Model Repair — Aligning Process Models to Reality

Dirk Fahland, Wil M.P. van der Aalst

Eindhoven University of Technology, The Netherlands

Abstract

Process mining techniques relate observed behavior (i.e., event logs) to modeled behavior (e.g., a BPMN model or a Petri net). Process models can be discovered from event logs and conformance checking techniques can be used to detect and diagnose differences between observed and modeled behavior. Existing process mining techniques can only uncover these differences, but the actual repair of the model is left to the user and is not supported. In this paper we investigate the problem of *repairing a process model w.r.t. a log* such that the resulting model can replay the log (i.e., conforms to it) and is as similar as possible to the original model. To solve the problem, we use an existing conformance checker that aligns the runs of the given process model to the traces in the log. Based on this information, we decompose the log into several sublogs of non-fitting subtraces. For each sublog, either a loop is discovered that can replay the sublog or a subprocess is derived that is then added to the original model at the appropriate location. The approach is implemented in the process mining toolkit ProM and has been validated on logs and models from several Dutch municipalities.

Keywords: process mining, model repair, Petri nets, conformance checking

1. Introduction

Process mining techniques aim to extract non-trivial and useful information from event logs [1, 2]. The process mining spectrum ranges from operational support techniques (predictions and recommendations) to techniques to identify bottlenecks and decision rules [1]. The two main (and best known) types of process mining are (1) process discovery and (2) conformance checking.

Process discovery techniques automatically construct a process model (e.g., a Petri net or a BPMN model) from an event log [1, 3–8]. The basic idea of control-flow discovery is very simple: given an event log containing a collection of traces, automatically construct a suitable process model "describing the behavior" seen in the log. However, given the characteristics of real-life event logs, it is notoriously difficult to learn useful process models from such logs. Event logs

Email addresses: d.fahland@tue.nl (Dirk Fahland), w.m.p.v.d.aalst@tue.nl (Wil M.P. van der Aalst)

only contain example behavior and do not explicitly indicate what is impossible. The fact that an event log does not contain a particular trace does not imply that that trace is impossible. Moreover, a process discovery technique needs to mediate between different concerns, e.g., *fitness* (ability to explain observed behavior), *simplicity* (Occam's Razor), *precision* (avoiding underfitting), and *generalization* (avoiding overfitting).

The second type of process mining is *conformance checking* [4, 9–17]. Here, an existing process model is compared with an event log of the same process. Conformance checking can be used to check if reality, as recorded in the log, conforms to the model and vice versa. The conformance check could yield that the model *does not describe the process executions observed in reality*: activities in the model are skipped in the log, the log contains events not described by the model, or activities are executed in a different order than described by the model.

In case an existing process model does not conform to reality one could — in principle — use process discovery to obtain a model that does. However, the discovered model is likely to bear no similarity with the original model, discarding any value the original model had, in particular if the original was created manually. A typical real-life example is the reference process model of a Dutch municipality shown in Fig. 1(left); when rediscovering the actual process using logs from the municipality one would obtain the model in Fig. 1(right).

Model Repair: between conformance checking and discovery

A more promising approach is to *repair* the original model such that it can replay (most of) the event log while staying close to the reference model (cf. Fig. 1(middle)). In [18], we introduced a new type of process mining: *model repair*. Like conformance checking we use a process model N and an event log L as input. If model N conforms to L (i.e., the observed behavior can be fully explained by the model), then there is no need to change N. However, if parts of N do not conform to L, these parts can repaired using the technique presented in this paper. Unlike discovery, the parts of the model that are not invalidated by the event log are kept as is. The resulting repaired model N' can be seen as a "synergy" of original process model N and event log L.

There are three main use cases for model repair:

- *Improving conformance checking diagnostics.* Conformance checking identifies discrepancies between modeled and observed behavior, e.g., by showing misalignments or places where tokens are missing during replay. However, it is not easy to see what the actual conformance problem is and how to resolve it. By highlighting the repaired parts of the repaired model, one can show discrepancies succinctly. As usual, conformance problems may lead to adaptations of the actual process (e.g., better work instructions or more control) or to changes of the model to reflect reality better. In the latter case the repaired model can be used as the new normative or descriptive model.

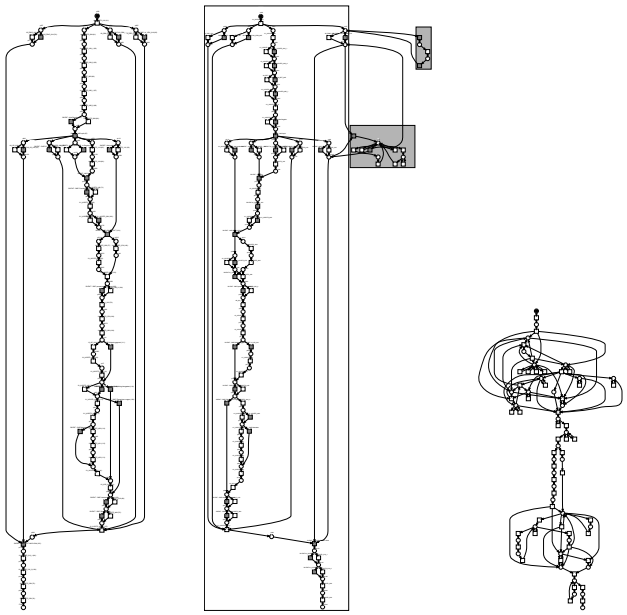


Figure 1: Original model (left), model (middle) obtained by repairing the original model w.r.t. a given log, and model (right) obtained by rediscovering the process without considering the original model. The highlighted parts in the repaired model have been added to better fit the observed behavior.

- *Monitoring process evolution.* Processes may evolve over time, e.g., workers start handling cases differently, informal procedures change, or people adapt to changing external circumstances. This implies that the corresponding process models need to be updated to be of any value. Model repair shows the parts of the old model that no longer fit. These parts can be adapted to reflect the new situation.

- *Supporting customization.* Different process variants and organizational units may share some initial reference model, e.g., a model describing best practices or a modeling template. For example, municipalities may use a standard reference model for processing building permits, but still there will be differences between municipalities (e.g., the moment of payment). Organizations using enterprise software may use reference models provided by the vendor, but their actual processes may deviate from these initial models. Model repair can be used to customize the initial reference model. The resulting model is close to the reference model, but more accurately describes reality for a specific process or organizational unit.

For all three use cases it is important that *the repaired model remains as close to the original model as possible.* For example, when monitoring process evolution one would not like to completely change the process repeatedly if the changes are gradual or local. For customization it is also important to stay as close to the initial reference model as possible (e.g., to more effectively communicate differences).

Since it may be undesirable to modify the process model to accommodate infrequent highly exceptional behavior, one may choose to repair the model only for frequently observed deviating behavior. The influence of the original model on the repaired model may be configurable. Effectively, there are two extremes. One extreme is to avoid changes to the original model as much as possible. The other extreme is to simply discover the process model from the event log with little consideration for the original model. In this paper, we avoid the latter case as much as possible. We try to stay as close to the original model as possible, e.g., to clearly communicate the differences.

Scope: repairing control-flow to fit an event log

In this paper we focus on *repairing control-flow problems* and only briefly discuss repairing other perspectives such as data flow and work distribution. There are two main reasons for this. First of all, the control-flow structure forms the backbone of the process model. Hence, other perspectives are only considered when the control-flow is fixed. Second, it is generally intractable to consider all perspectives at the same time. The choice to first focus on control-flow is common in process mining. Consider for example decision mining approaches that first fix the decision points and only then use classical data mining approaches to describe the choices in terms of e.g. decision trees [19]. Another example is the alignment-based approach to check conformance with respect to data and resources described in [20]. In [20] it is demonstrated that the alignment-based techniques also used in our repair approach can be extended to data and resources. However, already for conformance checking this is computationally challenging. For discovery and repair it seems better to first focus on the control-flow and in a second phase repair the other perspectives. In any case, it is straightforward to the extend the approach in this paper to also cover additional perspectives.

The main criterion for repair used in this paper is *fitness*, i.e., our first concern is to extend the model such that the observed behavior can be explained by the model. There are also other considerations such as avoiding overfitting and underfitting and the desire to produce a simple model. However, these are of secondary concern in this paper. It does not make any sense to reason about a model with poor fitness (as is demonstrated in [21]). If the model is unable to replay most of the events in the log, there is no point in trying to balance overfitting and underfitting. Moreover, the secondary concerns can be addressed in a pre- or post-processing phase. For example, infrequent paths can be removed to increase precision and simplicity.

Problem definition and approach

The concrete problem addressed in this paper reads as follows. We assume a Petri net N (a model of a process) and a log L (being a multiset of observed cases of that process) to be given. N conforms to L if N can execute each case in L, i.e., N can replay L. If N cannot replay L, then we have to *change* N to a Petri net N' such that N' can replay L and N' is as similar to N as possible.

We solve the repair problem in a compositional way: we identify subprocesses that have to be added in order to repair N. In more detail, we first compute

for each case $l \in L$ an *alignment* that describes at which parts N and l deviate. Based on this alignment, we identify transitions of N that have to be skipped to replay l and which particular events of l could not be replayed on N. Moreover, we identify the *location* at which N should have had a transition to replay each of these events. We group sequences of non-replayable events at the same location to a *sublog* L' of L. For each sublog L', we construct a small subprocess N' that can replay L' by using a process discovery algorithm. We then insert N' in N at the location where each trace of L' should have occurred. By doing this for every sublog of non-replayable events, we obtain a repaired model that can replay L. Moreover, by the way we repair N, we preserve the structure of N giving process stakeholders useful insights into the way the process changed. We observed in experiments that even in case of significant deviations we could identify relatively few and reasonably structured subprocesses: adding these to the original model always required fewer changes to the original model than a complete rediscovery. Repairing the model of Fig. 1(left) in this way yields the model shown in Fig. 1(middle).

Extending an earlier version of this article [18], we investigate several options for improving the above method of model repair. In particular, we show how choosing the "right" alignment of L to N influences repairs of N in a favorable way. We show how to decompose and align sublogs on non-fitting traces such that subprocesses added to N have a simple structure. We present heuristics for improved placement of subprocesses in N, and introduce a technique to identify loops that can be added to N instead of a subprocess. Also, we highlight techniques for preprocessing logs to improve the quality of the repaired model. The introduced techniques for model repair are evaluated experimentally and compared to a high-quality manual repair. We also investigate limitations of repeatedly repairing a model with respect to several logs.

The remainder of this paper is structured as follows. Section 2 recalls basic notions on logs, Petri nets and alignments. Section 3 investigates the model repair problem in more detail. Section 4 presents a solution to model repair based on subprocesses. Section 5 introduces a number of improvements on this basic model repair technique. We report on experimental results in Sect. 6 and discuss related work in Sect. 7. Section 8 concludes the paper.

2. Preliminaries

This section recalls the basic notions on Petri nets and introduces notions such as event logs and alignments.

2.1. Event Logs

Event logs serve as the starting point for process mining. A process model describes the life-cycle of *cases* of a particular type, e.g., insurance claims, customer orders, or patient treatments. Hence, each event refers to a case. Moreover, each event refers to some activity and all events corresponding to a particular case are ordered. In other words: each case is described by a

sequence of events. Next to its activity name, an event may have many other attributes, such a timestamp, the resource(s) involved, the transaction type (start, complete, abort, etc.), associated costs, etc. For example, in 2010 the IEEE Task Force on Process Mining standardized XES (`www.xes-standard.org`), a standard logging format that is extensible and supported by the OpenXES library (`www.openxes.org`) and by tools such as ProM, XESame, Disco, etc. XES allows for standard attributes such as timestamp, but also allows for the addition of any number of event attributes.

In this paper, we focus on control-flow and assume that an event is represented by an *action*. As a result, a *trace* is a sequence of actions, an event log can be defined as a multiset of traces. Each trace describes the life-cycle of a particular *case* (i.e., a *process instance*) in terms of the *activities* executed.

Definition 1 (Trace, Event Log)**.** Let Σ be a set of actions. A trace $l \in \Sigma^*$ is a sequence of actions. $L \in \mathbb{B}(\Sigma^*)$ is an event log, i.e., a multiset of traces.

An event log is a *multiset* of traces because there can be multiple cases having the same trace. If the frequency of traces is irrelevant, we refer to a log as a set of traces $L = \{l_1, \ldots, l_n\}$. In this simple definition of an event log, an event is fully described by an action and we cannot distinguish two cases having the same trace.

Additional attributes may be incorporated in the action, e.g., action $decide(goldcustomer, John, reject)$ may refer to a decision to reject a gold customer made by John. Action $decide(goldcustomer, Mary, reject)$ refers to the same decision made by Mary. Adding attributes to the action label results in a large number of different actions. Hence, process mining becomes difficult. Whereas conformance checking is still possible [20] discovery and repair quickly become intractable. Therefore, we propose a two-stage approach where first the control-flow is discovered or repaired. After fixing the control-flow one can consider the other attributes as shown in [19] and [22]. Note that the concept of alignments described in Section 2.4 can be extended to accommodate additional attributes and process models can be extended with decision rules and resource allocation rules. However, for simplicity we abstract from these perspectives.

2.2. Petri Nets

We use *labeled* Petri nets to describe processes. We first introduce unlabeled nets and then lift these notions to their labeled variant.

Definition 2 (Petri net)**.** A Petri net (P, T, F) consists of a set P of *places*, a set T of *transitions* disjoint from P, and a set of arcs $F \subseteq (P \times T) \cup (T \times P)$. A *marking* m of N assigns each place $p \in P$ a natural number $m(p)$ of *tokens*. A *net system* $N = (P, T, F, m_0, m_f)$ is a Petri net (P, T, F) with an *initial* marking m_0 and a final marking m_f.

We write ${}^\bullet y := \{x \mid (x, y) \in F\}$ and $y^\bullet := \{x \mid (y, x) \in F\}$ for the *pre-* and the *post-set* of y, respectively. Fig. 2

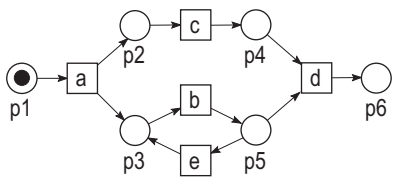


Figure 2: A net system N.

shows a simple net system N with the initial marking $[p_1]$ and final marking $[p_6]$. N will serve as our running example.

The semantics of a net system N are typically given by a set of *sequential runs*. A transition t of N is *enabled* at a marking m of N iff $m(p) \geq 1$, for all $p \in {}^\bullet t$. If t is enabled at m, then t may *occur* in the step $m \xrightarrow{t} m_t$ of N that reaches the *successor marking* m_t with $m_t(p) = m(p) - 1$ if $p \in {}^\bullet t \setminus t^\bullet$, $m_t(p) = m(p) + 1$ if $p \in t^\bullet \setminus {}^\bullet t$, and $m_t(p) = m(p)$ otherwise, for each place p of N. A sequential run of N is a sequence $m_0 \xrightarrow{t_1} m_1 \xrightarrow{t_2} m_2 \ldots \xrightarrow{t_k} m_f$ of steps $m_i \xrightarrow{t_{i+1}} m_{i+1}, i = 0, 1, 2, \ldots$ of N beginning in the initial marking m_0 and ending in the final marking m_f of N. The sequence $\sigma = t_1 t_2 \ldots t_k$ is an *occurrence sequence* of N. For example, in the net N of Fig. 2 transition a is enabled at the initial marking; abcd is a possible occurrence sequence of N.

The transitions of a Petri net can be *labeled* with names from an alphabet Σ. In particular, we assume label $\tau \in \Sigma$ denoting an *invisible* action. A *labeled Petri net* (P, T, F, ℓ) is a net (P, T, F) with a *labeling* function $\ell : T \to \Sigma$. A *labeled net system* $N = (P, T, F, \ell, m_0, m_f)$ is a labeled net (P, T, F, ℓ) with initial marking m_0 and final marking m_f. The semantics of a labeled net are the same as for an unlabeled net. Additionally, we can consider *labeled* occurrence sequences of N. Each occurrence sequence $\sigma = t_1 t_2 t_3 \ldots$ of N induces the *labeled* occurrence sequence $\ell(\sigma) = \ell(t_1)\ell(t_2)\ell(t_3)\ldots\ell(t_k)|_{\Sigma\setminus\{\tau\}}$ obtained by replacing each transition t_i by its label $\ell(t_i)$ and omitting all τ's from the result by projection onto $\Sigma \setminus \{\tau\}$. We say that N *can replay* a log L iff each $l \in L$ is a labeled occurrence sequence of N.

2.3. Relating Event Logs and Process Models

Our repair approach assumes that the event log and the labeled Petri net refer to a common set of actions Σ. Ideally, event log and process model refer to a common ontology. In reality this is often not the case. Besides *differences in naming*, there may be *differences in granularity and coverage*.

The event log may contain information about low level events that do not correspond to activities in the model. For example, the trace segmentation approach described in [23] aims to group low-level events into clusters, which represent the execution of a higher-level activity in the process model. By projecting these clusters onto the higher-level activities, the abstraction level can be lifted. In [24] the same problem is addressed by mining for common low-level repeating patterns that are replaced by higher level activities. The latter approach made it possible to successfully apply process mining to the event logs of Philips Healthcare's X-ray machines. In [25] a similar approach was applied to the low-level event logs of ASML's wafer steppers.

Whereas differences in granularity are difficult to tackle, differences in coverage are relatively easy. Events in the event log that do not correspond to any

activity can be removed from the event log. Activities in the model that are unobservable can be labeled as τ actions [1].

In the following we assume that event log and process model have been preprocessed and refer to a common set of actions Σ; though log and model may use just a subset of Σ.

2.4. Aligning an Event log to a Process Model

Conformance checking techniques investigate how well an event log $L \in \mathbb{B}(\Sigma^*)$ and a labeled net system $N = (P, T, F, \ell, m_0, m_f)$ fit together. The process model N may have been discovered through process mining or may have been made by hand. In any case, it is interesting to compare the observed example behavior in L and the potential behavior of N. In case the behavior in L is not possible according to N (N cannot replay L), we want to repair N.

In the following we recall a state-of-the-art technique in conformance checking [9–11]. We use the concept of *alignments* for identifying where L and N deviate, and hence where N has to be repaired. These alignments will allow us to determine a *minimal set of changes* that are needed to replay L on N. It essentially boils down to relating $l \in L$ to an occurrence sequence σ of N such that l and σ are as similar as possible. When putting l and σ next to each other, i.e., *aligning* σ and l, we will find (1) at which point a particular activity of N should have occurred but did not according to l and (2) at which point a particular activity of l occurred, but was not described by N [9].

Alignments are very different from well-known edit distances such as the Levenshtein distance [26]. First of all, we are aligning a trace and a model and not two traces or sets of traces. As a model has usually infinitely many traces (in case of loops) the least deviating trace cannot be found by enumerating all traces and computing their edit distance. Second, we would like to assign arbitrary costs to the different types of deviations. And finally, we are not just interested in the distance between trace and model, but in their *exact deviations*. Therefore, we turn the problem into an optimization problem. For applications of the edit distances to process mining, we refer to [27].

To explain the alignment concept consider a trace $l = \mathsf{accd}$ which is similar to the occurrence sequence $\sigma = \mathsf{abcd}$ of the net of Fig. 2 where trace l deviates from σ by skipping over b and having an additional c. An *alignment* shows a possible correspondence between trace l and occurrence sequence σ:

a	c	$\gg$	c	d
a	c	b	$\gg$	d

. The projection onto the first row (ignoring $\gg$) yields trace l. The projection onto the second row (ignoring $\gg$) yields trace σ. The $\gg$ in the first row corresponds to the skipping of b in the log and the $\gg$ in the second row corresponds to the skipping of the second c in the model. Obviously, such an alignment provides input for repairing the event log.

In [9–11] an approach was presented that allows to automatically align a trace l to an occurrence sequence of N with a minimal number of deviations (i.e., $\gg$ insertions) in an efficient way. All of this is based on the notion of an alignment and a cost function.

Definition 3 (Alignment)**.** Let $N = (P, T, F, \ell, m_0)$ be a labeled net system. Let $l = a_1 a_2 \dots a_m$ be a trace over Σ. A *move* is a pair $(b, s) \in (\Sigma \cup \{\gg\}) \times (T \cup \{\gg\}) \setminus \{(\gg, \gg)\}$. An *alignment* of l to N is a sequence $\alpha = (b_1, s_1)(b_2, s_2) \dots (b_k, s_k)$ of moves, such that

1. the restriction of the first component to actions Σ is the trace l, i.e., $(b_1 b_2 \dots b_k)|_\Sigma = l$,
2. the restriction of the second component to transitions T, $(s_1 s_2 \dots s_k)|_T$, is an occurrence sequence of N, and
3. transition labels and actions coincide (whenever both are defined), i.e., for all $i = 1, \dots, k$, if $s_i \neq \gg, \ell(s_i) \neq \tau$, and $b_i \neq \gg$, then $\ell(s_i) = b_i$.

Move (b_i, s_i) is called (1) a *move on model* iff $b_i = \gg \wedge s_i \neq \gg$, (2) a *move on log* iff $b_i \neq \gg \wedge s_i = \gg$, and (3) a *synchronous move* iff $b_i \neq \gg \wedge s_i \neq \gg$.

Note that in Fig. 2 transition names and transition labels coincide, i.e., $T = \Sigma$ and $\ell(t) = t$ for $t \in T$. However, the definition allows for multiple transitions having the same label and invisible transitions having a τ label.

For instance, for trace $l = \mathsf{accd}$ and the net of Fig. 2, a possible alignment would be $(\mathsf{a}, \mathsf{a})(\mathsf{c}, \mathsf{c})(\gg, \mathsf{b})(\mathsf{c}, \gg)(\mathsf{d}, \mathsf{d})$, also denoted as: $\begin{array}{|c|c|c|c|c|}\hline \mathsf{a} & \mathsf{c} & \gg & \mathsf{c} & \mathsf{d} \\ \hline \mathsf{a} & \mathsf{c} & \mathsf{b} & \gg & \mathsf{d} \\ \hline \end{array}$.

Each trace usually has several (possibly infinitely many) alignments to N. For instance, possible alignments for trace $l = \mathsf{abcbd}$ and the net of Fig. 2 are:
$\alpha_1 = \begin{array}{|c|c|c|c|c|}\hline \mathsf{a} & \mathsf{b} & \mathsf{c} & \mathsf{b} & \mathsf{d} \\ \hline \mathsf{a} & \mathsf{b} & \mathsf{c} & \gg & \mathsf{d} \\ \hline \end{array}$, $\alpha_2 = \begin{array}{|c|c|c|c|c|c|}\hline \mathsf{a} & \mathsf{b} & \mathsf{c} & \gg & \mathsf{b} & \mathsf{d} \\ \hline \mathsf{a} & \mathsf{b} & \mathsf{c} & \mathsf{e} & \mathsf{b} & \mathsf{d} \\ \hline \end{array}$, and
$\alpha_3 = \begin{array}{|c|c|c|c|c|c|c|}\hline \mathsf{a} & \mathsf{b} & \mathsf{c} & \mathsf{b} & \gg & \gg & \mathsf{d} \\ \hline \mathsf{a} & \gg & \gg & \gg & \mathsf{c} & \mathsf{b} & \mathsf{d} \\ \hline \end{array}$.

We are typically interested in a best alignment, i.e., one that has as few non-synchronous moves (move on model or move on log) as possible. One way to find a best alignment is to use a cost function on moves, and to find an alignment with the least total cost.

Definition 4 (Cost function, cost of an alignment)**.** Let $\kappa : \Sigma \cup T \to \mathbb{N}$ define for each transition and each action a non-negative cost: $\kappa(x) \geq 0$ for all $x \in \Sigma \cup T$. The costs of an invisible action is set to zero: $\kappa(x) = 0$ if $x \in T$ and $\ell(x) = \tau$. The *cost* of a move (b, s) is $\kappa(b, s)$ and is defined as follows: $\kappa(b, s) = 0$ iff $b \neq \gg \neq s$ (synchronous move), $\kappa(b, s) = \kappa(s)$ iff $b = \gg$ (move on model), and $\kappa(b, s) = \kappa(b)$ iff $s = \gg$ (move on log). The cost of an alignment $\alpha = (b_1, s_1) \dots (b_k, s_k)$ is $\kappa(\alpha) = \sum_{i=1}^{k} \kappa(b_i, s_i)$.

The cost for actions (in Σ) and for visible transitions (in N) can be chosen freely; the specific choice of costs depends on the use case, as we discuss later. Costs for actions and transitions raise the cost of an alignment for every move on log or move on model. By minimizing the costs of an alignment, we avoid moves on log and moves on model in favor of synchronous moves.

Consider for example above alignments α_1, α_2, and α_3 for trace $l = \mathsf{abcbd}$ and the net of Fig. 2, and a standard cost function κ assigning unit costs to all undesirable moves, i.e., $\kappa(x) = 1$ for all $x \in \Sigma \cup T$. $\kappa(\alpha_1) = 1$, $\kappa(\alpha_2) = 1$, and $\kappa(\alpha_3) = 5$. Hence, α_1 and α_2 are clearly better alignments than α_3.

Definition 5 (Best alignment)**.** Let $N = (P, T, F, \ell, m_0)$ be a labeled net system. Let κ be a cost function over moves of N and Σ. Let l be a trace over Σ. An alignment α (of l to N) is a *best* alignment (wrt. κ) iff for all alignments α' (of l to N) holds $\kappa(\alpha') \geq \kappa(\alpha)$.

Note that a trace l can have several best alignments with the same cost (cf. α_1 and α_2). A best alignment α of a trace l can be found efficiently using an A*-based search over the space of all prefixes of all alignments of l. The cost function κ thereby serves as a very efficient heuristics to prune the search space and guide the search to a best alignment.

1. The default cost function is *uniform* and assigns each deviation cost 1; the corresponding best alignment has the least number of deviations.
2. A non-uniform cost function allows to compute alignments with specific properties. For instance, one can set costs based on how probable a particular deviation is (low probability implies high costs). Then a log trace is aligned to the most probable model trace showing the most probable deviations.
3. Generally, the higher the costs of a move, the more likely it is avoided in an alignment and replaced by other moves. This way, one can reveal a different set of deviations that explain the difference between log and model.

In Sect. 5, we explore cost functions which yield more favorable model repairs than a uniform cost function. More technical details on the cost function are given in [10, 11].

Using the notion of best alignment we can relate any trace $l \in L$ to an occurrence sequence of N. Hence, in the remainder, we can assume to have an "oracle" that maps both fitting and non-fitting cases onto paths and states in the model.

3. Model Repair: The Problem

The model repair problem is to transform a model N that does not conform to a log L into a model N' that conforms to L but is as close to N as possible. We review the state-of-the-art in conformance checking and investigate the model repair problem in more detail.

3.1. Conformance of a Process Model to a Log: Problem Dimensions

To repair a model, one first needs to diagnose the mismatches between model N and log L. To understand what needs to be repaired we use the state-of-the-art in conformance checking [9]. Conformance checking techniques can be used to point out differences between modeled and observed behavior, and thus point out parts that need to be repaired. Therefore, we briefly review conformance checking literature.

Conformance checking can be done for various reasons. First of all, it may be used to audit processes to see whether reality conforms to some normative

or descriptive model. Deviations may point to fraud, inefficiencies, and poorly designed or outdated procedures. Second, in process evolution or process customization conformance checking helps detecting differences between an outdated model or a reference model and reality. Finally, conformance checking can be used to evaluate the results of process discovery techniques. In fact, genetic process mining algorithms use conformance checking to select the candidate models used to create the next generation of models [5].

Numerous conformance measures have been developed in the past [4, 9–17]. These can be categorized into four quality dimensions for comparing model and log: (1) *fitness*, (2) *simplicity*, (3) *precision*, and (4) *generalization* [1]. A model with good *fitness* allows for most of the behavior seen in the event log. A model has a perfect fitness if all traces in the log can be replayed by the model from beginning to end. The *simplest* model that can explain the behavior seen in the log is the best model. This principle is known as Occam's Razor. A model is *precise* if it is not "underfitting", i.e., the model does not allow for "too much" behavior. A model is *general* if it is not "overfitting", i.e., the model is likely to be able to explain unseen cases [1, 9].

The fitness of a model N to a log L can be computed using the alignments of Sect. 2.4. For example, fitness can be defined as the fraction of moves on log or moves on model relative to all moves [9]. The aligned event log can also be used as a starting point to compute other conformance metrics such as precision and generalization [9, 28]. However, as discussed earlier, the focus will be on fitness.

3.2. Repairing a Process Model to Conform to a Log: Guiding Forces

Although there are many approaches to compute conformance and to diagnose deviations given a log L and model N, we are not aware of techniques to repair model N to conform to log L.

There are two "forces" guiding such repair. First of all, there is the need to improve conformance. Second, there is the desire to clearly relate the repaired model to the original model, i.e., repaired model and original model should be similar. Given metrics for conformance and closeness of models, we can measure the weighted sum or harmonic mean of both metrics to judge the quality of a repaired model. If the first force is weak (i.e., minimizing the distance is more important than improving the conformance), then the repaired model may remain unchanged. If the second force is weak (i.e., improving the conformance is more important than minimizing the distance), then repair can be seen as plain process discovery. In the latter case, the initial model is irrelevant and it is better to use conventional discovery techniques.

Figure 3 illustrates the trade-off between both forces thus defining the *repair spectrum*. On the left-hand-side of the spectrum, the focus is on keeping the original model even if there are conformance problems. On the right-hand-side of the spectrum, the focus is on creating a model that fits the log best even if the resulting model is very different from the original one. Typically, model repair is applied in settings in-between these two extremes. For example, only those parts of the original model that are clearly problematic are repaired. Alternately, one

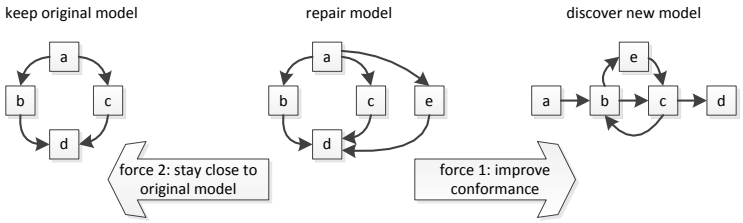


Figure 3: The repair spectrum: balancing between two forces.

could make the minimal set of changes needed to reach a predefined conformance level (e.g., 95% of all cases can be replayed).

The trade-off shown in Fig. 3 creates a major challenge: How to identify which parts of a model shall be kept, and which parts of a model shall be considered as non-conformant to the log and hence changed, preferably automatically? The latter is a *local* process discovery problem which requires balancing the four quality dimensions of conformance as well.

3.3. Addressing Different Quality Dimensions: Trade-Offs

In this paper, we primarily focus on fitness, which is often seen as the most important quality dimension for process models. A model that does not fit a given log (i.e., the observed behavior cannot be explained by the model) is repaired using the information available in the alignments.

It does not make much sense to consider other quality dimensions [1, 2, 9] (i.e., precision, generalization, and simplicity) when the model has poor fitness. See [21] for experimental results showing that good fitness is a primary concern before addressing secondary concerns such as precision, generalization, and simplicity. For example, in [28] it is shown that only for a fitting model precision [14] can be measured adequately.

Our technique for model repair will cater for fitness, and address precision as a side effect. Simplicity is indirectly taken into account as we remain as close to the original model as possible. Moreover, generalization and precision can be balanced, for instance using a post-processing technique such as the one presented in [29].

For model repair basically the same experiences apply as for classical process discovery: while repairing, one should not be forced to extend the model to allow for all observed infrequent behavior — it could result in overly complicated, spaghetti-like models. Therefore, we propose the following approach.

1. Given a log L and model N, determine the multiset L_f of fitting traces and the multiset L_n of non-fitting traces in L.
2. Split the multiset of non-fitting traces L_n into L_d and L_o. Traces L_d are considered as deviating and the model needs to be repaired to address these. Traces L_o are considered as outliers and do not trigger repair actions.
3. Repair model N based on the multiset $L' = L_f \cup L_d$ of traces. L' should perfectly fit the repaired model N', but there may be many candidate models N'.

4. Return a repaired model N' that can be easily related back to the original model N, and in which changed parts are structurally simple.

3.4. Filtering The Event Log Before Repair: Three Approaches

Before replaying the event log on the model, we remove all cases that are not meeting basic quality constraints. For example, there may be cases where clearly the initial or final part of the trace is missing. This may be caused by the fact that these cases have not yet finished or were already running before the recording of events started. Other signs of corrupted event data are missing timestamps or non-monotonously increasing timestamps. When the event log contains transactional information (e.g., start and complete events), one can also see other log-related problems (e.g., activities that were started but never completed). Note that even when we consider just activity names in Σ for conformance, the other attributes still provide useful information indicating whether a trace should be considered for repair or not.

Besides removing corrupted traces, one also needs to carefully consider the set of actions appearing in the event log L but not in the model N: $A = \{a \in l \mid l \in L\} \setminus \{\ell(t) \mid t \in T\}$. It may be that there are very rare events or events that correspond to uninteresting actions deliberately not included in the model. These events should be removed from L before replaying the event log on the model.

After replaying the event log, we obtain the multiset of non-fitting traces L_n The step of separating this multiset into the multiset of deviating traces L_d and the multiset of outlier traces L_o is critical for repair. Figure 3 already illustrated the trade-off between minimizing the distance to the original model and ensuring conformance. For each deviation revealed by the alignments of log and model, we *need to decide whether the model is "wrong" and the log is "right" or the model is "right" and the log is "wrong"*. To support this step we propose three approaches: *frequency-based filtering*, *trace clustering*, and *manual inspection using trace alignment*.

For *frequency-based filtering* we can look (1) at the frequencies of non-conforming traces, (2) at the frequencies of particular deviations, and (3) at the frequencies of deviations in a particular state.

If most traces are frequent and there are just a few traces that are infrequent, e.g., traces that are unique, then one may consider removing the infrequent ones as their effect on conformance is minimal.

By computing alignments, we can replay any case in the model even if it is deviating. The $\gg$ symbols in the alignment show where problems occur. If a problem is very infrequent, the log rather than the model may be repaired. For example, if there are only few b moves on log $(b, \gg)$, one may repair this by removing the corresponding b's from the log. If there are only few s moves on model $(\gg, s)$, one may repair this by adding the corresponding moves to the log. Note that by doing this L_d does not only contain a subset of the original non-fitting traces, but also (partly) modified traces.

One can also consider the states in which according to the alignment there is a problem. Later, we will show that it is easy to identify the severity of problems

in the different states. This information can be used to discard particular deviations or not. Things that happen infrequently have a low impact on the overall conformance; therefore, one may choose to discard these.

Another approach is to apply *trace clustering* to the multiset of non-fitting traces L_n [6, 7, 30]. This results in more homogenous groups of traces. Per group one can decide to add the traces to L_d or L_o. Particularly interesting is the approach in [7] which creates a "rest cluster" of possible outliers.

Despite these automated approaches, one often needs to use domain knowledge to further split the multiset of non-fitting traces L_n into L_d and L_o. Fortunately, this is facilitated by the *trace alignment* technique presented in [31]. The main idea of trace alignment is to align traces for visual inspection showing events that occur out of order. See [32] for a case study using trace alignment to identify outliers.

All of the techniques mentioned are (partially) supported by existing plug-ins in ProM. The plugin "Filter Log Using Simple Heuristics" allows to detect and filter incomplete cases and outlier events; it was used in the case study of Sect. 6. More advanced filtering techniques are available in plug-ins such as "ActiTrac", "Trace Alignment (with Guide Tree)", "Filter Out Unmapped Event Classes", "Construct Log From Alignment", and "Align Log to Model".

In the remainder, we assume L' to be given, i.e., outliers L_o of L are removed. If an event log is noisy and one includes also undesired traces L_o, it makes no sense to repair the model while enforcing a perfect fit as the resulting model will be spaghetti-like and not similar to the original model.

4. Repairing Processes by Adding Subprocesses

In the following, we present a solution to model repair. We first sketch a naïve approach which completely repairs a model w.r.t. the quality dimension of *fitness* but scores poorly in terms of *precision*. We then define a more advanced approach that also caters for precision. Section 5 presents more refined repair techniques that address *simplicity* and improve *similarity* to the original model.

4.1. Naive Solution to Model Repair – Fitness

Alignments give rise to a naive solution to the model repair problem that we sketch in the following. It basically comprises to extend N with a τ-transition that skips over a transition t whenever there is a move on model $(\gg, t)$, and to extend N with a self-looping transition t with label a whenever there is a move on log $(a, \gg)$. This extension has to be done for all traces and all moves on log/model. The crucial part is to identify the locations of these extensions.

Figure 4 illustrates how the non-fitting log $L = \{\mathsf{acfced}, \mathsf{abccfed}\}$ aligns to the net N of Fig. 2. The nets below each alignment illustrate the differences between log L of Fig. 4 and net N of Fig. 2. After replaying ac, the net is in marking $[\mathsf{p4}, \mathsf{p3}]$ and the log requires to replay f which is not enabled in the net. Thus a log move $(\mathsf{f}, \gg)$ is added. Similarly, c is not enabled at this marking and log move $(\mathsf{c}, \gg)$ is added. Then e should occur, which requires to move

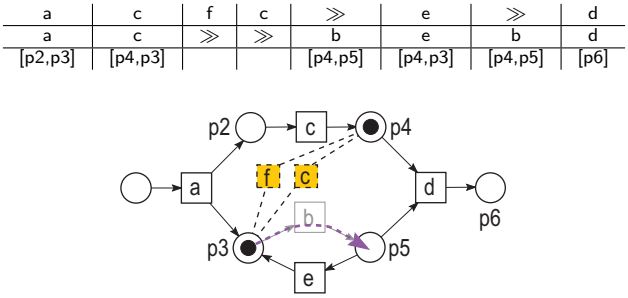


Figure 4: Alignments of log $L = \{\mathsf{acfced}, \mathsf{abccfed}\}$ to the net of Fig. 2.

the token from p3 to p5, i.e., a model move $(\gg, \mathsf{b})$. Correspondingly, the rest of the alignment, and the second alignment is computed. The third line of the alignment describes the marking that is reached in N by replaying this prefix of the alignment on N.

Using this information, the extension w.r.t. a move on model $(\gg, t)$ is trivial: we just have to create a new τ-labeled transition t^* that has the same pre- and post-places as t. For a log move $(a, \gg)$ the alignment tells in which marking m of N action a should have occurred (the "enabling location" of this move). In principle, adding an a-labeled transition t_a that consumes from the marked places of m and puts the tokens back immediately, repairs N w.r.t. to this move on log. However, we improve the placement of t_a by checking if two moves on log $(a, \gg)$ would overlap in their enabling locations. If this is the case, we only add one a-labeled transition that consumes from and produces on this overlap only.

Figure 5(left) shows how model N of Fig. 2 would be repaired w.r.t. the alignment of Fig. 4. The move on model $(\gg, \mathsf{b})$ requires to repair N by adding a τ transition that allows to skip b as shown in Fig. 5. The move on log $(\mathsf{c}, \gg)$ occurs at two different locations $\{\mathsf{p4}, \mathsf{p3}\}$ and $\{\mathsf{p4}, \mathsf{p5}\}$ in the different traces. They overlap on p4. Thus, we repair N w.r.t. $(\mathsf{c}, \gg)$ by adding a c-labeled transition that consumes from and produces on p4. Correspondingly for $(\mathsf{f}, \gg)$. The extended model that is shown in Fig. 5(left) can replay log L of Fig. 4 without any problems.

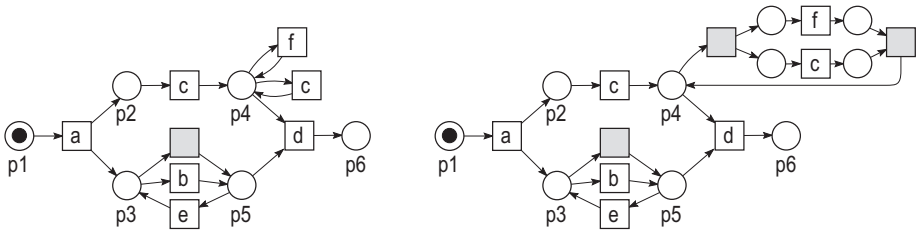


Figure 5: Result of repairing the net of Fig. 2 w.r.t. the log of Fig. 5 by the naive approach (left) and by adding subprocess (right).

4.2. Identify Subprocesses – Precision

The downside of the naive solution to model repair is that the repaired model has low precision. For a log L where a best alignment contains only few synchronous moves, i.e., N does not conform to L, many τ-transitions and self-loops are added. We observed in experiments that often many self-loops were added to N at the same location. In such a case, the resulting model locally permits arbitrary sequences of previously non-replayable events, making the model less precise w.r.t. log L. In the following, we turn this observation into a structured approach to model repair that addresses precision and fitness together.

Instead of just recording for individual actions $a \in \Sigma$ their enabling locations w.r.t. log moves, we now record enabling locations of *sequences of log moves*. Each maximal sequence of log moves (of the same alignment) that all occur at the same location is a *non-fitting subtrace*. We group non-fitting subtraces at the same location Q into a non-fitting *sublog* S at that location. We then *discover* from S a *subprocess* N_S that can replay S by using a mining algorithm that guarantees perfect fitness of N_S to S. We ensure that N_S has a unique start transition and a unique end transition. We then add subprocess N_S to N and let the start transition of N_S consumes from Q and let the end transition of N_S produce on Q, i.e., the subprocess models a structured loop that starts and ends at Q.[1]

Figure 5(right) illustrates this idea. The model depicted is the result of repairing N of Fig. 2 by adding subprocesses as described by the alignments of Fig. 4. We can identify two subtraces cf and fc that occur at the same sublocation $\mathsf{p4}$. Applying process discovery on the sublog $\{\mathsf{cf}, \mathsf{fc}\}$ yields the subprocess at the top right of Fig. 5(right) that puts c and f in parallel. The two grey-shaded silent transitions indicate the start and end of this subprocess.

4.3. Formal Definitions

The formal definitions read as follows. For the remainder of this paper, let N be a Petri net system, let L be a log. For each trace $l \in L$, assume an arbitrary

[1]The term *subprocess* is here used in the sense of a part of the process that has a unique entry point and a unique exit point. It should be read as a plain extension of the existing process model, and not be confused with the idea of factoring out process parts into another document or organizational entity.

but fixed best fitting alignment $\alpha(l)$ to be given. Let $\alpha(L) = \{\alpha(l) \mid l \in L\}$ be the alignments of the traces in L to N.

Whenever an alignment $\alpha(l)$ has a log move $(a_i, \gg)$, the net N is in a particular marking m_i, which we call the *location* of the move.

Definition 6 (Location of a log move)**.** Let $\alpha = (a_1, t_1) \dots (a_n, t_n)$ be an alignment w.r.t. $N = (P, T, F, m_0, m_f, \ell)$. For any move (a_i, t_i), let m_i be the marking of N that is reached by the occurrence sequence $t_1 \dots t_{i-1}|_T$ of N. For all $1 \leq i \leq n$, if $(a_i, t_i) = (a_i, \gg)$ is a log move, then the *location* of $(a_i, \gg)$ is the set $loc(a_i, \gg) = \{p \in P \mid m_i(p) > 0\}$ of places that are marked in m_i.

For example in Fig. 4, $loc(\mathsf{c}, \gg) = \{\mathsf{p4}, \mathsf{p3}\}$ in the first alignment and $loc(\mathsf{c}, \gg) = \{\mathsf{p4}, \mathsf{p5}\}$ in the second alignment.

Any two consecutive log moves have the same location m as the marking m of N does not change in a log move. We group these consecutive moves into a *subtrace* at location m.

Definition 7 (Subtrace)**.** A *subtrace* (β, Q) is a *maximal* sequence $\beta = (a_i, \gg) \dots (a_{i+k}, \gg)$ of consecutive log moves of α having the same location Q, i.e., $loc(a_j, \gg) = loc(a_i, \gg) = Q, i \leq j \leq i + k$, and no longer sequence of log moves has this property.

We write $\beta(L)$ for the set of all subtraces of all alignments $\alpha(L)$ of L to N. To simplify notation, we write $\beta = (\beta, Q)$ and $loc(\beta) = Q$ when no confusion can arise.

For example, in Fig. 5, fc is a subtrace of the first alignment at location $\{\mathsf{p4}, \mathsf{p3}\}$ and cf is a subtrace of the second alignment at location $\{\mathsf{p4}, \mathsf{p5}\}$. We could repair the net by adding two subprocesses, one that can replay fc at $Q_1 = \{\mathsf{p4}, \mathsf{p3}\}$ and one that can replay cf at $Q_2 = \{\mathsf{p4}, \mathsf{p5}\}$. However, we could instead just add one subprocess that can replay fc *and* cf at location $Q_1 \cap Q_2 = \{\mathsf{p4}\}$.

This observation gives rise to two notions. A *sublocation* of a subtrace β is a subset of its location $loc(\beta)$. A *sublog* is a set of subtraces; the location of the sublog is the sublocation shared by all traces in the sublog, or any subset of it.

Definition 8 (Sublog)**.** Let $\alpha(L)$ be an alignment. A non-empty set $S \subseteq \beta(L)$ of subtraces together with a non-empty location $Q \subseteq \bigcap_{\beta \in S} loc(\beta)$ is a *sublog* (S, Q) of $\alpha(L)$.

Each sublog will yield a subprocess that is added to N. The way these sublogs are organized will influence the quality of the repaired model. To ensure fitness of the repaired model, each subtrace has to be in some sublog. A set $\mathcal{S} = \{(S_1, Q_1), \dots, (S_n, Q_n)\}$ of sublogs is *complete* (w.r.t. $\beta(L)$) iff $S_1 \cup \dots \cup S_n = \beta(L)$. A complete set $\mathcal{S}$ can, for instance, be constructed by putting any two subtraces at the same location into the same sublog. We will see in Sect. 5.4 that one can use more refined methods for constructing $\mathcal{S}$ that also addresses the simplicity of the repaired model.

We now have all notions to formally define how to repair model N w.r.t. log L. For each sublog (S, Q) discover a process model N_S and connect it to the location Q in N.

Definition 9 (Subprocess of a sublog)**.** Let L be a log, let N be a Petri net, let $\alpha(L)$ be an alignment of L to N, and let (S, Q) be a sublog of $\alpha(L)$.

Let $S^+ = \{start\ a_1 \dots a_k\ end \mid (a_1, \gg) \dots (a_k, \gg) \in S\}$ be the sequences of events described in S extended by a *start* event and an *end* event ($start, end \notin \Sigma$).

Let $\mathcal{M}$ be a process discovery algorithm that returns for any log a fitting model (i.e., a Petri net that can replay the log). Let $N_S = \mathcal{M}(S^+)$. Then (N_S, Q) is the *subprocess* of S.

The discovery algorithm $\mathcal{M}$ will produce transitions labeled with the actions occurring in S^+ and a start transition t_{start} with label *start* and an end transition t_{end} with label *end*. In the following, we assume that ${}^\bullet t_{start} = \emptyset$ and $t_{end}{}^\bullet = \emptyset$, i.e., that start and end transitions have no pre- or post-places. In case $\mathcal{M}$ produced pre- and post-places for start and end, these places can be safely removed without changing that N_S can replay S^+. When repairing N, we connect t_{start} and t_{end} to the location Q of the subprocess.

Algorithm 1 defines how to repair a Petri net N w.r.t. a log L by adding subprocesses. The algorithm takes as input a complete set of sublogs $\mathcal{S}$ obtained from an alignment $\alpha(L)$ of L to N; it returns the *subprocess-repaired model* of N w.r.t. $\mathcal{S}$.

Algorithm 1 Subprocess-based repair

procedure RepairSubprocess(net N, complete set of sublogs $\mathcal{S}$)
  $N' \leftarrow N$ *// create copy for repair*
  for all $t \in T_{N'}, \ell(t) \neq \tau$ **do**
    if exists model move $(\gg, t)$ in some subtrace in $\mathcal{S}$ **then**
      add to N' a new transition t_τ with ${}^\bullet t_\tau = {}^\bullet t, t_\tau{}^\bullet = t^\bullet, \ell'(t_\tau) = \tau$
  for all $(S, Q) \in \mathcal{S}$ **do**
    $(N_S, Q) \leftarrow$ the subprocess of (S, Q) according to Def. 9
    $start_S, end_S \leftarrow$ the start and end transitions of N_S
    add N_S to N' *// assume N_S and N' are disjoint*
    for all $p \in Q$ **do** add arcs $(p, start_S), (end_S, p)$ to N'
    set $\ell'(start_S) := \tau, \ell'(end_S) := \tau$
  return repaired net N'

Theorem 1. *Let L be a log, let N be a Petri net. Let $\alpha(L)$ be the alignments of the traces of L to N and $\mathcal{S}$ be a complete set of sublogs of $\alpha(L)$. Let N' be a subprocess-repaired model of N w.r.t. $\mathcal{S}$. Then each trace $l \in L$ is a labeled occurrence sequence of N', that is, N' can replay L.*

Sketch. The theorem holds from the observation that each alignment $\alpha = (a_1, t_1) \dots (a_n, t_n) \in \alpha(L)$ of L to N can be transformed into an alignment of L to N' having synchronous moves or model moves on invisible transitions only, as follows.

Every move on model $(\gg, t_i)$ of α w.r.t. N is replaced by a model move $(\gg, t_{i,\gg})$ w.r.t. N' on the new invisible transition $t_{i,\gg}, \ell(t_{i,\gg}) = \tau$ which allows to skip over t_i.

Every move on log $(a, \gg)$ w.r.t. N is part of a subtrace $\beta = (a_1, \gg) \dots (a_k, \gg)$ of a sublog $(S, Q) \in \mathcal{S}$. By adding the subprocess N_S at location Q, the subtrace β is replaced by a sequence $(\gg, start_S)(a_1, t_1) \dots (a_k, t_k)(\gg, end_S)$ of synchronous moves in the subprocess N_S. Moves $(\gg, start_S)$ and $(\gg, end_S)$ are harmless because they are made silent by relabeling $start_S$ and end_S with τ. □

This theorem concludes the basic techniques for repairing a process model w.r.t. fitness to a given log. Observe that original model N is preserved *entirely* as we only add *new transitions* and *new subprocesses*. By taking a best alignment $\alpha(L)$ of L to N, one ensures that number of new τ-transitions and the number of new subprocesses is minimal.

4.4. Optional subprocesses instead of loops – better precision

The quality of the subprocess-based repair can be improved in some cases. Algorithm 1 adds for each sublog (S, Q) a subprocess N_S that consumes from and produces on the same set Q of places, i.e., the subprocess is a loop. If this subprocess is entered in each trace of L only once, then N_S is also executed exactly once. Thus, N could be repaired by inserting N_S in sequence (rather than as a loop), by refining the places $Q = \{q_1, \dots, q_k\}$ to places $\{in_1, \dots, in_k\}$ and $\{out_1, \dots, out_k\}$ with

1. ${}^\bullet in_j = {}^\bullet q_j, in_j{}^\bullet = \{start_S\}, j = 1 \dots, k$, and
2. $out_j{}^\bullet = q_j{}^\bullet, {}^\bullet out_j = \{end_S\}, j = 1 \dots, k$.

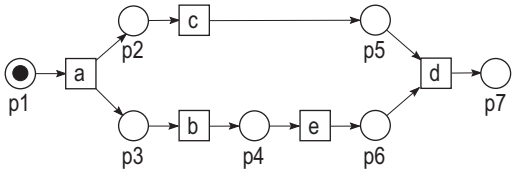
If N_S is entered at most once per case of L, we can additionally add a transition t^S_{skip} with ${}^\bullet(t^S_{skip}) = \{in_1, \dots, in_k\}$ and $(t^S_{skip})^\bullet = \{out_1, \dots, out_k\}$ which allows to skip N_S.

A model repaired by adding N_S in an (optional) sequence has a higher precision than the model repaired by adding N_S in a loop (Alg. 1). In the latter, the model N_S can be executed an arbitrary number of times, in the former (at most) once. However, this improvement for precision is only correct if there is no other subprocess N'_S with an overlapping location $Q' \cap Q \neq \emptyset$, as our repair technique does not discover orderings between different subprocesses at overlapping locations.

5. Improving Repair

Section 4 introduced a basic technique for repairing a process model: it adds a subprocess wherever a part of the log cannot be replayed on the model. The resulting model N' fits the log, and the use of subprocesses (as a loop or in sequence) caters for the precision of N'. In this section, we present techniques which specifically address *simplicity* of N'. We show techniques

1. for identifying loops in sublogs and repairing N by inserting a single transition rather than adding a large subprocess, and



$$\alpha(l) = \begin{array}{|c|c|c|c|c|c|c|c|c|c|} \hline \mathsf{a} & \mathsf{b} & \mathsf{c} & \gg & \mathsf{d} & \mathsf{b} & \mathsf{c} & \mathsf{b} & \mathsf{c} & \mathsf{d} \\ \hline \mathsf{a} & \mathsf{b} & \mathsf{c} & \mathsf{e} & \mathsf{d} & \gg & \gg & \gg & \gg & \gg \\ \hline \end{array}$$

Figure 6: Model that shall be repaired w.r.t. the trace abcdbcbcd; $\alpha(l)$ is the corresponding alignment.

2. for removing from N' unused or rarely used parts (after loops and subprocesses are added); both techniques are optional.

In addition, we show that simplicity of N' can be improved by computing alignments and sublogs with favorable qualities; we present pre-processing techniques

1. for picking a specific alignment of log L to original model N that yields less changes to N, and
2. for aligning, decomposing, and clustering subtraces into sublogs to improve the structure of subprocesses in N', and
3. how to pick a more specific location on where a subprocess or loop is added to N based on locations of subtraces.

These three techniques can be used to pre-process the input of any of the repair techniques. Each of the pre-processing techniques can be applied on its own or in combination with others in the given order. The complete algorithm for our repair technique is given at the end of this section.

5.1. Identifying Loops

Up to now, we repaired N by adding subprocesses which can replay sublogs of non-fitting events. This section and the next one, introduce two different operations for repairing process models which address simplicity.

5.1.1. Observation: subprocesses cannot handle repeating behavior well

A loop of a process manifests itself in the log L in repetitive patterns. For instance, the trace $l =$ abcdbcbcd has the repetitive pattern of events bcd with some events occasionally missing, e.g., event d in the second iteration. If the original model N can only replay a single occurrence of the pattern bcd but not several iterations of the pattern, then N has to be repaired. However, the example of Figures 6 and 7 shows that the subprocess-based repair of Sect. 3 in this case yield a counter-intuitive and unnecessarily complex model.

Aligning the trace $l =$ abcdbcbcd to the model N of Fig. 6 yields for instance the alignment $\alpha(l)$. The first iteration of the pattern bcd can still be replayed on N, but the second and third iteration cannot and lead to the sublog bcbcd. Repairing N w.r.t. $\alpha(l)$ according to Sect. 4 results in the model shown in Fig. 7;

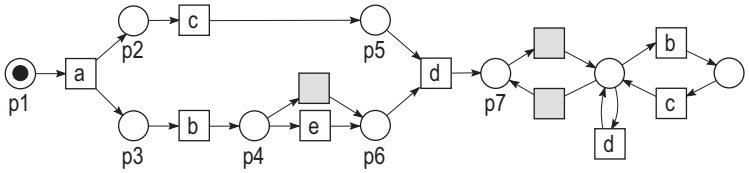


Figure 7: Sub-process based repair of the model of Fig. 2 w.r.t. the trace abcdbcbcd.

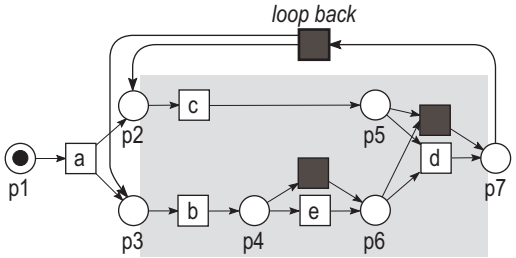


Figure 8: Loop-based repair of the model of Fig. 2 w.r.t. the trace abcdbcbcd.)

the subprocess added at place p7 allows to replay the two additional iterations bc and bcd.

The model of Fig. 7 clearly has disadvantages: it has a rather complex structure, and there is no explicit loop that describes the three iterations bcd, bc, bcd. Instead, different parts of the model describe the same behavior multiple times.

5.1.2. Idea: discover structured loops

A more favorable solution for repair would be the model shown in Fig. 8 that defines a structured loop. The loop body is indicated by the grey-shaded area, the "loop back" transition takes a token from the loop exit (place p7) back to its entry (places p2, p3). This model can replay the trace $l =$ abcdbcbcd. In the following, we present a technique that achieves this repair.

To repair a model N w.r.t. structured loops, we have to identify three things. (1) We have to identify whether a trace has several iterations of a repetitive pattern from which only the first iteration can be replayed in N and the others cannot. In the model N, we have to identify (2) the part that can replay the first iteration of the repetitive pattern, so that (3) by adding a "loop back" transition also the second, third, ... iteration of the pattern can be replayed.

The challenge that we face is that not all iterations of the pattern are identical, for instance in case of skipped events or alternative paths within the loop body. This makes it difficult to identify the patterns that are repeated. Our solution is to avoid finding the patterns explicitly, but rather take each subtrace β as a hypothesis that "it could be an iteration of a loop". We test the hypothesis by searching for a loop body that, when extended by a loop back transition, can replay β. If this loop body exists, the hypothesis is true and the identified loop repairs N. If it does not exist, N cannot replay β through a loop and we use the subprocess-based repair.

We illustrate the idea by an example; the algorithm is given afterwards. In

Fig. 6, the alignment $\alpha(l)$ yields the sublog $(\{\mathsf{bcbcd}\}, \{\mathsf{p7}\})$. From the activities $\mathsf{b}, \mathsf{c}, \mathsf{d}$ and the loop exit $\mathsf{p7}$, we identify the loop hypothesis consisting of all nodes between places $\mathsf{p2}, \mathsf{p3}$ and $\mathsf{p7}$, see the grey-shaded area in Fig. 8. When adding a loop back transition from $\mathsf{p7}$ to $\mathsf{p2}, \mathsf{p3}$, we can replay bcbcd with model moves $(\gg, \mathsf{e})$ and $(\gg, \mathsf{d})$. Thus, the hypothesis holds and we can add the loop back transition and two τ-transitions to repair the model of Fig. 6 which results in the model of Fig. 8.

There are a few more non-trivial aspects to finding loops which we discuss and solve as we present the formal definitions and algorithms.

5.1.3. Algorithm for loop discovery

For a sublog (S, Q) that cannot be replayed in N we find a potential loop body and test whether it is one, as follows.

1. The actions $\Sigma(S)$ in S are the activities of the potential loop body. The location Q contains the places that were marked when any subtrace $\beta \in S$ could not be replayed on N. If there is a loop, the location Q contains the *exit* of the loop body. The loop body thus consists of all transitions labeled with $\Sigma(S)$ that precede places in Q. We search in N for the smallest connected subnet N_S that contains these transitions and ends with Q. The places of N_S that have no pre-transition are the *loop entry*. We complete N_S to a loop by adding a loop back transition that consumes from the exit Q and produces on the entry of the loop.
2. To test whether (S, Q) describes loop iterations, we test whether the loop N_S can replay S when there are tokens initially in Q (the loop exit). To account for variations in loop iterations, e.g., skipped events, we compute an alignment $\alpha(S)$ of S to N_S where costs log moves are set high relative to costs for model moves (e.g., factor 100). If $\alpha(S)$ contains no log moves, then N_S can replay the loop, that is, the hypothesis holds. We can repair N by adding the identified loop back transition. If $\alpha(S)$ contains model moves, corresponding τ-transitions have to be added to N as well as, see Alg. 1.

Formal definitions. The loop discovery algorithm is given in Alg. 2. It extends the given original model $N = (P, T, F, \ell, m_0, m_f)$ with a loop back transition for each sublog $(S, Q) \in \mathcal{S}$ that indeed describes a loop. The algorithm builds on some notions. Let *distance*(N, x, Q) denote the length of a shortest path from node x to some node $q \in Q$ along the arcs F. We define the *subnet* $N[T']$ *induced by a set* $T' \subseteq T$ of transitions as $N[T'] = (P', T', F', \ell', m_0', m_f')$ where $P' = \bigcup_{t \in T'} ({}^\bullet t \cup t^\bullet)$, $F' = F|_{(P' \times T') \cup (T' \times P')}$, $\ell' = \ell|_{P' \cup T'}$, $m_0'(p) = 1$ if there is no arc $(p, t) \in F'$ and $m_0'(p) = 0$ otherwise, and $m_f'(p) = 1$ if there is no arc $(t, p) \in F'$ and $m_f'(p) = 0$ otherwise. In $N[T']$, places without pre-transition (post-transition) are in the initial (final) marking.

The net N_{loop} returned by Alg. 2 is repaired w.r.t. log moves in $\mathcal{S}$ that can be attributed to iterative behavior. N_{loop} is *not* repaired w.r.t. model moves and w.r.t. non-iterative behavior. These repairs can be achieved by computing an

Algorithm 2 Repairing a model by discovering and adding loops

procedure AddLoops(original model N, sublogs $\mathcal{S}$, cost function κ)
  $N_{loop} \leftarrow N$ *// copy of N*
  for all $(S, Q) \in \mathcal{S}$ **do**
    $T_S \leftarrow \emptyset$ *// transitions of the loop body*
    for each log move $(a, \gg)$ in a subtrace in S **do**
      choose $t_a \in T_S$ with $\ell(t_a) = a$ and $distance(N, t_a, Q)$ minimal
      $T_S \leftarrow T_S \cup \{t_a\}$
    for each $t \in T$ on a path in N from $t_1 \in T_S$ to $t_2 \in T_S$ **do**
      $T_S \leftarrow T_S \cup \{t\}$ *// add all transitions in between*
    $N_S \leftarrow N[T_S]$ *// copy of potential loop body*
    $entry_S \leftarrow \{p \in P_S \mid m_{0,S}(p) > 0\}$
    $exit_S \leftarrow \{p \in P_S \mid m_{f,S}(p) > 0\}$ *// test the loop body*
    add a new transition $loop_S$ with ${}^\bullet loop_S = entry_S$, $loop_S^\bullet = exit_S$ to N_S
    $\alpha \leftarrow alignment(S, N_S, \kappa)$
    if α contains no log move **then**
      add transition $loop_S$ with ${}^\bullet loop_S = entry_S$, $loop_S^\bullet = exit_S$ to N_{loop}
  return N_{loop}

alignment $\alpha_{loop}(L)$ of L to N_{loop} and then applying the subprocess-based repair. The alignment $\alpha_{loop}(L)$ contains less deviations than the original alignment $\alpha(L)$ of L to N, because some deviations are already repaired in N_{loop}. The full procedure is given in Sect. 5.6.

Iteration preserving alignments. Algorithm 2 will be ineffective if loop iterations are scattered over different sublogs; it works best if many loop iterations end up in the same subtrace in some sublog. This can be influenced by finding a particular alignment $\alpha(l)$ that contains log moves "as late as possible".

A negative example is the following alignment of trace abcdbcbcd to the model of Fig. 6, $\alpha'(l) = \begin{array}{|c|c|c|c|c|c|c|c|c|c|}\hline \mathsf{a} & \mathsf{b} & \mathsf{c} & \mathsf{d} & \mathsf{b} & \mathsf{c} & \gg & \mathsf{b} & \mathsf{c} & \mathsf{d} \\ \hline \mathsf{a} & \mathsf{b} & \gg & \gg & \gg & \mathsf{c} & \mathsf{e} & \gg & \gg & \mathsf{d} \\ \hline \end{array}$. Alignment $\alpha'(l)$ has the same cost as $\alpha(l)$ of Fig. 6, but differs in its representation of loop iterations. Our approach would find a loop body for subtrace $(\mathsf{bc}, \{\mathsf{p5}, \mathsf{p6}\})$ of $\alpha'(l)$, but not for $(\mathsf{cdb}, \{\mathsf{p2}, \mathsf{p4}\})$, because transition d lies after the loop exit $\{\mathsf{p2}, \mathsf{p4}\}$. Alignment $\alpha'(l)$ violates the notion of having log moves "as late as possible" by the log move $(\mathsf{c}, \gg)$ that occurs before the synchronous move (c, c).

For discovering loops, we prefer $\alpha(l)$ of Fig. 6 over $\alpha'(l)$. In $\alpha(l)$ the first loop iteration is completely described by synchronous moves and all subsequent iterations are grouped into one subtrace. An alignment of this kind is *iteration preserving*, which we characterize as follows:

1. For any synchronous move (a, t) that is preceded by a sequence $(a_1, \gg) \ldots (a_k, \gg)$ of log moves, no log move $(a_i, \gg)$ could have matched t, i.e., $a \neq a_i$ for all $1 \leq i \leq k$.
2. A model move $(\gg, t)$ is either preceded by a synchronous move or by another model move, i.e., log moves are put last.

The ProM plugin that implements the technique of Sect. 2.4 for finding a best alignment uses a heuristics that returns iteration preserving alignments [10, 11]. However, it might be that the first iteration can only be completed when a large number of model moves is used. In that case, the alignment we are looking for would not be a best alignment as additional model moves increase the cost. In such a situation, the cost for model moves can be set to 0 (the same as synchronous moves), allowing to replay all events of the first iteration at no extra cost.

5.2. Removing unused and infrequent parts

So far, we repaired N by adding new transitions, subprocesses, and loops that allow to replay all traces of log L. However, N may also contain transitions for which there is no event in L. These superfluous transitions, so far, remain in the repaired model N'; if superfluous transitions were removed, we would repair N also in terms of *precision* w.r.t. L and *simplicity*.

The idea for removing unused parts is simple. We align the log L to the repaired model N'. The alignment contains only synchronous moves and model moves on transitions of N'. We count for each transition t of N' how often it participates in a move. If that number is 0, then t is not used, and we remove t. In the same way, unused places can be removed. See Alg. 3 for the corresponding procedure.

Algorithm 3 Removing unused or rarely used nodes

procedure RemoveInfrequent(model N, log L, cost fct. κ, thresh. k)
  $\alpha \leftarrow \mathit{alignment}(L, N, \kappa)$
  for all $t \in T_N$ **do** $\mathit{used}(t) \leftarrow$ number of times t occurs in a move in α
  for all $p \in P_N$ **do** $\mathit{used}(p) \leftarrow \sum_{t \in {}^\bullet p} \mathit{used}(t) + m_0(p)$
  for all $x \in T_N \cup P_N$ where $\mathit{used}(x) \leq k$ **do**
    remove x and all adjacent arcs from N
  return N

The threshold parameter allows to remove also nodes that are rarely used. Removing them will impair fitness but improve simplicity. Thus, a user may seek a favorable spot in the model repair spectrum, as discussed in Sect. 3.

5.3. Picking specific alignments for repair

The algorithms introduced so far take *alignments* and *sublogs* as input to repair a process model N w.r.t. a log L. In the next three sections, we investigate which *properties of an alignment or a sublog* yield more favorable repairs, and how to obtain alignments and sublogs with such properties.

Uniform deviations yield simpler repairs. The basis for repairing N w.r.t. L is an alignment of N to L that highlights were L and N deviate. So far, we considered an alignment α that shows the *smallest number* of deviations between N and L, as it is computed by the cost-based approach of Sect. 2.4. However,

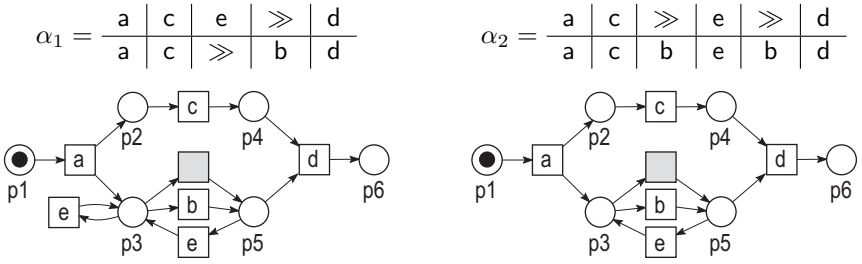


Figure 9: The alignment α_2 has more uniform deviations compared to α_1. Repairing the net of Fig. 2 w.r.t. α_2 yields a simpler model (right) than the repair w.r.t. α_1 (left).

the simplicity of the repaired model N' does not only depend on the number of deviations, but also on their *uniformity*. The more uniform the deviations are the more likely it is they can be repaired in the same way, for instance, with the same transition or subprocess.

Figure 9 shows two alignments α_1 and α_2 of the trace $l = \mathsf{aced}$ to the net of Fig. 2. Both alignments have the same number of deviations, but α_2 is more uniform as both deviations are a b-move on model $(\mathsf{b}, \gg)$, whereas α_1 has one log move $(\gg, \mathsf{e})$ and one model move $(\mathsf{b}, \gg)$. Repairing the net of Fig. 2 w.r.t. α_1 or α_2 yields the respective two nets shown in Fig. 9. The deviations in α_2 can both be repaired by the same τ-labeled transition, whereas the repair for α_1 needs an additional e-labeled transition; the net in Fig. 9(right) is simpler and more similar to the original net of Fig. 2. Next, we present a technique that helps picking alignments with more uniform deviations, such as α_2.

Compute a global cost function. The best alignment of L to N is defined w.r.t. a cost function κ; we can adjust κ to obtain a particular alignment. The idea is to find a cost function κ^* in which a log move that occurs very rarely has high costs (and hence is avoided), and a log move that occurs very frequently has low costs (and hence is preferred); correspondingly for model moves. For instance in Fig. 9, when action b has lower costs than event e, then alignment α_2 has less cost than alignment α_1.

Such a cost function κ^* can be found by looking *globally* on all deviations between N and L, as follows. First compute an alignment α with a given cost function κ; κ can be uniform or already take other aspects into account. Count for each action $a \in \Sigma$ and each transition $t \in T$ the number of respective log moves and model moves in α. The most frequent move yields the "most efficient" repair whereas all other moves yield "less efficient" repairs. We obtain κ^* by scaling the original costs in κ with respect to how frequent a move occurs: the most frequent move keeps its costs, the cost of any other move is scaled up more the less frequent it is. Algorithm 4 gives the explicit definition to compute κ^*.

Computing a best alignment w.r.t. κ^* will prefer moves with lower costs (i.e., which yield a more efficient repair across the entire log L) and avoid moves with higher costs (i.e., which yield a more expensive repair w.r.t. entire L). The above procedure could be iterated, i.e., compute $\kappa^{**} = \textsc{GlobalCost}(L, N, \kappa^*)$. However, in experiments we found the alignments returned for κ^* and κ^{**} to be

Algorithm 4 Computing a global cost function

procedure GLOBALCOST(log L, net N, cost function κ)
  $\alpha \leftarrow$ *alignment*(N, L, κ)
  for all $a \in \Sigma$ **do** $dev(a) \leftarrow$ number of log moves $(a, \gg)$ in α
  for all $t \in T$ **do** $dev(t) \leftarrow$ number of model moves $(\gg, t)$ in α
  $devMax \leftarrow \max_{x \in \Sigma \cup T} dev(x)$ *// most efficient repair*
  $\kappa^*(x) \leftarrow \kappa(x) \cdot \frac{devMax}{dev(x)}$, for all $x \in \Sigma \cup T$
  return global cost function κ^*

identical.

5.4. Aligning, decomposing, and clustering subtraces into sublogs

The repair technique of Sect. 3 extracts sublogs from an alignment of L to N and adds a subprocess to N for each sublog. The *simplicity* of the repaired model depends on the *number of subprocesses* added (fewer means simpler), and how simple *each subprocess* is. So far, we only gave a naïve procedure to extract sublogs: group subtraces at the same location. In the following, we investigate the forces that influence simplicity of subprocesses and propose a procedure for extracting sublogs that aims at a simple repaired model.

5.4.1. Simplicity of subprocesses

A subprocess N_S added to N will be simpler if the sublog (S, Q) from which N_S is derived contains many similar traces and no outliers (or noise). The number of subprocesses in turn can be reduced by grouping as many traces as possible into the same sublog. This observation reveals two forces that affect simplicity: (1) any two subtraces with overlapping locations should be in the same sublog, and (2) dissimilar traces should not be in the same sublog.

We address these two forces with the following approach. First partition the subtraces $\beta(L)$ into sets $Sub_1, \ldots, Sub_k$ of similar subtraces, so that traces in different sets are dissimilar. Then subtraces that are in the same set Sub_i and have overlapping locations are grouped into the same sublog, which may yield several sublogs depending on the overlap. We confirmed in experiments that repairing N w.r.t. the sublogs obtained in this way will yield a simpler structure of the repaired process model. We present both steps in the following.

5.4.2. Partitioning subtraces based on similarity

Figure 10 illustrates the problem of partitioning the subtraces $\beta(L)$ into sets of similar subtraces by an example. The subtraces to the left are very diverse. If we group all of them into a single sublog, we could discover the subprocess N_0, which has a rather complex structure. If we decided to put each subtrace into a different sublog, we would obtain simple subprocesses, but these contain certain structures multiple times. For instance, the sequence cdef would be represented twice. Simplicity can be achieved by partitioning the subtraces into sets of mutually similar traces.

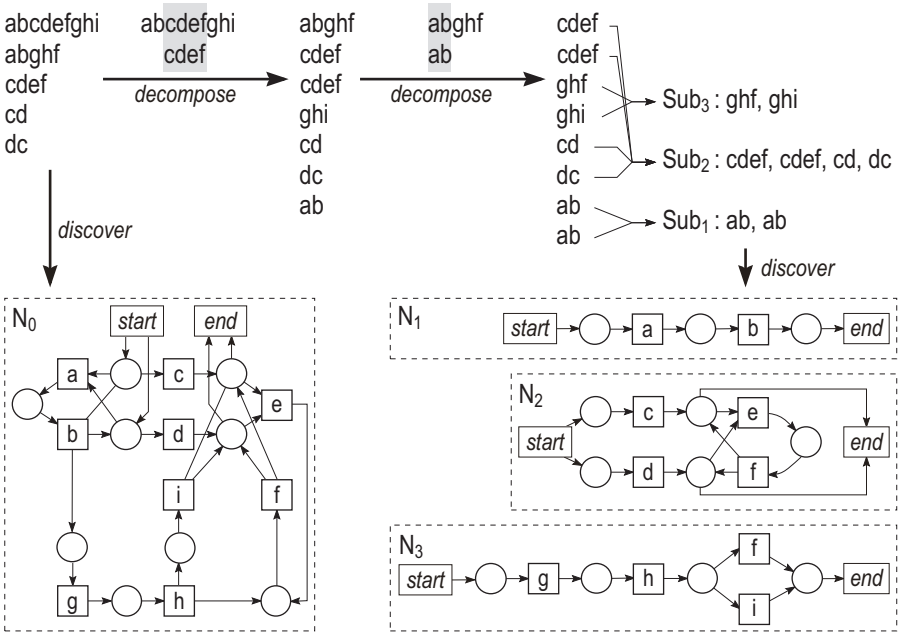


Figure 10: Decomposing and aligning sublogs to obtain multiple, simpler subprocesses.

However, there is no obvious partition of the subtraces shown in Fig. 10. Subtrace cdef is similar to cd and to a part of abcdefghi, but cd is rather different from abcdefghi. Also abghf and cdef are similar to a part of abcdefghi, but cdef and abghf are not similar to each other. In any case, we either end up with sublogs of dissimilar traces (yielding a complex subprocess), or with many small sublogs (yielding duplicated structures). In the following, we propose a technique that avoids this trade-off.

Idea: decompose subtraces. Partitioning the subtraces of Fig. 10 fails, because some traces are only *similar to a part* of another trace. We avoid this problem by decomposing each subtrace into parts that are similar to other subtraces.

The algorithmic idea is as follows. We call two subtraces *similar* if they share 50% of their actions (regardless of the ordering). A long subtrace β decomposes into shorter subtraces $\beta = \beta_{prefix}\beta'\beta_{suffix}$ whenever β' is also a subtrace found by the alignment (and β and β' themselves are dissimilar). Decomposition starts with the longest subtrace β and the longest part β'. This way, we decompose into the longest similar subtraces that we can find. When no subtrace can be decomposed anymore, any two subtraces are either clearly similar or clearly dissimilar. Formal definitions are given below.

Figure 10 illustrates this procedure. In the first step, abcdefghi decomposes into ab, cdef, ghi because of cdef. From the resulting set of subtraces, abghf decomposes into ab and ghf because of ab. In the resulting set, no subtrace can be decomposed anymore, e.g., trace cdef is not decomposed into cd and ef because cd and cdef are similar (they share at least 50% of the actions).

Grouping pairwise similar traces yields the three sets Sub_1, Sub_2, Sub_3 shown in Fig. 10(right). From these, we can discover the subprocesses N_1, N_2, and N_3 which have a simpler structure; only N_2 and N_3 overlap on transition f. Note that by adding N_1, N_2, and N_3 to the same location (their start and end transitions are connected to the same places), the 3 subprocesses together replay the subtraces of Fig. 10(left).

Formal definitions. Let $\Sigma(\beta)$ denote the actions occurring in a subtrace β. Subtraces (β_1, Q_1) and (β_2, Q_2) are *similar*, written $(\beta_1, Q_1) \sim (\beta_2, Q_2)$, iff they share at least 50% of the actions, i.e., $\frac{|\Sigma(\beta_1) \cap \Sigma(\beta_2)|}{|\Sigma(\beta_1)|} \geq 0.5$ and $\frac{|\Sigma(\beta_1) \cap \Sigma(\beta_2)|}{|\Sigma(\beta_2)|} \geq 0.5$. Let $\approx$ be the transitive closure of $\sim$, i.e., $\approx$ is the equivalence relation induced by $\sim$. Let Sub be a set of subtraces; $\approx$ partitions Sub into its equivalence classes $Sub_1, \ldots, Sub_k$ where for each $i = 1, \ldots, l$, $(\beta_1, Q_1), (\beta_2, Q_2) \in Sub_i$ iff $(\beta_1, Q_1) \approx (\beta_2, Q_2)$. Finally, we write $Sub[n]$ for the set of subtraces in Sub that have length n. Algorithm 5 shows the complete algorithm.

Algorithm 5 Decomposing and aligning subtraces

procedure AlignSubtraces(subtraces Sub)
  $n_{\max} \leftarrow$ length of longest subtrace in Sub
  for $n = n_{\max} \ldots 1$ **do**
    for all $(\beta, Q) \in Sub[n]$ **do**
      choose $\beta_0\beta_1\beta_2 = \beta$ where $\beta_1 \in Sub[k]$, $k < n$ maximal, $\beta \not\sim \beta_1$
      if $\beta_0\beta_1\beta_2$ exist **then**
        $Sub \leftarrow Sub \setminus \{(\beta, Q)\}$
        $Sub \leftarrow Sub \cup \{(\beta_1, Q), (\beta_2, Q), (\beta_3, Q)\}$
  $Sub \leftarrow Sub \setminus Sub[0]$ *// remove empty subtraces*
  $Sub_1, \ldots, Sub_k \leftarrow$ equivalence classes of Sub wrt $\approx$
  return $Sub_1, \ldots, Sub_k$

5.4.3. Grouping subtraces into sublogs

Algorithm AlignSubtraces returns sets $Sub_1, \ldots, Sub_k$ of similar subtraces. By Def. 8, a set Sub_i of subtraces is only a log if their joint location is non-empty. Otherwise we split Sub_i based on the locations of its subtraces.

As before, locations of subtraces may overlap in a way that some subtraces could be placed in more than one sublog. Consider for example the subtraces $(\mathsf{abc}, \{\mathsf{p}, \mathsf{q}\}), (\mathsf{abd}, \{\mathsf{q}, \mathsf{r}\}), (\mathsf{abe}, \{\mathsf{q}, \mathsf{r}\}), (\mathsf{abf}, \{\mathsf{p}, \mathsf{s}\})$. Their joint locations is empty; when grouping the traces into sublogs, we have to choose whether $(\mathsf{abc}, \{\mathsf{p}, \mathsf{q}\})$ should be in one log with $(\mathsf{abd}, \{\mathsf{q}, \mathsf{r}\}), (\mathsf{abe}, \{\mathsf{q}, \mathsf{r}\})$ (locations overlap on q or in one log with $(\mathsf{abf}, \{\mathsf{p}, \mathsf{s}\})$ (overlap on p).

In experiments, we found that the repaired model is more structured when we consider the "frequency" of a place w.r.t. a set Sub of subtraces. Whenever a subtrace has a place p in its location, the process should have been able to replay that trace when there was a token on p. The more often p occurs, the more *frequent* it is w.r.t. behavior that has to be repaired. We interpret a frequent place p as more important for the placement of a subprocess. In

contrast, a less frequent place just happened to be marked when a subtrace had to be executed; it is less significant on where to add a subprocess to repair a model. Thus, we propose to group subtraces into sublogs based on the frequency of the places in their locations, starting with the most frequent places. In our example above, place q would be the most frequent one, and we would group $(\mathsf{abc}, \{\mathsf{p}, \mathsf{q}\}), (\mathsf{abd}, \{\mathsf{q}, \mathsf{r}\}), (\mathsf{abe}, \{\mathsf{q}, \mathsf{r}\})$ into one sublog with location $\{\mathsf{q}\}$ and $(\mathsf{abf}, \{\mathsf{p}, \mathsf{s}\})$ into another sublog with location $\{\mathsf{p}, \mathsf{s}\}$.

Formal definitions. For a set Sub of subtraces and a place $p \in P$, let $Sub[p]$ denote the subtraces of Sub having p in their location, $Sub[p] = \{(\beta, Q) \in Sub \mid p \in Q\}$. Algorithm 6 groups subtraces Sub into sublogs.

Algorithm 6 Grouping subtraces to sublogs

procedure GROUPINTOSUBLOGS(subtraces Sub)
  $\mathcal{S} \leftarrow \emptyset$
  while $Sub \neq \emptyset$ **do**
    choose place $p \in P$ with $|Sub[p]|$ is maximal
    $S \leftarrow Sub[p]$ *// all subtraces at place p*
    $Q \leftarrow \bigcap_{\beta \in S} loc(\beta)$ *// at least $p \in Q$*
    $\mathcal{S} \leftarrow \mathcal{S} \cup \{(S, Q)\}$
    $Sub \leftarrow Sub \setminus Sub[p]$
  return $\mathcal{S}$

5.4.4. Complete algorithm

Algorithm 7 shows the complete procedure which extracts from the set of all subtraces $\beta(L)$, a complete set $\mathcal{S} = \text{GROUPINTOALIGNEDSUBLOGS}(\beta(L))$ of sublogs, where each sublog contains only similar traces. We noticed in experiments that aligned sublogs yield simpler subprocesses, and that we were able to to identify more loops from aligned sublogs than from unaligned ones.

Algorithm 7 Aligning subtraces to sublogs

procedure GROUPINTOALIGNEDSUBLOGS(subtraces Sub)
  $\mathcal{S} \leftarrow \emptyset$
  $Sub_1, \ldots, Sub_k \leftarrow$ ALIGNSUBTRACES(Sub)
  for $i = 1, \ldots, k$ **do**
    $\mathcal{S} \leftarrow \mathcal{S} \cup$ GROUPINTOSUBLOGS(Sub_i)
  return $\mathcal{S}$

5.5. Improving the placement of a subprocess

A sublog (S, Q) returned by Alg. 7 may still have a large location Q. The location Q could be reduced further if additional information from the entire original log L is taken into account. The idea is that a place $p \in Q$ is more significant for a subtrace in S if it was marked last before that subtrace had to be executed. The subset of places in Q that were marked last the most often are

considered as most significant; all other less significant places are removed from Q.

Formal definitions. Recall from Sect. 4.3 that each subtrace $\beta \in S$ was extracted from an alignment $\alpha(l)$ of a trace $l \in L$ to the model N. Let (a, t) be the synchronous move that precedes β in $\alpha(l) = \ldots (a, t)(\gg, t'_1) \ldots (\gg, t'_k)\beta \ldots$, i.e., we ignore the model moves $(\gg, t'_i)$. The places marked last before β are the post-places $t^\bullet$ in N; we write $\mathit{last}(\beta, N) := t^\bullet$. Given sublog (S, Q), the number of times a place $p \in Q$ was marked last before a subtrace $\beta \in S$ is defined as $\mathit{last}(p, S, N) = |\{\beta \in S \mid p \in \mathit{last}(\beta, N)\}|$. The higher $\mathit{last}(p, S, N)$, the more significant is p for S. Algorithm 8 reduces the location of each sublog to the most significant places. Note that the location of a sublog (S, Q) remains unchanged if $\mathit{last}(\beta, N) \cap Q = \emptyset$ as $\mathit{last}(p, S, N) = 0$ for all p.

Algorithm 8 Picking relevant locations of sublogs

procedure PickRelevantLocations(net N, sublogs $\mathcal{S}$)
  $\mathcal{S}' \leftarrow \emptyset$
  for all $(S, Q) \in \mathcal{S}$ **do**
    $\mathcal{S}' \leftarrow \mathcal{S}' \cup \{(S, \{p \in Q \mid \mathit{last}(p, S, N) \text{ is maximal}\})\}$
  return $\mathcal{S}'$

We found in experiments that by this technique, subprocesses introduced during repair are less likely to have overlapping locations.

5.6. Complete model repair procedure

We experimentally evaluated the effect each of the techniques introduced so far. Based on the insights gained there (see Sect. 6), we designed the extended repair Algorithm 9. It takes as input the original model N and the log L. In addition, the user can provide a cost function κ to specify how severe deviations w.r.t. particular transitions or events are; a uniform cost function with costs 1 can be used as well. Finally, threshold k will be used when removing infrequently used parts from N.

The algorithm has three major steps: (1) First repair the model for loops using AddLoops (Alg. 9), (2) then for subprocesses and skipped activities using RepairSubprocess (Alg. 1), and (3) finally remove infrequent nodes (Alg. 3). The algorithm can be configured in various ways:

1. Steps (1) and (3) can be omitted.
2. Repairing for loops is most effective if the cost function κ_{loop} used for computing the alignment uses 0 costs for model moves and high costs for log moves; though other cost functions are possible.
3. Subprocess-based repair yields less subprocesses when the cost function κ_{fit} is computed by GlobalCost (Alg. 4); though one could also omit GlobalCost and set $\kappa_{fit} \leftarrow \kappa$ for the given cost function κ.
4. In steps (1) and (2), sublogs are extracted from the respective alignment by GroupIntoAlignedSublogs (Alg. 7); sublogs can also be extracted directly without aligning them by $\mathcal{S} \leftarrow$ GroupIntoSublogs$(\beta(L))$.

Algorithm 9 Extended procedure for model repair

procedure RepairExtended(model N, log L, cost function κ, threshold k)
  // (1) Repair for loops
  $\kappa_{loop} \leftarrow$ cost function with $\forall t \in N_T : \kappa(t) = 0, \forall a \in \Sigma(L) : \kappa(e) = 100$
  $\alpha_{loop}(L) \leftarrow$ *alignment*(N, L, κ_{loop})
  $\beta_{loop}(L) \leftarrow$ all subtraces of $\alpha_{loop}(L)$
  $\mathcal{S}_{loop} \leftarrow$ GroupIntoAlignedSublogs$(\beta_{loop}(L))$
  $\mathcal{S}_{loop} \leftarrow$ PickRelevantLocations$(N, \mathcal{S}_{loop})$
  $N_{loop} \leftarrow$ AddLoops(N, L, κ_{loop})

  // (2) Repair for subprocesses and skipped events
  $\kappa_{fit} \leftarrow$ GlobalCost(N_{loop}, L, κ)
  $\alpha(L) \leftarrow$ *alignment*(N, L, κ_{fit})
  $\beta(L) \leftarrow$ all subtraces of $\alpha(L)$
  $\mathcal{S} \leftarrow$ GroupIntoAlignedSublogs$(\beta(L))$
  $\mathcal{S} \leftarrow$ PickRelevantLocations$(N_{loop}, \mathcal{S})$
  $N_{fit} \leftarrow$ RepairSubprocess$(N_{loop}, L, \kappa_{fit})$

  // (3) Remove infrequent
  $N_{repaired} \leftarrow$ RemoveInfrequent$(N_{fit}, L, \kappa_{fit}, k)$
  return $N_{repaired}$

5. Procedure PickRelevantLocations (Alg. 8) to pick more specific locations for loops and subprocesses can be omitted.

Experimental results show that the complete procedure yields best results as we discuss in the next section.

6. Experimental Evaluation

The technique for model repair presented in this paper is implemented in the Process Mining Toolkit ProM 6 in the package *Uma*, available from `http://www.promtools.org/prom6/`. We briefly discuss some implementation details and then present an experimental evaluation of our techniques.

6.1. Implementation in ProM

The ProM package *Uma* provides a *Repair Model* plugin that implements Alg. 9. This plugin calls three separate plugins *Repair Model (find loops)*, *Repair Model (find subprocesses)*, and *Repair Model (remove unused parts)* which implement the three main repair steps, respectively. Each of these repair plugins takes as input a Petri net N, a log L, and a best-fitting alignment $\alpha(L)$ of L to N. The alignment $\alpha(L)$ can be computed in ProM 6 using the Conformance Checker of [9–11]. Additionally, we provide a plugin *Align Log And Model for Repair (global costs)* which computes an alignment using a global cost function (Alg. 4). Each of the repair plugins has an option to align sublogs of non-fitting events (Alg. 7) and to pick optimal sublocations (Alg. 8).

Table 1: Deviations of 10 logs from Dutch municipalities to a reference model and properties of manually repaired models.

	log		deviations			manual repair		
			moves on		per	add		similarity-
	traces	length	model	log	case	$\|P\|$	$\|T\|$	distance
M1	434	1-51	3327	310	1-26			
M2	286	1-72	1885	323	1-41			
M3	481	2-82	3079	1058	1-49			
M4	324	1-37	2667	192	2-21			
M5	328	2-43	3107	342	2-25			
$M1^f$	249	24-40	681	229	1-12	5	20	0.068
$M2^f$	180	23-70	516	240	1-41	6	27	0.095
$M3^f$	222	22-82	465	598	1-49	44	82	0.158
$M4^f$	239	15-37	1216	180	2-17	23	43	0.117
$M5^f$	328	13-43	1574	280	2-16	15	36	0.111

In an earlier version of this paper [18], sublogs were obtained by grouping subtraces that share the same location in a greedy way (pick sublogs with the largest overlap of places first).

In *Repair Model (find subprocesses)*, subprocesses are discovered from sublogs using the ILP miner [8] which guarantees to return a model that can replay the sublog. The returned model is then simplified according to [29] and added to N as a subprocess as defined in Alg. 1.

6.2. Experiment Data

We validated our implementation on real-life logs from a process that is shared by five Dutch municipalities. Figure 1(left) shows the reference base model that is used in several municipalities. However, each municipality runs the process differently as demanded by the "couleur locale". As a result, the process observed in each municipality substantially deviates from the base model.

We obtained 5 raw logs (M1-M5) from the municipalities' information systems. From these we created filtered logs ($M1^f$-$M5^f$) by removing all cases that clearly should not fit the base model, for instance because they lacked the start of the process or were incomplete (see Sect. 3 for the discussion). Table 1 shows the properties of these 10 logs (over 44 different actions) discussed in the following. The table lists the number of traces, minimum and maximum length, and the properties of a best alignment of the log to the model of Fig. 1(left) as the total number of model moves, number of log moves and the minimum and maximum number of deviations (log move or model move) per case. None of the traces could be replayed on the base model, in some cases deviations were severe.

6.3. Manual Repair

To obtain a "gold standard" regarding model repair the reference model was repaired manually by a modeling expert. For repair, deviations between filtered log and model in the alignment were identified and classified based on background information about the process. We observed that

1. activities were executed in reality in a different order than described by the process model,
2. groups of activities were executed repeatedly where the process model only allows for a single execution,
3. some activities had been skipped in some cases, and
4. some activities had never been executed.

Based on this classification, the model was repaired:

1. Where a different execution order was required, the ordering of activities was adjusted, for instance by reversing activity order, making activities parallel, or swapping activities and gateways in the process.
2. When a group of activities had been executed repeatedly, the model was repaired by either implementing a loop that allows for repeated execution of these activities, or by inserting a new block of activities. The latter repair was conducted in situations when repeated activities occurred interleaved with the main process flow or much later than their original occurrence in the process.
3. Optional activities were skipped by adding a τ-transition, preferably skipping over a sequence of optional activities where applicable.
4. Activities never executed were removed from the model.

After applying all repairs, the repaired model was checked for conformance, and additional changes were applied until the repaired model conformed to the given log. Figure 12(right) shows the manually repaired model for log M2^f, the changes are highlighted: one subprocess consisting of 10 activities was added, a group of 2 activities had to be moved to a different branch, one loop was implemented, and several activities had to be made optional. Table 1 shows the differences of the manually repaired models for each filtered log to the reference model which had 59 places, 68 transitions, and 152 arcs. Log M3^f required by far the largest changes in particular in adding new subprocesses and loops. Logs M4^f and M5^f required many repairs for optional activities. We also measured the *graph similarity distance* [33] of each manually repaired model to the reference model. The similarity distance roughly indicates the fraction of the original model that has to be changed to obtain the repaired/rediscovered model, i.e., 0.0 means identical models. The manual repairs increased the distance to the reference model between .068 and .158.

The time to manually repair one model was between 1hr for $M1^f$, around 3hrs for $M3^f$, $M4^f$ and $M5^f$ each, and 5hrs for $M2^f$ for which it was hardest to derive repair actions from the found deviations.

6.4. Automatic Repair

To evaluate the techniques for automatically repairing models, we conducted a series of experiments. In the first experiment, we repaired the reference model for each of the 10 logs using the subprocess-based repair of Alg. 1 followed by removing unused nodes with Alg. 3 based on an alignment of uniform cost function. The other experiments investigated the influence of the improvements

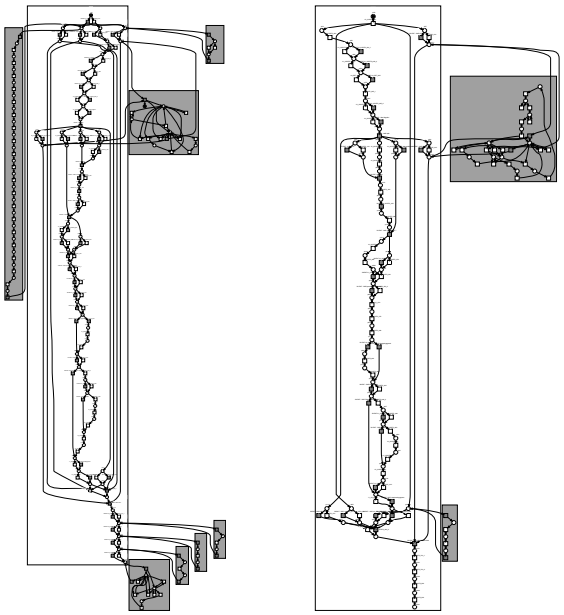


Figure 11: Result of repairing Fig. 1(left) w.r.t. M1 (left) and M2^f (right).

in model repair that can be achieved by the variants proposed in Sect. 5. The quality of the automatically repaired models is measured in terms of graph similarity distances to the reference model and to the manually repaired models.

6.4.1. Subprocess-based repair

Applying subprocess-based repair (Alg. 1 and 3) on the base model of Fig. 1(left) and the filtered log M1^f yields the model of Fig. 1(middle). Repairing the base model w.r.t. the raw log M1 results in the model shown in Fig. 11(left). Repairing the base model w.r.t. the filtered log M2^f yields the model of Fig. 11(right). In each case, model repair requires only several seconds; a best alignment (needed for repair) could be obtained in about a minute per log. We checked all resulting models for their capability to replay the respective log and could confirm complete fitness for all models.

Moreover, we could re-identify the original model as a sub-structure of the repaired model, making it easy to understand the made repairs in the context of the original model. Table 2 shows for each log the number of added subprocesses, the average and maximal number of transitions per subprocess, and the total number of added and of removed transitions in the entire process. We can see that in the worst case, M3, the number of transitions in the model has more than tripled. Nevertheless, this large number of changes is nicely structured in subprocesses: between 2 and 10 subprocesses were added per log, the largest subprocess had 37 transitions, the average subprocess had 6-13 transitions. We identified alternatives, concurrent actions, and loops in the different subprocesses. Yet, simplification [29] ensured a simple structure in all subprocesses, i.e., graph complexity between 1.0 and 2.0. Model repair also

Table 2: Results on subprocess-based automatic repairs for the logs from Table 1.

	subprocesses			change to original				discover
	#	added $\|T\|$		total $\|T\|$		sim-dist.		sim-dist.
		avg.	max.	add.	rem.	to orig.	to man.	to orig.
M1	7	7	21	69	3	0.144		0.476
M2	5	10	23	65	3	0.147		0.486
M3	10	13	37	151	3	0.199		0.542
M4	8	7	13	71	4	0.139		0.541
M5	6	9	24	60	3	0.143		0.540
$M1^f$	2	6	9	25	4	0.074	.094	0.473
$M2^f$	2	12	21	37	5	0.103	.131	0.539
$M3^f$	7	10	26	87	5	0.164	.176	0.543
$M4^f$	6	7	13	60	4	0.124	.145	0.542
$M5^f$	4	9	23	51	3	0.111	.146	0.541

allowed 25%-30% of the original transitions to be skipped by new τ-transitions; only few original transitions could be removed entirely.

To measure the effectiveness of automatic model repair, we computed the graph similarity distance [33] between automatically repaired model and original model, between automatically repaired model and manually repaired model, and between a completely rediscovered model and the original model. The rediscovered model was obtained with the ILP miner [8] (ensuring fitness) and subsequently simplified by the technique of [29] using the same settings as for subprocess simplification.

We observed that every automatically repaired model is significantly more similar to the reference model (.074-.199) than the rediscovered models are to the reference model (.473-.543). This indicates that model repair indeed takes the original model structure by far more into account than model discovery. The numbers also match the observations one can make when comparing Fig. 1(middle) to Fig. 1(right). The manually repaired models are slightly closer to the reference model in terms of similarity distance (compare Tables 1 and 2), e.g., .094 for automatic repair and .068 for manual repair for $M1^f$. However, the automatic repairs have a different characteristic than the manual repairs as the distance between automatically and manually repaired models is larger than the difference between automatically repaired model and reference model. Filtering logs prior to repair, as discussed in Sect. 3, positively affects the similarity of the repaired model to the original model, i.e., compare similarity of repaired Mi and Mi^f to original. Also, the effect of filtering is stronger when repairing a model than when rediscovering a model, i.e., rediscovered models all had similar distance to the original, regardless of whether the log was filtered.

6.4.2. Improvements by repair variants

We conducted 7 additional experiments varying on (1) whether to repair for structured loops (Alg. 2 in Sect. 5.1), (2) whether to align sublogs (Alg. 7 in Sect. 5.4), (3) whether to compute a global cost function for alignment (Alg. 4 in Sect. 5.3), and (4) whether or not to remove unused nodes from the model (Alg. 3

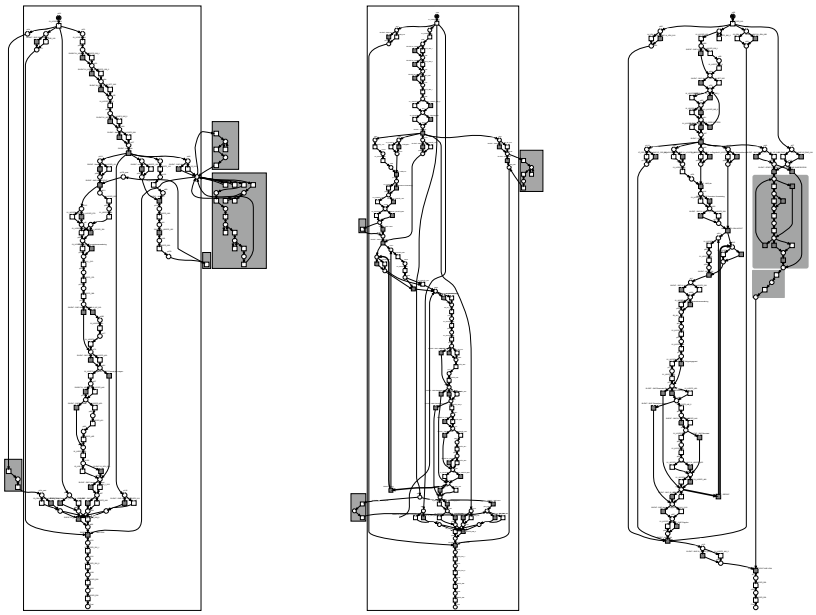


Figure 12: Result of repairing Fig. 1(left) w.r.t. $M2^f$ by aligning sublogs (left), discovering loops (middle), by manual repair (right).

in Sect. 5.2). In all 7 experiments, we grouped sublogs according to Alg. 6 and picked locations according to Alg. 8. Figure 12 compares the result of repairing Fig. 1(left) w.r.t. log $M2^f$ by using aligned sublogs Fig. 12(left) and by discovering loops (middle) with the manually repaired model (right). Here, aligning sublogs splits the large sublog that leads to the large subprocess of Fig. 11(right) into three smaller sublogs, which yield three additional subprocesses in Fig. 12(left). In Fig. 12(middle), the largest sublog was identified as a loop.

The main effects we observed in all logs are the following.

1. Loop detection is effective. Except for log $M1^f$, we could identify and repair loops for each of the logs, ranging between 1 and 4 loops. The smallest loop bodies had 1 transition, the largest loop bodies had between 19 and 23 transitions (for M2, M4, M5, and their filtered variants). The discovered loops were structured (i.e., single entry and single exit) and contained sequences of activities, concurrent activities, and alternative branches; we did not observe structured loops nested insider other loops. We could confirm that the discovered loop bodies coincide with the loops constructed in the manually repaired models.
   For each identified loop, the repaired model had 1 subprocess of the size of the loop body less compared to repairs without loop detection. Thus, a repair with loop discovery added between 27% and 89% of nodes *less* compared to a repair without loops.
   These subprocesses were loop-free except for loops of length 1 or 2, which are introduced by the ILP Miner in case of optional activities. The notable

exception was M3^f, where no large loop could be identified. Here, the repair revealed a rather complex subprocess having unstructured loops (i.e., multiple entries and exits).
This indicates that our technique was capable of detecting all structured loops in the given data set, but cannot handle unstructured loops.

2. Most loops were found when the cost function κ_{loop} of Alg. 9 used costs 0 for model moves. By such a cost function, the alignment can capture loop iterations that contain skipped events. However, we observed a tradeoff. For log M4 and M4^f, some loop iterations contained a large number of skipped actions w.r.t. the loop body. The repair had to add more τ-transitions to the loop body than the corresponding subprocess would have had.
3. Without aligning sublogs, no loops could be identified. We observed that without aligning sublogs, many of the subtraces did contain complete loop iterations, but these were preceded or interleaved with other events. In these cases, Alg. 2 cannot verify the loop hypothesis. When sublogs were aligned, loop iterations were separated from other events and loops could be found.
4. Aligning sublogs also leads to smaller subprocesses that are localized better in the repaired model. The number of subprocesses increased between 2 and 4 for filtered logs and between 5 and 9 for unfiltered logs (log M3 constitutes an extreme case with 23 very small subprocesses).
If loop detection was not used, average subprocess size reduced by 14%-54% (avg. -32%) for filtered logs and by 31%-67% (avg. -42%) for unfiltered logs. If loop detection was used (which always happens in Alg. 9 before aligning sublogs for subprocess discovery), average subprocess size reduced by 8%-54% (avg. -24%) for filtered logs and by 20%-62% (avg. -39%) for unfiltered logs. In particular sublogs which contain iterative behavior contribute to the separation and reduction of sublogs. As sublogs with iterative behavior are not present after loop detection, the reduction effect is slightly less.
5. Computing a global cost function allows to reduce the size of subprocesses that cannot be implemented as loops. In these, we observed a reduction between 17% and 52% for the average subprocess size in filtered logs; in unfiltered logs we observed reductions of up 33% but some subprocesses also increased in size by 1 or 2 transitions.
6. Most superfluous nodes (between 4% and 12% of all nodes) could be found when the alignment was based on a global cost function. No superfluous nodes were found when aligning sublogs of an alignment with a uniform cost function.

Altogether, the results suggest to repair models by the following procedure: first detect loops based on aligned sublogs with a specific cost function that ignores model moves, then add subprocesses based on aligned sublogs that uses a global cost function for alignments, and finally remove superfluous nodes. Algorithm 9 was designed based on these insights. For this algorithm, graph

Table 3: Model quality after direct and after iterative repairs with Alg. 9

sim.-dist	$M1^f$	$M2^f$	$M3^f$	$M4^f$	$M5^f$
direct repair vs. original	0.070	0.073	0.176	0.111	0.099
repeated vs. original	0.070	0.080	0.143	0.148	0.195
repeated vs. direct repair	0.0	0.021	0.144	0.119	0.166

similarity distance to the original model improves by 3% to 31% compared to a plain subprocess-based repair (compare first line of Tab. 3 to Tab. 2).

6.5. Repeated Repairs

Finally, we evaluated how often our automatic repair technique can be applied without requiring human refactoring of the repaired model. For this, we consecutively repaired the reference model first for log $M1^f$ and then the resulting model for $M2^f$ and so on. In each iteration i, we measured the similarity of the repaired model to the model obtained by direct automatic repair w.r.t. log Mi^f and to the original process model. Table 3 shows the results.

We observed that the repeatedly repaired model has a similar distance to the original model as the directly repaired model. Occasionally, the repeatedly repaired model would be even more similar to the original model ($M3^f$). However, after a certain number of iterations, the repeatedly repaired model collects additional repair artifacts, making it less similar to the original than the directly repaired model.

Based on these insights repeated repairs can in principle be applied, depending on the use case. If one is interested in understanding differences to the original model, then the directly repaired model will have a higher quality. In uses cases such as process evolution, the original model will be outdated after the first repair, and the repaired model serves as reference. The results of Tab. 3 suggest that the repaired model should be improved, e.g., in a manual refactoring step, to maintain its quality before the next evolution step is performed.

7. Related Work

The model repair technique presented in this paper largely relates to three research streams: conformance checking of models, iterative process discovery, and changing models to reach a particular aim.

Various conformance checking techniques that relate a given model to an event log have been developed in the past. Their main aim is to quantify *fitness*, i.e., how much the model can replay the log, and if possible to highlight deviations where possible [9–12, 16]. The more recent technique of [9–11] uses alignments to relate log traces to model executions which is a prerequisite for the repair approach presented in this paper. Besides fitness, other metrics [4, 9, 13–15, 17] (*precision*, *generalization*, and *simplicity*) are used to describe how good a model represents reality. Precision and generalization are currently considered in our approach only as a side-effect and are not a guiding factor for model repair. Incorporating these measures into model repair is future work. Simplicity is

considered in our approach in the sense that changes should be as tractable as possible, which we could validate experimentally.

Iterative process discovery techniques solve a problem that is very similar to model repair. The problem is to continuously improve an existing (discovered) process model based on observed behavior. There are 2 basic types of iterative process discovery which both take an existing model N and a log L of most recent process executions as input. The first type, such as the technique proposed in [34], discovers from L a model N_L describing the executions L, and then merges N and N_L into a new model N'. The second type of incremental discovery extracts the ordering relations R_N between activities in N and the ordering relations R_L between events in L and merges them into updated ordering relations R' from which then the incrementally updated process model N' is derived. In [35] R' is derived from the ordering relations with highest support and confidence in R_N and R_L. In [36], R' are the relations of R_N superseded by the relations R_L of the log. In the context of the model repair problem, the approach in [34] discovering N_L from L may result in a too complex model to merge into N, as the model in Fig. 1(right) shows. The same problem occurs for the set of all ordering relations R_L of L in [36]. In contrast, the approach of [35] cannot guarantee a fitting model as support and confidence of original model and log are balanced, possibly neglecting ordering relations of the log. Our approach of separating non-fitting subtraces from fitting traces and repairing the model for non-fitting traces avoids these problems.[2]

A different approach to enforcing similarity of repaired model to original model could be model transformation incorporating an edit distance. The work in [37] describes similarity of process model variants based on edit distance. Another approach to model repair is presented in [38] to find for a given model a most similar sound model (using local mutations). The work in [39] considers repairing incorrect service models based on an edit distance. These approaches do not take the behavior in reality into account. Other approaches to adjust a model to reality, adapt the model at runtime [40, 41], i.e., an individual model is created for each process execution. This paper repairs a model for multiple past executions recorded in a log; it extends on an earlier version by a number of improvements of the model repair techniques including better structured subprocesses and loop detection. The approach of [29, 42] uses observed behavior to structurally simplify a given model obtained in process discovery.

8. Conclusion

This paper addressed, for the first time, the problem of repairing a process model w.r.t. a given log. We proposed a repair technique that preserves the original model structure and introduces subprocesses into the model to permit to replay the given log on the repaired model. We validated our technique on

[2]Unfortunately, the discussed techniques were not available in tool prototypes for experimental evaluation.

real-life event logs and models, and showed that the approach is effective. The repaired model allows to understand how the original model deviated and had to be changed to achieve conformance to the log.

Our proposed technique of model repair covers the entire problem space of model repair between confirming conformance and complete rediscovery. In case of complete fitness, the model is not changed at all. In case of an entirely unfitting model (no synchronous move), the old model is effectively replaced by a rediscovered model. In case of partial fitness, only the non-fitting parts are rediscovered. This allows to apply our technique also in situations where the given model is understood as a *partial model* (created by hand) that is then completed using process discovery on available logs.

The technique can be configured. The cost-function influences the best-fitting alignment found; discovering structured loops, grouping of traces into sublogs, and identifying sublocations for inserting new subprocesses allows for various solutions. Any process discovery algorithm can be used to discover subprocesses; the concrete choice depends on the concrete conformance notion addressed.

In our future work we would like to consider other conformance metrics such as generalization and precision. Moreover, in our current approach we abstract from extra logging information such as the resource executing or initiating the activity and the timestamp of the event. We would like to incorporate this information when repairing the model. For example, resource information can give valuable clues for repair. Finally, we would like to address process model repair also for other perspectives such as the data perspective. Misconformances on data can already be discovered in an alignment-based approach [20].

Acknowledgements. We thank M. Kunze and R.M. Dijkman for providing us with an implementation of the graph similarity distance and the anonymous reviewers for their fruitful suggestions. The research leading to these results has received funding from the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement n° 257593 (ACSI).

References

[1] W. M. P. van der Aalst, Process Mining: Discovery, Conformance and Enhancement of Business Processes, Springer, 2011.

[2] IEEE Task Force on Process Mining, Process Mining Manifesto, in: BPM Workshops, Vol. 99 of LNBIP, Springer, 2012, pp. 169–194.

[3] W. M. P. van der Aalst, A. Weijters, L. Maruster, Workflow Mining: Discovering Process Models from Event Logs, IEEE Transactions on Knowledge and Data Engineering 16 (9) (2004) 1128–1142.

[4] S. Goedertier, D. Martens, J. Vanthienen, B. Baesens, Robust Process Discovery with Artificial Negative Events, Journal of Machine Learning Research 10 (2009) 1305–1340.

[5] A. Medeiros, A. Weijters, W. M. P. van der Aalst, Genetic Process Mining: An Experimental Evaluation, Data Mining and Knowledge Discovery 14 (2) (2007) 245–304.

[6] G. Greco, A. Guzzo, L. Pontieri, D. Saccà, Discovering Expressive Process Models by Clustering Log Traces, IEEE Transaction on Knowledge and Data Engineering 18 (8) (2006) 1010–1027.

[7] J. D. Weerdt, M. D. Backer, J. Vanthienen, B. Baesens, Leveraging Process Discovery With Trace Clustering and Text Mining for Intelligent Analysis of Incident Management Processes, in: IEEE Congress on Evolutionary Computation (CEC 2012), IEEE Computer Society, 2012, pp. 1–8.

[8] J. van der Werf, B. van Dongen, C. Hurkens, A. Serebrenik, Process Discovery using Integer Linear Programming, Fundamenta Informaticae 94 (2010) 387–412.

[9] W. M. P. van der Aalst, A. Adriansyah, B. van Dongen, Replaying History on Process Models for Conformance Checking and Performance Analysis, WIREs Data Mining and Knowledge Discovery 2 (2) (2012) 182–192.

[10] A. Adriansyah, B. van Dongen, W. M. P. van der Aalst, Conformance Checking using Cost-Based Fitness Analysis, in: EDOC 2011, IEEE Computer Society, 2011, pp. 55–64.

[11] A. Adriansyah, B. F. van Dongen, W. M. P. van der Aalst, Towards Robust Conformance Checking, in: BPM 2010 Workshops, Vol. 66 of LNBIP, 2011, pp. 122–133.

[12] T. Calders, C. Guenther, M. Pechenizkiy, A. Rozinat, Using Minimum Description Length for Process Mining, in: SAC 2009, ACM Press, 2009, pp. 1451–1455.

[13] J. E. Cook, A. L. Wolf, Software Process Validation: Quantitatively Measuring the Correspondence of a Process to a Model, ACM Transactions on Software Engineering and Methodology 8 (2) (1999) 147–176.

[14] J. Munoz-Gama, J. Carmona, A Fresh Look at Precision in Process Conformance, in: BPM 2010, Vol. 6336 of LNCS, Springer, 2010, pp. 211–226.

[15] J. Munoz-Gama, J. Carmona, Enhancing Precision in Process Conformance: Stability, Confidence and Severity, in: CIDM 2011, IEEE, Paris, France, 2011, pp. 184–191.

[16] A. Rozinat, W. M. P. van der Aalst, Conformance Checking of Processes Based on Monitoring Real Behavior, Information Systems 33 (1) (2008) 64–95.

[17] J. D. Weerdt, M. D. Backer, J. Vanthienen, B. Baesens, A Robust F-measure for Evaluating Discovered Process Models, in: CIDM 2011, IEEE, 2011, pp. 148–155.

[18] D. Fahland, W. M. P. van der Aalst, Repairing process models to reflect reality, in: BPM 2012, Vol. 7481 of Lecture Notes in Computer Science, Springer, 2012, pp. 229–245.

[19] A. Rozinat, W. M. P. van der Aalst, Decision Mining in ProM, in: International Conference on Business Process Management (BPM 2006), Vol. 4102 of LNCS, Springer, 2006, pp. 420–425.

[20] M. D. Leoni, W. M. P. van der Aalst, B. van Dongen, Data- and Resource-Aware Conformance Checking of Business Processes, in: Business Information Systems (BIS 2012), Vol. 117 of LNBIP, Springer, 2012, pp. 48–59.

[21] J. Buijs, B. van Dongen, W. M. P. van der Aalst, On the Role of Fitness, Precision, Generalization and Simplicity in Process Discovery, in: OTM Federated Conferences, 20th International Conference on Cooperative Information Systems (CoopIS 2012), Vol. 7565 of LNCS, Springer, 2012, pp. 305–322.

[22] M. Song, W. M. P. van der Aalst, Towards Comprehensive Support for Organizational Mining, Decision Support Systems 46 (1) (2008) 300–317.

[23] C. Günther, A. Rozinat, W. M. P. van der Aalst, Activity Mining by Global Trace Segmentation, in: BPM 2009 Workshops, Proceedings of the Fifth Workshop on Business Process Intelligence (BPI'09), Vol. 43 of LNBIP, Springer, 2010, pp. 128–139.

[24] R. P. J. C. Bose, W. M. P. van der Aalst, Abstractions in Process Mining: A Taxonomy of Patterns, in: Business Process Management (BPM 2009), Vol. 5701 of LNCS, Springer, 2009, pp. 159–175.

[25] A. Rozinat, I. de Jong, C. Günther, W. M. P. van der Aalst, Process Mining Applied to the Test Process of Wafer Scanners in ASML, IEEE Transactions on Systems, Man and Cybernetics, Part C 39 (4) (2009) 474–479.

[26] V. Levenshtein, Binary Codes Capable of Correcting Deletions, Insertions, and Reversals, Soviet Physics-Doklady 10 (8) (1966) 707–710.

[27] R. P. J. C. Bose, W. M. P. van der Aalst, Trace Alignment in Process Mining: Opportunities for Process Diagnostics, in: Business Process Management (BPM 2010), Vol. 6336 of LNCS, Springer, 2010, pp. 227–242.

[28] A. Adriansyah, J. Munoz-Gama, J. Carmona, B. Dongen, W. M. P. van der Aalst, Alignment Based Precision Checking, in: Workshop on Business Process Intelligence (BPI 2012), Vol. 132 of LNBIP, Springer, 2012, pp. 137–149.

[29] D. Fahland, W. M. Aalst, Simplifying Discovered Process Models in a Controlled Manner, Information Systems.
URL http://dx.doi.org/10.1016/j.is.2012.07.004

[30] R. P. J. C. Bose, W. M. P. van der Aalst, Trace Clustering Based on Conserved Patterns: Towards Achieving Better Process Models, in: BPM 2009 Workshops, Proceedings of the Fifth Workshop on Business Process Intelligence (BPI'09), Vol. 43 of LNBIP, Springer, 2010, pp. 170–181.

[31] R. P. J. C. Bose, W. M. P. van der Aalst, Process diagnostics using trace alignment: Opportunities, issues, and challenges, Inf. Syst. 37 (2) (2012) 117–141.

[32] R. P. J. C. Bose, W. M. P. van der Aalst, Analysis of Patient Treatment Procedures, in: BPM Workshops'11, Vol. 99 of LNBIP, 2011, pp. 165–166.

[33] R. M. Dijkman, M. Dumas, L. García-Bañuelos, Graph Matching Algorithms for Business Process Model Similarity Search, in: BPM, Vol. 5701 of LNCS, 2009, pp. 48–63.

[34] E. Kindler, V. Rubin, , W. Schäfer, Incremental Workflow Mining based on Document Versioning Information, in: Unifying the Software Process Spectrum (SPW 2005), Revised Selected Papers, Vol. 3840 of LNCS, Springer, 2005, pp. 387–301.

[35] A. Kalsing, G. S. do Nascimento, C. Iochpe, L. H. Thom, An Incremental Process Mining Approach to Extract Knowledge from Legacy Systems, in: EDOC'2010, IEEE, 2010, pp. 79–88.

[36] W. Sun, T. Li, W. Peng, T. Sun, Incremental workflow mining with optional patterns and its application to production printing process, International Journal of Intelligent Control and Systems 12 (1) (2007) 45–55.

[37] C. Li, M. Reichert, A. Wombacher, Discovering Reference Models by Mining Process Variants Using a Heuristic Approach, in: BPM 2009, Vol. 5701 of LNCS, Springer, 2009, pp. 344–362.

[38] M. Gambini, M. L. Rosa, S. Migliorini, A. H. M. ter Hofstede, Automated Error Correction of Business Process Models, in: BPM 2011, Vol. 6896 of LNCS, Springer, 2011, pp. 148–165.

[39] N. Lohmann, Correcting Deadlocking Service Choreographies Using a Simulation-Based Graph Edit Distance, in: BPM 2008, Vol. 5240 of LNCS, Springer, 2008, pp. 132–147.

[40] S. W. Sadiq, W. Sadiq, M. E. Orlowska, Pockets of flexibility in workflow specification, in: ER'2001, Vol. 2224 of LNCS, 2001, pp. 513–526.

[41] M. Reichert, P. Dadam, ADEPTflex-Supporting Dynamic Changes of Workflows Without Losing Control, JIIS 10 (2) (1998) 93–129.

[42] D. Fahland, W. M. P. van der Aalst, Simplifying Mined Process Models: An Approach Based on Unfoldings, in: Business Process Management (BPM 2011), Vol. 6896 of LNCS, Springer, 2011, pp. 362–378.