynn, M.T.: Impact-driven process model repair. TOSEM **25**(4), 1–60 (2016)
21. Schumann, G., Kruse, F., Nonnenmacher, J.: A practice-oriented, control-flow-based anomaly detection approach for internal process audits. In: Serv Oriented Comput. pp. 533–543 (2020)
22. Vanderbei, R.J.: Linear Programming. Springer (2020)
23. Weinzierl, S., Wolf, V., Pauli, T., Beverungen, D., Matzner, M.: Detecting temporal workarounds in business processes – a deep-learning-based method for analysing event log data. J Bus Ana **5**(1), 76–100 (2022)
24. Yang, S., Sarcevic, A., Farneth, R.A., Chen, S., Ahmed, O.Z., Marsic, I., Burd, R.S.: An approach to automatic process deviation detection in a time-critical clinical process. J Biomed Inform **85**, 155–167 (2018)