Process Model Repair by Detecting Unfitting Fragments*

Alexey A. Mitsyuk[1], Irina A. Lomazova[1], Ivan S. Shugurov[2], and Wil M.P. van der Aalst[3,1]

[1] National Research University Higher School of Economics,
20 Myasnitskaya ul., 101000 Moscow, Russia.
{amitsyuk,ilomazova}@hse.ru
[2] Technical University of Munich,
Arcisstraße 21, Munich, Bavaria, 80333, Germany.
shugurov94@gmail.com
[3] Eindhoven University of Technology,
P.O. Box 513, NL-5600 MB, Eindhoven, The Netherlands.
w.m.p.v.d.aalst@tue.nl

Abstract. Process models often do not adequately reflect the behavior of real-life systems. In the general case, it is possible to construct a new adequate model by applying one of the discovery algorithms. At the same time, there are cases when the original model is of particular value. In such cases, it is better to apply model repair algorithms. Those algorithms construct a model which reflects real behavior according to some criteria. Moreover, the repaired model remains as similar to the original one as possible. This paper proposes a modular approach which consists of three parts: (1) decomposing the process model and event log into model fragments and sub-logs, (2) selecting the fragments which need to be repaired, (3) repairing the selected fragments using a process discovery algorithm.

Keywords: process mining, process model repair, process model decomposition, Petri nets, divide and conquer

1 Introduction

Modern process-aware information systems can be designed using model-driven software engineering approaches. However, the exact implementation of design models in real-life systems is a rare case. Moreover, processes tend to change during the system's life-cycle. A process owner usually wants relevant, up-to-date models describing the process. In this paper we suggest a method, which can *repair* a model.

Real behavior of a system can be studied by analyzing its *event logs*. One can *discover* the model of a real system from them [3]. Moreover, process engineers

* This work is supported by the Basic Research Program of the National Research University Higher School of Economics.

can diagnose the discrepancies between observed (event logs) and modeled (process models) behaviors using *conformance checking* techniques [3]. It is possible to use the results of conformance checking as an input for model repair [8].

Process model repair aims at improving the quality of a model (according to some quality criteria), while changing as few of its parts as possible. The *general model repair problem* can be defined as follows. The initial model N is a Petri net. The event log L does not conform to N according to some predefined criteria. The general repair problem is to find such a model N^r (repaired model) that L *better* conforms N^r according to this criteria and N^r is as similar to N as possible.

In this paper, we propose a new method for model repair. Our approach is based on the idea of *patch-up*s. We find the non-conforming fragments in a model and replace them with conforming ones. This paper describes the constraints, as well as pros and cons of this approach. We propose a general modular scheme which can be implemented using different algorithms. In this paper, we show one possible implementation of fitness-aware repair. The general scheme employs the *divide & conquer* concept.

2 Related Work

One of the first papers in the field of process model repair was published in 2011 [8], where the authors formulated the problem of repair. Later in [10] they focused on fitness-aware control-flow repair. Models here are labeled workflow nets. The approach separates non-conforming and conforming behaviors from the event log. Non-conforming traces are then grouped into non-conforming sub-logs, and for each of them the corresponding sub-process workflow model is discovered. Then these sub-process are added to the initial model in such a way that the repaired model successfully replays the given event log. This approach also allows to remove infrequently used parts of the model, and hence to make the repaired model more simple. The problem of model *simplification* is close to model repair. [9, 17] present a method for simplifying the model structure in such a way that it can still replay most of the behavior described in the log. Using frequency metrics for model simplification is also described in [11].

Another view on process repair, called *impact-driven* repair, is introduced in [16]. This procedure can be applied when a number of repair operations is limited, and various repairs have different values. Based on the notion of alignments [4], the paper suggests assigning costs to change operations. The approach investigates the space of all possible repairs. Authors proposed and evaluated a set of algorithms which can seek optimal (w.r.t. to assigned costs for repair operations) repairs. In other words, the repair for Petri nets has been reduced to an optimization problem. Plenty of experiments were conducted to select the best algorithm with suitable parameters.

One more contribution to the field of model repair is [5], where the well-structured process models (process trees) are improved w.r.t. the four classical quality metrics: fitness, precision, generalization, and simplicity. The authors use

a special genetic discovery algorithm, and pay attention to similarity between the repaired and original models.

In this paper, we use the *divide & conquer* principle which has already been employed within process mining. The core papers on it are [1, 2] and [20]. In [2] the author considers the task of model and log decompositions in context of process mining. The practical questions of applying the *divide & conquer* principle in *process mining* are considered in [20]. In particular, the author proposed a modular scheme for process discovery. Hompes et al. [12] investigated the so-called *re-composition* techniques. These methods help to select the most appropriate size of decomposition fragments during process discovery [13]. Model and event log decompositions have been used for boosting the performance of conformance checking [15].

In contrast to the existing approaches, this paper presents model repair based on model decomposition, which allows keeping the initial model structure unchanged as much as possible. This is important when a source model is readable and well structured, i.e. it was developed by experts. If there were just minor *local* changes in the process, we can repair the model by correcting its local fragments.

3 Preliminaries

Multi-sets, Functions, and Sequences. Let $\mathbb{N}$ denote the set of natural numbers (including zero). A multi-set over a set S is a function $b : S \to \mathbb{N}$. By $\mathcal{B}(S)$ we denote the set of all multi-sets over S. Further by $b = [e, e, e, c, d, d] = [e^3, c, d^2]$ we designate the multi-set over $S = \{e, c, d\}$, where $b(e) = 3, b(c) = 1, b(d) = 2$. By abuse of notation, we extend set operations to multi-set in the standard way.

For a function $f\colon X \to Y$, $dom(f)$ denotes its domain, and $f\colon X \nrightarrow Y$ denotes a partial function. A function $f{\restriction_Q}$ is a *function projection* of a (partial) function f onto a set $Q \subseteq X$, iff $dom(f{\restriction_Q}) = dom(f) \cap Q$ and $\forall x \in dom(f{\restriction_Q})\colon f{\restriction_Q}(x) = f(x)$. This notation can be extended to multi-sets, e.g. $[e^3, c, d^2] \restriction_{\{c,d\}} = [c, d^2]$.

By X^* we denote the set of all finite sequences over a set X, and we use triangle brackets for sequences, e.g. $\sigma = \langle x_1, x_2, ..., x_n \rangle$ is a sequence of length n. By $\sigma_1 \cdot \sigma_2$ we denote the concatenation of two sequences, $\sigma{\restriction_Q}$ is the projection of sequence σ onto the set Q.

Process Models. In this paper, a process model is a labeled workflow net. A Petri net is a triple (P, T, F), where P and T are disjoint sets of *places* and *transitions*, and $F\colon (P \times T) \cup (T \times P) \to \mathbb{N}$ is a flow relation. For a transition $t \in T$ a *preset* $\bullet t$ and a *postset* $t\bullet$ are defined as the subsets of P such that $\bullet t = \{p | F(p, t) \neq 0\}$ and $t\bullet = \{p | F(t, p) \neq 0\}$. A labeled Petri net is a tuple (P, T, F, l), where $l\colon T \to \mathcal{U}_A \cup \{\tau\}$ is a labeling function, which maps transitions to activity labels from $\mathcal{U}_A$. A transition t is called invisible, if $l(t) = \tau$, otherwise it is called visible. A *marking* of a Petri net is a function $M\colon P \to \mathbb{N}$, i.e. a multi-set of places. A transition $t \in T$ is *enabled* in a marking M iff $\forall p \in P\ M(p) \geq$

$F(p,t)$. An enabled transition t may *fire* yielding a new marking M', such that $M'(p) = M(p) - F(p,t) + F(t,p)$ for each $p \in P$ (denoted $M \xrightarrow{t} M'$).

A *workflow net (WF-net)* $N = (P, T, F, l)$ is a labeled Petri net, such that
(1) there is one source place $i \in P$ such that $\bullet i = \emptyset$, and $M_i = [i]$,
(2) there is one sink place $o \in P$ such that $o\bullet = \emptyset$, and $M_o = [o]$,
(3) every node $n \in P \cup T$ is on a path from i to o.

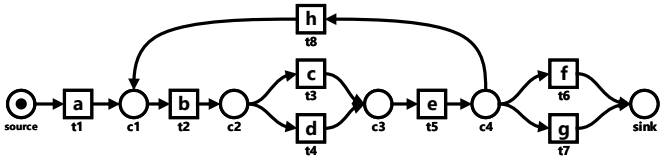

Fig. 1: The Model of Compensation Requests Processing *(activities have the following labels: a = Register request; b = Check ticket; c = Examine casually; d = Examine thoroughly; e = Decide; f = Send rejection letter; g = Pay compensation; h = Re-initiate request)*

An example of a WF-net is shown in Figure 1. This model is a variation of the conventional process mining example which can be found in [3]. It represents processing of compensation requests in a company.

We will also use the following notations. $T_v(N)$ is a set of all labeled (visible) transitions in WF-net N: $l(t \in T_v) \neq \tau$. $T_v^u(N)$ is a set of visible transitions in WF-net N with unique labels, that is $\forall t, t' \in T_v^u : l(t) \neq l(t')$ iff $t \neq t'$. The union of two WF-nets is a WF-net $N^U = N^1 \cup N^2$, which is built by the union of sets of places, transitions, and flows: $N^1 \cup N^2 = (P^1 \cup P^2, T^1 \cup T^2, F^1 \cup F^2, l^u)$, where $l^u \in (T^1 \cup T^2) \nrightarrow \mathcal{U}_A$ is a union of l^1 and l^2, $dom(l^u) = dom(l^1) \cup dom(l^2)$, $l^u(t) = l^1(t)$ if $t \in dom(l^1)$, and $l^u(t) = l^2(t)$ if $t \in dom(l^2) \setminus dom(l^1)$.

Event Logs. Let $A \subseteq \mathcal{U}_A$ be a set of *activities*. A *trace* σ is a finite sequence of activities from A, i.e. $\sigma \in A^*$. An *event log* L is a finite multi-set of traces, i.e. $L \in \mathcal{B}(A^*)$, i.e. $L = \left[\langle a,b,c,d,e\rangle^3, \langle a,b,a,d,e\rangle^5, \langle a,d,c,b,e\rangle\right]$ is an event log. A projection of an event log $L = [\sigma_1, \sigma_2, \ldots, \sigma_n]$ onto a set of activities B is a log $L\restriction_B = [\sigma_1\restriction_B, \sigma_2\restriction_B, \ldots, \sigma_n\restriction_B]$, where $\sigma_i\restriction_B$ is a projection of the trace σ_i onto B.

Let N be a WF-net with transition labels from A, an initial marking $[i]$, and a final marking $[o]$. Let σ be a trace over A. We say that a trace $\sigma = a_1, \ldots, a_k$ *perfectly fits* N iff there exists a sequence of firings $[i] = m_0 \xrightarrow{t_1} \ldots \xrightarrow{t_k} m_{k+1} = [o]$ in N, s.t. the sequence of activities $\lambda(t_1), \lambda(t_2), \ldots, \lambda(t_k)$ after deleting all invisible activities coincides with σ. A log L *perfectly fits* N iff every trace from L perfectly fits N. E.g. the following traces perfectly fit the model in Figure 1: $[\langle a,b,c,e,f\rangle, \langle a,b,c,e,h,b,d,e,g\rangle, \langle a,b,d,e,g\rangle]$.

4 Modular Repair Scheme

We consider the following repair problem: given a workflow net N and an event log L, which does not perfectly fit N, to construct a model N^r (repaired model),

so that L *perfectly* fits N^r. We propose a modular approach for model repair. In this paper, we have deliberately limited the scope of its application but it is also applicable to the general model repair problem.

The main idea here is to define a general description for a class of model repair algorithms, based on the *divide & conquer* principle. This scheme can be refined into different repair algorithms by choosing appropriate methods as its actual procedural parameters. The repair scheme consists of building blocks, each of which executes one of the steps. A graphical representation of the model repair scheme is shown in Figure 2. Building blocks are shown with black frames. The following blocks are present: (1) Decomposition, (2) Projection, (3) Selection, (4) Repair, (5) Composition, (6) Evaluation. Each block contains an applicable algorithm. Arcs show the data transfer between building blocks.

The input of the scheme is a pair (L, N) where L is a normative event log and N is a non-conforming initial model. Model N is decomposed into several fragments using one of decomposition algorithms. In the projection block, an algorithm splits the event log into sub-logs which correspond to the model fragments. The selection block contains a conformance checking algorithm. This algorithm calculates the conformance level for each pair (L^i, N^i). According to this number, all model fragments are separated into two sets. The first one contains conforming (*good*) fragments. The second one consists of model fragments which do not conform corresponding sub-logs (*bad*). The repair block contains an algorithm which somehow replaces non-conforming model fragments by conforming ones. The composition block is paired with the decomposition one. The result can be evaluated in the evaluation block by using any of the conformance checking methods. This is a general scheme.

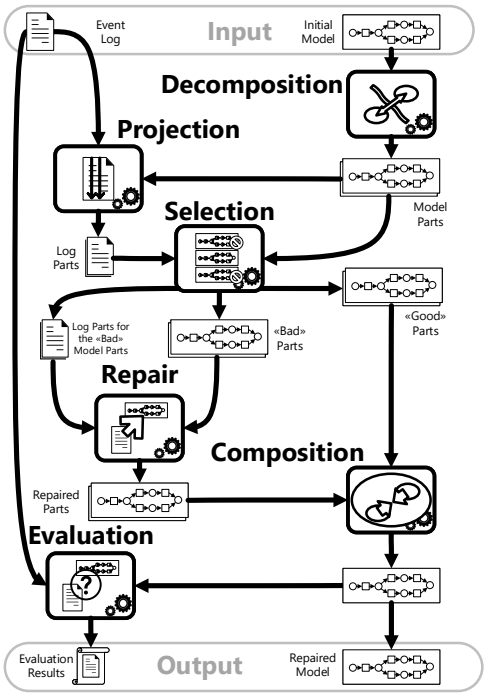

Fig. 2: Modular Repair Scheme

Definition 1 (Modular Repair Scheme). *Let $\mathcal{U}_A$ denote the universal set of activities, and $\mathcal{U}_N$ be the set of all WF-nets with transitions labeled over $\mathcal{U}_A$.*

The modular repair scheme is a procedure, which takes an event log $L \in \mathcal{B}(\mathcal{U}_A^*)$ and a WF-net $N \in \mathcal{U}_N$ as an input and outputs a repaired model N^r, which perfectly fits L, i.e. $N^r = ModularRepair(L, N)$.

The modular repair scheme has four procedural parameters eval, split, repair, and compose, where $eval \in (\mathcal{B}(\mathcal{U}_A^*) \times \mathcal{U}_N) \to [0;1]$ *is an evaluation function,* $repair \in \mathcal{B}(\mathcal{U}_A^*) \times \mathcal{U}_N) \to \mathcal{U}_N$ *is a repair function,* $split \in \mathcal{U}_N \to \mathcal{P}(\mathcal{U}_N)$ *is a decomposition function which is always paired with a related composition function* $compose \in \mathcal{P}(\mathcal{U}_N) \to \mathcal{U}_N$.

In Section 5, we propose a specific set of algorithms which can be used to implement the scheme. Obviously, these algorithms cannot be chosen arbitrarily. The main constraint comes from the requirement of *valid decompositions* [2]. Valid decomposition assumes that the only elements, which can be shared between different fragments, are visible transitions with unique (in the initial net) labels. We slightly modify the definition in [2] for our purpose.

Definition 2 (Valid Decomposition for WF-nets). *Let* $N \in \mathcal{U}_N$ *be a WF-net with a labeling function* l. $D = \{N^1, N^2, \ldots, N^n\} \subseteq \mathcal{U}_N$ *is a valid decomposition if and only if*
- $N^i = (P^i, T^i, F^i, l^i)$ *is a Petri net for all* $1 \leq i \leq n$,
- $l^i = l\restriction_{T^i}$ *for all* $1 \leq i \leq n$,
- $P^i \cap P^j = \emptyset$ *for* $1 \leq i < j \leq n$,
- $T^i \cap T^j \subseteq T_v^u(N)$ *for* $1 \leq i < j \leq n$, *and*
- $N = \bigcup_{1 \leq i \leq n} N^i$.

$\mathcal{D}(N)$ *is the set of all valid decompositions of* N.

It is proved in [2] that each valid decomposition can be used to decompose both the *process discovery* and *conformance checking*. It means that one can split the model into several fragments using valid decomposition and then unambiguously compose the initial model from model fragments using the corresponding composition function. More formally, the following statements are valid.

Theorem 1 (Checking for Perfect Fitness Can Be Decomposed [2]). *Let* $L \in \mathcal{B}(A^*)$ *be an event log with* $A \subseteq \mathcal{U}_A$ *and let* $N \in \mathcal{U}_N$ *be a WF-net. For any valid decomposition* $D = \{N^1, N^2, \ldots, N^n\} \in \mathcal{D}(N)$*: L is perfectly fitting WF-net N if and only if for all* $1 \leq i \leq n$*:* $L\restriction_{A_v(N^i)}$ *is perfectly fitting* N^i.

Theorem 2 (Discovery can be decomposed [2]). *Let* $L \in \mathcal{B}(A^*)$ *be an event log with* $A = \{a \in \sigma \mid \sigma \in L\} \subseteq \mathcal{U}_A$, $\mathcal{C} = \{A^1, A^2, \ldots, A^n\}$ *with* $A = \bigcup \mathcal{C}$, *and* $disc \in \mathcal{B}(\mathcal{U}_A^*) \to \mathcal{U}_N$ *a discovery algorithm. Let* $distr\_disc(L, \mathcal{C}, disc) = \{N^1, N^2, \ldots, N^n\}$ *and* $N = \bigcup_{1 \leq i \leq n} N^i$. $N \in \mathcal{U}_N$ *and* $D = \{N^1, N^2, \ldots, N^n\}$ *is a valid decomposition of* N, $distr\_disc(L, \mathcal{C}, disc) \in \mathcal{D}(N)$.

Using these results we can formulate the conditions for fitness repair.

Definition 3 (*Perfect Fitness* Repair for WF-nets). *Let* $L \in \mathcal{B}(A^*)$ *be an event log with* $A \subseteq \mathcal{U}_A$, *and let* $N \in \mathcal{U}_N$ *be a WF-net such that* $fitness(L, N) < 1$. *Let* $N' = ModularRepair(L, N)$ *be a modular repair scheme. ModularRepair is a* perfect fitness *repair if* $fitness(L, N') = 1$.

Proposition 1 (Sufficient Conditions for Perfect Fitness Repair). *ModularRepair is a* perfect fitness *repair if*

1. *split is a valid decomposition;*
2. *repair is a perfect discovery, for any event log it makes perfectly fitting model;*
3. *compose is a transition fusion which merge all transitions with equal labels.*

Proof. This proposition is a direct corollary of the Theorem 1. Let N be a WF-net model with $\mathit{fitness}(L, N) < 1$. By applying *split* N is decomposed into a set of fragments $D = \{N^1, N^2, \ldots, N^n\}$. Each fragment N^i is then evaluated, let $f_i = \mathit{fitness}(L{\restriction}_{A_v(N^i)}, N^i)$. By Theorem 1 $\exists j{:}\, 1{\leq}j{\leq}n, \mathit{fitness}(L{\restriction}_{A_v(N^j)}, N^j) < 1$, since the decomposition is valid. Fragments with not perfect fitness should be repaired. Then repair function *repair* is applied to each such fragment. For each i, $1{\leq}i{\leq}n$ let $N_r^i = \mathit{repair}(L{\restriction}_{A_v(N^i)}, N^i)$ if $\mathit{fitness}(L{\restriction}_{A_v(N^i)}, N^i) < 1$, and $N_r^i = N^i$ if $\mathit{fitness}(L{\restriction}_{A_v(N^i)}, N^i) = 1$. Since we use a perfect discovery function, we get $\forall i{:}\, 1{\leq}i{\leq}n$, $\mathit{fitness}(L{\restriction}_{A_v(N_r^i)}, N_r^i) = 1$. Note that the set of activities was not changed. By Theorem 2 the set $D_r = \{N_r^1, N_r^2, \ldots, N_r^n\}$ is a valid decomposition of the net $N_r = \mathit{compose}(D_r)$. Moreover, each fragment perfectly fits the corresponding event log projection. Hence, by Theorem 1 we get $\mathit{fitness}(L, N_r) = 1$. □

5 Modular Repair using Maximal Decomposition, Inductive and ILP Miners

The proposed general scheme can use specific algorithms as building blocks. In particular, the *maximal decomposition* is a valid decomposition [2]. Maximal decomposition is based on partitioning of net edges and is defined as follows. Each edge resides strictly in a single fragment. Transitions with unique labels are placed on borders of fragments. This is needed to support the causal dependencies between fragments. As a consequence, each transition, which has a unique label in the original net, will reside in two or more fragments. And finally, splitting saves an initial marking of the net. A place in a fragment is marked iff it was marked in the original net. It is important to mention that for a given system net there can be one and only one maximal decomposition. Maximally decomposed net can be easily composed back by fusion of transitions with the same labels. *Maximal decomposition is valid* by construction [2].

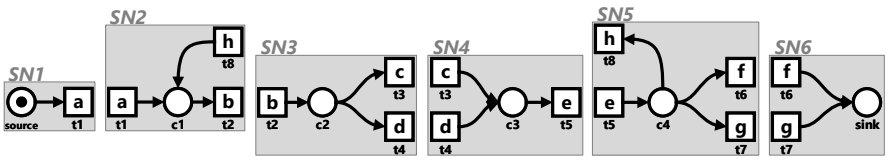

Fig. 3: A Maximal Decomposition of the Model from Figure 1

We developed Algorithm 1 for constructing the maximal decomposition. It works as follows. Function `decompose` starts from an arbitrary place of initial net. It calls `handlePlace` function while there are places to traverse. The `handlePlace` function is needed to start the recursive procedure `handlePlaceRecursively`. This procedure traverses successor and predecessor nodes of a

Algorithm 1 Maximal Decomposition Construction

Input: $N = (P, T, F, M_{init}, M_{final})$ — a Petri net with markings
Output: DN — a decomposition (set of Petri nets)

function DECOMPOSE(N)
  $HandledPlaces \leftarrow emptymap$; $DN \leftarrow \emptyset$;
  for all $p \in P$ **do**
    if $p \notin keys(HandledPlaces)$ **then**
      $N' = (P', T', F', M'_{init}, M'_{final})$; ▷ make new empty fragment
      $HandledPlaces \leftarrow HandledPlaces \cup$ HANDLEPLACE(N, p, N');
      $marking(N') \leftarrow CopyMarking(N)$;
      $DN \leftarrow DN \cup \{N'\}$;
    end if
  end for
  return DN;
end function
function HANDLEPLACE(N, p, N')
  $PDict, TDict \leftarrow emptymap$; ▷ mapping of nodes from original net to a current fragment
  HANDLEPLACERECURSIVELY(N, p, N', $PDict$, $TDict$);
  return $PDict$;
end function
function HANDLEPLACERECURSIVELY(N, p, N', $PDict$, $TDict$)
  if $p \in keys(PDict)$ **then**
    return ;
  end if
  $PDict[p] \longleftarrow createPlace(N', label(p))$; ▷ create new place in N' with label of p
  HANDLEPRECEDINGTRANSITIONS(N, p, N', $PDict$, $TDict$);
  HANDLESUBSEQUENTTRANSITIONS(N, p, N', $PDict$, $TDict$);
end function
function HANDLE(PRESEDING/SUBSEQUENT)TRANSITIONS(N, p, N', $PDict$, $TDict$)
  for all $f \in (incoming/outgoing)(N, p)$ **do**
    $t' \longleftarrow null$; $t \leftarrow (source/target)(f)$;
    if $t \in keys(TDict)$ **then**
      $t' \longleftarrow TDict[t]$; ▷ a transition has already been handled
    else
      $t' \longleftarrow createTransition(N', label(t), visible(t))$;
      $TDict[t] \longleftarrow t'$; ▷ new transition
    end if
    $addArc(N', (t'/PDict[p]), (PDict[p]/t'))$;
    if $\neg visible(t')$ **then**
      for $f' \in (incoming/outgoing)(N, t)$ **do**
        HANDLEPLACERECURSIVELY(N, $(source/target)(f')$, N', $PDict$, $TDict$);
      end for
    end if
  end for
end function

place. The algorithm splits the net fragments if it finds visible transition. Note, that all visible transitions in the initial net are assumed to have unique labels. The proof of this algorithm correctness is straightforward and rather technical, so we omit it here. Figure 3 shows the result of decomposing the example model (Figure 1) according to this algorithm. It was decomposed into six net fragments. Note, that the result is always unique.

A number of process discovery algorithms have been proposed in the literature [11, 14, 22]. In this paper, we use the *Inductive miner* [14]. This algorithm guarantees perfect fitness of a discovered model when executed with particular settings (*inductive miner infrequent* with *zero noise threshold*). It uses sequential (and recursive) inductive inferences to build the so-called process trees, which can be simply translated into well-structured workflow nets. Another discovery

algorithm, which we also use, is *ILP (Integer Linear Programming)*-based algorithm [22]. It also guarantees perfect fitness with particular settings. Further we show experiments with both algorithms.

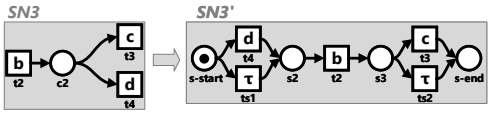

Fig. 4: This Model is Discovered by Inductive Miner

There have been published a lot of papers on log-model conformance evaluation [4, 7, 15]. Among the main quality dimensions are *fitness* and *precision* [3]. Fitness measures the proportion of traces in an event log which can be successfully replayed by the model. All traces from a perfectly fitting log can be replayed. Precision shows the proportion of model runs which have corresponding traces in an event log. That is, perfect precision means the model allows only the behavior presented in the log.

We consider *fitness* as the main criteria for evaluation of repair results. For fitness checking we use *alignment-based fitness evaluation function* (cf. [4] for details). Let us consider an example.

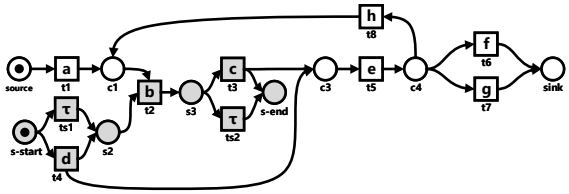

Fig. 5: The Petri Net after Transition Fusion

Suppose, the model shown in Figure 3 does not correspond exactly to the current behavior in terms of the order of implementation of events with labels b and d. Now in contrast to the behavior of the model d always precedes b if both appear in the trace. Namely, our log contains traces $\langle a, d, b, e, f\rangle$ and $\langle a, b, c, e, g\rangle$, and there are no traces in which b precedes d. Then by applying the alignment-based fitness evaluation algorithm to model fragments shown in Figure 3 we obtain that fragment $SN3$ does not perfectly fit the log. So, we discover a correct WF-net for this fragment using Inductive miner (see Figure 4).

After discovery of the correct model fragment and fusion of all fragments we obtain the net shown in Figure 5. This model perfectly fits the log with traces $\langle a, d, b, e, f\rangle$ and $\langle a, b, c, e, g\rangle$. One can easily see that repaired model is not a WF-net. It contains tokens in places *source* and *s-start* in initial marking. The final marking for the net consists of two tokens in places *sink* and *s-end*. However, this model can be easily transformed into an equivalent WF-net by adding invisible *start* and *finish* transitions. Note, that we have removed places *s-start* and *s-end*, since they restrict the model behavior and may generate a deadlock. However, this generates another problem — increases the number of possible model runs, and hence and reduces the model precision. To avoid such problems it is better not to change fragment border nodes during the repair transformations. This can be done by considering larger fragments.

6 Evaluation

We have implemented the approach as a plug-in for ProM Framework [21] which is a well-known tool in the process mining community. We already described some details of implementation in [19]. In order to evaluate our method we selected several artificial WF-nets. For these models we generated event logs using the tool Gena [18]. Then the initial models were slightly changed. These updated models were repaired using our approach. Selected results are shown in Table 1.

Table 1: Selected Experimental Results

Model	Method	Miner	Fitness	Precision	Sim	N-f	N-bf	Size	Time
SM1	Initial Model		1	0,91	-	-	-	40	-
	Discovery	Ind	1	0,91	0,51	-	-	47	829
		ILP	1	0,91	0,57	-	-	38	10069
SM1-BL-1	Changed Model		0,94	0,86	-	-	-	40	-
	Repair	Ind	1	0,57	0,97	19	1	40	2531
		ILP	1	0,57	0,97	19	1	40	2531
SM1-BL-2	Changed Model		0,89	0,89	-	-	-	40	-
	Repair	Ind	1	f	0,89	19	3	45	2983
		ILP	1	0,5	0,94	19	3	39	3215
SM1-BNL-1	Changed Model		0,84	0,91	-	-	-	40	-
	Repair	Ind	f	f	0,75	19	4	75	3363
		ILP	f	f	0,89	19	4	41	5310
LM2	Initial Model		1	0,59	-	-	-	133	-
	Discovery	Ind	1	f	f	-	-	166	2920
		ILP	1	0,53	f	-	-	137	1816898
LM2-BL-1	Changed Model		0,98	0,59	-	-	-	133	-
	Repair	Ind	1	0,28	f	63	2	133	10745
		ILP	1	0,28	f	63	2	133	9588
LM2-BL-2	Changed Model		0,99	0,59	-	-	-	133	-
	Repair	Ind	1	0,38	f	63	1	133	8048
		ILP	1	0,38	f	63	1	133	12847
LM2-BL-3	Changed Model		0,97	0,59	-	-	-	133	-
	Repair	Ind	1	0,23	f	63	3	133	8097
		ILP	1	0,23	f	63	3	133	8855
LM2-BNL-1	Changed Model		0,98	0,59	-	-	-	133	-
	Repair	Ind	1	f	f	63	4	142	8300
		ILP	1	0,19	f	63	4	130	9823

Two models `SM1` and `LM2` were selected. Both models are larger than the example which is shown in Figure 1. One can find the number of nodes in each model in `Size` column. `Fitness` and `Precision` columns show the corresponding metrics (calculated using technique from [4]). The `Sim` column contains similarity evaluation results. *Calculate Graph Edit Distance Similarity* ProM plug-in [12] was used for calculating similarity. In turn, this plug-in is based on the theoretical backgrounds presented in [6]. The method returns a number between 0 (models are completely different) and 1 (models are equal). This plug-in takes into consideration not only the structure of both models but also node labels. The `N-f` column shows the overall number of decomposition fragments. A number of changed (repaired) fragments is shown in `N-bf` column. We show `Time` in milliseconds[4]. However, we understand the relativity of this calculation. It is also interesting to compare the results for repair and direct discovery. For comparison, we used two discovery algorithms which provide perfect fitness: `Ind`uctive

[4] Test configuration: Intel Core i7-3630QM, 2.40 GHz; 4 GB RAM; Windows 7 x64

and ILP miners. BL models were changed locally — we replaced neighbor transitions, which were connected via one or two places. BNL models contain more substantial changes — we replaced distant transitions.

The results in Table 1 were obtained using the deletion of sink and source places. An implementation of the algorithm for alignments calculation consumes a lot of memory in such cases, as it checks all possible behaviors of the model. In some cases the memory of our machine (4GB) was too small. These cases are marked with f in corresponding cell. Existing algorithm for similarity calculation also failed in some cases. These are cases, when a repaired model contains many silent transitions. Again, the number of possible combinations is too large.

Table 1 shows how our repair approach provided perfectly fitting models. However, we need to improve the precision of repaired models. In cases with local changes (BL) the approach shows acceptable results. It is interesting that the repair using ILP is faster than direct discovery (even with all the overhead on checking the fragments' fitness).

7 Conclusion

This paper proposes a modular approach for process model repair. The approach can be instantiated using different algorithms as building blocks. The main idea is to use model decomposition and then replace unfitting model fragments instead of complete re-discovery. A proposed approach has been evaluated on several artificial process models.

The method we propose allows to keep the initial model structure as much as possible. This is crucial when the initial model is of some special value, i.e. it has a clear structure and/or was developed by human experts, who will continue working with it. The experiments we made show that repairing a model based on its decomposition can be very helpful. However, there are still open problems and questions.

First, our approach significantly reduces the precision of the model. To cope with this problem, we plan to propose more subtle decomposition techniques to avoid problems with fragment border nodes. The re-composition method [12] may be also helpful for improving the quality of obtained models.

Second, we plan to compare the presented approach with other existing model repair methods. Moreover, the approach evaluation on real-life cases is needed. It would be also interesting to consider decomposition based model repair taking into account a combination of fitness, precision, generalization, and simplicity metrics, and to study the applicability of this method to the cases of non-local changes.

Acknowledgments. The authors thank the anonymous reviewers for important and helpful remarks and suggestions.

References

1. van der Aalst, W.M.P.: Decomposing Process Mining Problems Using Passages. In: Haddad, S., Pomello, L. (eds.) PETRI NETS 2012. LNCS, vol. 7347, pp. 72–91. Springer-Verlag, Berlin (2012)
2. van der Aalst, W.M.P.: Decomposing Petri Nets for Process Mining: A Generic Approach. Distributed and Parallel Databases 31(4), 471–507 (2013)
3. van der Aalst, W.M.P.: Process Mining - Data Science in Action, Second Edition. Springer (2016)
4. Adriansyah, A.: Aligning observed and modeled behavior. Ph.D. thesis, Technische Universiteit Eindhoven (2014)
5. Buijs, J.C.A.M., La Rosa, M., Reijers, H.A., van Dongen, B.F., van der Aalst, W.M.P.: Improving Business Process Models Using Observed Behavior. In: Cudre-Mauroux, P., Ceravolo, P., Gasevic, D. (eds.) SIMPDA 2012. LNBIP, vol. 162, pp. 44–59. Springer-Verlag, Berlin (2013)
6. Dijkman, R., Dumas, M., van Dongen, B.F., Käärik, R., Mendling, J.: Similarity of Business Process Models: Metrics and Evaluation. Inf. Syst. 36(2), 498–516 (2011)
7. van Dongen, B.F., Carmona, J., Chatain, T.: A unified approach for measuring precision and generalization based on anti-alignments. In: BPM. LNCS, vol. 9850, pp. 39–56. Springer (2016)
8. Fahland, D., van der Aalst, W.M.P.: Repairing Process Models to Reflect Reality. In: Barros, A., Gal, A., Kindler, E. (eds.) In: Proc. of the BPM 2012. LNCS, vol. 7481, pp. 229–245. Springer (2012)
9. Fahland, D., van der Aalst, W.M.P.: Simplifying discovered process models in a controlled manner. Inf. Syst. 38(4), 585–605 (2013)
10. Fahland, D., van der Aalst, W.M.P.: Model Repair - Aligning Process Models to Reality. Inf. Syst. 47, 220–243 (2015)
11. Günther, C.W., van der Aalst, W.M.P.: Fuzzy mining: Adaptive process simplification based on multi-perspective metrics. BPM 2007, pp.328–343., Springer (2007)
12. Hompes, B.F.A.: On Decomposed Process Discovery: How to Solve a Jigsaw Puzzle with Friends. Master's thesis, Eindhoven University of Technology (2014)
13. Hompes, B.F.A., Verbeek, H.M.W., van der Aalst, W.M.P.: Finding suitable activity clusters for decomposed process discovery. In: SIMPDA 2014. vol. 1293, pp. 16–30. CEUR-WS.org (2014)
14. Leemans, S.J.J., Fahland, D., van der Aalst, W.M.P.: Discovering block-structured process models from incomplete event logs. In: Ciardo, G., Kindler, E. (eds.) PETRI NETS 2014, LNCS, vol. 8489, pp. 91–110. Springer (2014)
15. Munoz-Gama, J.: Conformance Checking and Diagnosis in Process Mining - Comparing Observed and Modeled Processes, LNBIP, vol. 270. Springer (2016)
16. Polyvyanyy, A., van der Aalst, W.M.P., ter Hofstede, A.H.M., Wynn, M.T.: Impact-driven process model repair. ACM TOSEM. 25(4), 28:1–28:60 (2017)
17. de San Pedro, J., Carmona, J., Cortadella, J.: Log-based simplification of process models. In: BPM. LNCS, vol. 9253, pp. 457–474. Springer (2015)
18. Shugurov, I.S., Mitsyuk, A.A.: Generation of a Set of Event Logs with Noise. In: Proc. of the SYRCoSE 2014. pp. 88–95 (2014)
19. Shugurov, I.S., Mitsyuk, A.A.: Iskra: A Tool for Process Model Repair. Proceedings of the Institute for System Programming of the RAS 27(3), 237–254 (2015)
20. Verbeek, H.M.W.: Decomposed Process Mining with Divide And Conquer. In: BPM 2014 Demos, vol. 1295, pp. 86–90. CEUR-WS.org (2014)

21. Verbeek, H.M.W., Buijs, J.C.A.M., van Dongen, B.F., van der Aalst, W.M.P.: ProM 6: The Process Mining Toolkit. In: La Rosa, M. (ed.) Proc. of BPM Demo Track 2010. CEUR-WS.org, vol. 615, pp. 34–39 (2010)
22. van der Werf, J.M.E.M., van Dongen, B.F., Hurkens, C.A.J., Serebrenik, A.: Process discovery using integer linear programming. Fundam. Inform. 94(3-4), 387–412 (2009)