Scalable Conformance Checking of Business Processes

Daniel Reißner[1], Raffaele Conforti[1], Marlon Dumas[2], Marcello La Rosa[1], and Abel Armas-Cervantes[1]

[1] Queensland University of Technology, Australia
{da.reissner, raffaele.conforti, m.larosa, a.armascervantes}@qut.edu.au
[2] University of Tartu, Estonia
marlon.dumas@ut.ee

Abstract. Given a process model representing the expected behavior of a business process and an event log recording its actual execution, the problem of business process conformance checking is that of detecting and describing the differences between the process model and the log. A desirable feature is to produce a minimal yet complete set of behavioral differences. Existing conformance checking techniques that achieve these properties do not scale up to real-life process models and logs. This paper presents an approach that addresses this shortcoming by exploiting automata-based techniques. A log is converted into a deterministic automaton in a lossless manner, the input process model is converted into another minimal automaton, and a minimal error-correcting synchronized product of both automata is calculated using an A* heuristic. The resulting automaton is used to extract alignments between traces of the model and traces of the log, or statements describing behavior observed in the log but not captured in the model. An evaluation on synthetic and real-life models and logs shows that the proposed approach outperforms a state-of-the-art method for complete conformance checking.

Key words: Conformance checking, Process Mining, Automata, Behavioral Alignment

1 Introduction

Modern information systems maintain detailed business process execution trails. For example, an enterprise resource planning system keeps records of key events related to a company's order-to-cash process, such as the receipt and confirmation of purchase orders, the delivery of products, and the creation and payment of invoices. Such records can be grouped into an *event log* consisting of sequences of events (called *traces*), each consisting of all event records pertaining to one case of a process.

Conformance checking techniques exploit such event logs in order to determine if and to what extent the actual behavior of a process conforms to a process model capturing its expected behavior. A conformance checking technique takes as input an event log and a process model, and returns a set of differences between the model and the log. In real-life scenarios, the set of differences between an event log and a process model can be large. Hence it is necessary to represent them in a way that is compact and interpretable, yet complete, or as exhaustive as desired by the user.

State-of-the-art techniques for computing a complete set of differences include *behavioral alignment* [15] and *(all-optimal) trace alignment* [2]. The former computes a set of statements describing behavioral relations that exist in the model but not in the log. The latter computes minimal alignments between each trace in the log that cannot be parsed by the model, and a corresponding trace that can be parsed by the model. These techniques however do not scale up to large and noisy event logs. For example, our experimental evaluation (reported later) shows that the all-optimal trace alignment technique in [2] takes more than five minutes to compute an incomplete set of alignments over real-life and noisy logs and sometimes does not converge after hours. These execution times make it impractical to use these techniques in an interactive setting, e.g. when conformance checking is performed multiple times to iteratively repair a process model so as to better fit the log. Additionally, scalability issues of conformance checking techniques indirectly affect several process mining techniques, such as model repair [25, 7] or process discovery [8, 17], which rely on conformance checking to justify the quality of their outputs.

This paper aims to tackle this scalability issue by proposing an automata-based technique for conformance checking. In our approach, an event log is encoded as sequences of words and compressed into a minimal Deterministic Acyclic Finite State Automaton (DAFSA). Concomitantly, the process model is transformed into another automaton (its *reachability graph*). The two automata are combined into an error-correcting product automaton whose transitions correspond to either a synchronous move (on both automata) or an asynchronous move (i.e. a move on the automaton of the log that does not exist in the model or vice-versa). The produced automaton contains a minimal number of asynchronous moves. From this product automaton, we can extract either the optimal alignments of each trace in the log and a corresponding trace in the model (as in [2]) or a set of behavioral difference statements (as in [15]). Thus, our approach unifies these two previous approaches, while achieving higher scalability, as shown by an evaluation on synthetic and real-life process models and logs.

The next section discusses existing conformance checking techniques in more detail. Section 3 introduces the proposed approach, while Section 4 presents its evaluation. Section 5 summarizes the contributions and discusses improvement avenues.

2 Related Work

Conformance checking techniques detect two types of discrepancies between a process model and a log: behavior observed in the log that is disallowed by the model (*unfitting behavior*), and behavior allowed by the model but not observed in the log (*additional behavior*). A simple approach to detect and measure unfitting behavior is *token-based replay* [26]. The idea is to replay each trace against the model, represented as a Petri net. The transitions in the model are fired following the order dictated by a given trace. To fire, a transition needs to be enabled, i.e. it requires at least one token in each of its incoming places. When a transition cannot fire because it is not enabled, the technique determines which tokens need to be *added* to enable it. Once a trace has been replayed, if there are any tokens left in a non-sink place of the Petri net, they are labeled as *remaining tokens*. The *fitness* between the model and the log is quantified in terms of the number of added and remaining tokens (replay errors). An extended version of this approach, namely *continuous semantics fitness* [4], achieves higher performance at the expense of incompleteness. Another extension [32] decomposes the model into single-entry single-exit fragments, such that each fragment can be replayed independently. Other extensions based on model decomposition are discussed in [23].

Replay fitness methods fail to identify a minimum number of errors required to explain unfitting log behavior, thus overestimating the magnitude of differences. Trace alignment fitness [2] addresses this limitation. For each trace in the log, this technique identifies the closest trace reproducible by the model and aligns the two traces by highlighting the points where mismatches occur. This log-model alignment is achieved in several steps. The first step consists in transforming every trace in the log into a Petri net. The result is a sequence of transitions, one per event in the trace. Next, a product is computed between the Petri net of the trace and the Petri net of the model. This is done by pairing transitions of the two models that have matching labels. The product between the two Petri nets is used to create a transition system representing all possible alignments, i.e. matches and mismatches. This transition system is explored, using the A^* search algorithm, to retrieve the alignments with the minimum number of mismatches. An exhaustive and complete version of this technique, namely *all-optimal alignments*, computes all minimal alignments between each log trace and the model. *One-optimal alignment* is an alternative technique that achieves higher scalability at the expense of incompleteness. This technique computes only one alignment for each log trace, hence missing on some behavioral differences. Several heuristics-based approaches, such as sequential prefix alignments [29] or decomposing trace replay technique [27], improve on the scalability for identifying alignments. Those approaches, however, drop the guarantee to find the optimal alignments and thus trade accuracy for performance.

Approaches for identifying additional behavior include *negative-events precision* [31] and *ETC precision* [22]. The former adds negative events to the traces in the log. Given a trace, an event is negative if it is never observed after a given trace prefix. Additional behavior is identified by replaying these extended traces over the model. Whenever a negative-event is successfully replayed, the approach marks it as additional behavior. ETC precision generates a prefix automaton from the log, where each state corresponds to a distinct trace prefix in the log. The states of the prefix automaton are matched with the states of the model. When a state in the model enables a transition that it is not enabled in the matching state of the automaton, it is marked as additional behavior. The approach has been extended in [1] to handle tasks with duplicate labels and unfitting traces by means of trace alignment. The technique proposed in this paper is complementary the above ones, since it computes trace alignments that can be used for example to speed up the technique in [1].

An approach for fast approximate computation of fitness and precision metrics is presented in [18]. This technique computes these metrics over subsets of process tasks and aggregates the results at a process level. This technique has been shown to be highly scalable, however, it does not identify the behavioral differences between the model and the log, but it merely computes the fitness and precision metrics. As part of this paper, we are interested in a complete list of exact differences, hence a comparison with the approach presented in [18] is out of scope.

Another conformance checking technique, namely *behavioral alignment* [15], addresses the problems of detecting unfitting behavior and additional behavior in a unified setting. In this technique, both the input event log and the process model are transformed into event structures. A minimal error-correcting product of these two event structures is then computed. Based on this product, a set of statements are derived, which characterize all behavioral relations between tasks captured in the model but not observed in the log and vice-versa. While producing a complete set of differences, which is smaller in number than the number of trace alignments, this technique suffers from similar scalability requirements as the all-optimal alignment.

The approach herein presented uses automata as novel and memory-efficient representation for event logs and process models. By mapping the problem of conformance checking to that of synchronizing a DAFSA representing the event log, and a finite state machine (FSM) representing the model, the proposal unifies the techniques proposed in [15] and [2]. This allows us to extract both a set of optimal trace alignments and a set of difference statements. Thus, the paper aims at improving the efficiency of state of the art conformance checking techniques leveraging automata and memoization techniques. Unlike [15] though, we only focus on detecting unfitting behavior.

3 Approach

Figure 1 outlines the steps of the proposed method and their respective inputs and outputs. First, the input process model is expanded into a reachability graph (1). In parallel, the event log is compressed into a minimal DAFSA (2). The resulting reachability graph and DAFSA are then compared (3) to create an error-correcting synchronized product automaton (herein called a Partial Synchronized Product or PSP), wherein each state is a pair of a state in the reachability graph and a state in the DAFSA. From this result, we can directly enumerate a set of optimal trace alignments or derive a set of behavioral difference statements via further analysis (4). The rest of this section introduces some preliminary definitions, followed by a description of each of the steps.

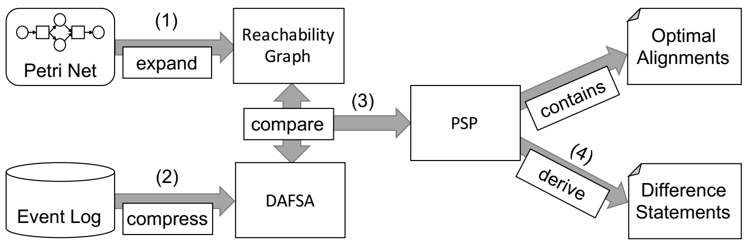

Fig. 1. Overview of the approach.

For illustration purposes, we will use the loan application process model displayed in Fig. 2. The process starts when a credit application is received, then the credit history and the income sources are checked. Then, once the application is assessed, either a credit offer is made, the application is rejected or additional information is requested (the latter leading to a re-assessment).

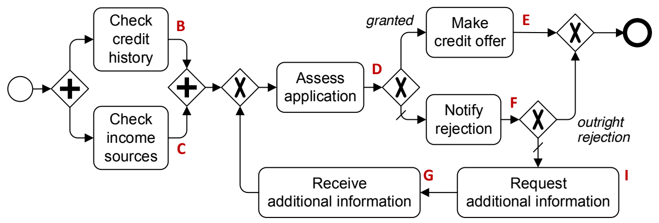

Fig. 2. Example loan application process model adapted from [15].

3.1 Preliminaries

Our approach relies on the notion of finite state machine defined as follows.

Definition 1 (Finite State Machine (FSM)). *Let L be a finite non-empty set of labels. A finite state machine is a directed graph $\mathscr{F} = (N, \mathrm{A}, s, R)$, where N is a finite non-empty set of states, $\mathrm{A} \subseteq N \times L \times N$ is a set of arcs, $s \in N$ is an initial state, and $R \subseteq N$ is a set of final states.*

An arc in a FSM is a triplet (n_s, l, n_t), where n_s is the *source* state, n_t is the *target* state and l is the *label* associated to the arc. The set of incoming and outgoing arcs of a state n is defined as $\blacktriangleright n = \{(n_s, l, n_t) \in \mathrm{A} \mid n = n_t\}$ and $n\blacktriangleright = \{(n_s, l, n_t) \in \mathrm{A} \mid n = n_s\}$, respectively. Finally, a sequence of (contiguous) arcs in a FSM is called a *path*.

3.2 From Event Log to DAFSA

Logs recording the execution of activities in a business process are called *event logs*. These logs represent the executions of process instances as *traces* – sequences of activity occurrences (*a.k.a. events*). A trace can be represented as a sequence of labels, such that each label signifies an event. Generally speaking an event log is a multiset of traces containing several occurrences of the same trace. However, in the context of this paper, we are only interested in the distinct executions of a business process and, therefore, we define a log as a set of traces.

Definition 2 (Trace and event log). *Let L be a finite set of labels. A* trace *is a finite sequence of labels $\langle l_1, \ldots, l_n \rangle \in L^*$, such that $l_i \in L$ for any $1 \leq i \leq n$. An* event log *$\mathscr{L}$ is a set of traces.*

Event logs can be represented as *Deterministic Acyclic Finite State Automata* (DAFSA), which are acyclic and deterministic FSMs. A DAFSA can represent words, in our context *traces*, in a compact manner by exploiting prefix and suffix compression.

Definition 3 (DAFSA). *A DAFSA is an acyclic and deterministic finite state machine $\mathscr{D} = (N_{\mathscr{D}}, \mathrm{A}_{\mathscr{D}}, s_{\mathscr{D}}, R_{\mathscr{D}})$, where $N_{\mathscr{D}}$ is a finite non-empty set of states, $\mathrm{A}_{\mathscr{D}} \subseteq N_{\mathscr{D}} \times L \times N_{\mathscr{D}}$ is a set of arcs, $s_{\mathscr{D}} \in N_{\mathscr{D}}$ is the initial state, $R_{\mathscr{D}} \subseteq N_{\mathscr{D}}$ is a set of final states.*

Daciuk et al. [11] presents an efficient algorithm for constructing a DAFSA from a set of words, such that every word is a path from the initial state to a final state. Conversely it holds, that every path from an initial state to a final state represents a word present in the given set of words. We reuse this algorithm to construct a DAFSA from an event log, where every trace in the log represents a word. The complexity of building the DAFSA is $O(|L| \cdot \log n)$, where L is the set of distinct event labels, and n is the number of states in the DAFSA.

Given a path from the initial state to a state $n \in N_{\mathscr{D}}$, we refer to the labels associated to the arcs in the path as the *prefix* of n, and, analogously, given a path from n to a final state, we refer to the labels associated to such path as a suffix of n. Note that the prefix of the initial state is $\{\langle\rangle\}$. By abuse of notation, the set of prefixes of a state n is represented by $\mathit{pref}(n) = \bigcup_{(n_s,l,n_t) \in \blacktriangleright n} \{x \oplus l \mid x \in \mathit{pref}(n_s)\}$, where $\oplus$ denotes the concatenation operator. Similarly, the set of suffixes of n is represented by $\mathit{suff}(n) = \bigcup_{(n_s,l,n_t) \in n\blacktriangleright} \{l \oplus x \mid x \in \mathit{suff}(n_t)\}$, and if n is a final state then $\{\langle\rangle\} \in \mathit{suff}(n)$. Prefixes and suffixes are said to be *common* iff they are shared by more than one trace.

Definition 4 (Common prefixes and suffixes). *Let $\mathscr{D} = (N_{\mathscr{D}}, \mathrm{A}_{\mathscr{D}}, s_{\mathscr{D}}, R_{\mathscr{D}})$ be a DAFSA. The set of common prefixes of $\mathscr{D}$ is the set $\mathscr{P} = \{\mathit{pref}(n) \mid n \in N_{\mathscr{D}} \wedge |n\blacktriangleright| > 1\}$. The set of common suffixes of $\mathscr{D}$ is the set $\mathscr{S} = \{\mathit{suff}(n) \mid n \in N_{\mathscr{D}} \wedge |\blacktriangleright n| > 1\}$.*

Log
$\langle B,D,E\rangle$
$\langle C,B,D,E\rangle$
$\langle B,D,E\rangle$
$\langle C,B,D,F\rangle$
$\langle B,D,F\rangle$

$\Rightarrow$

Common Prefixes:	**Common Suffixes:**
$\langle B,D\rangle$, $\langle C,B,D\rangle$	$\langle D,F\rangle$, $\langle D,E\rangle$

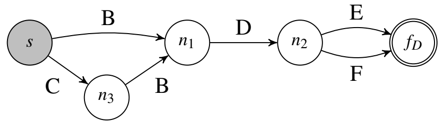

Fig. 3. Example log for our loan application process, and its DAFSA representation.

Figure 3 depicts an example of an log containing activities of the loan application process in Fig. 2 and its corresponding DAFSA representation. For the sake of readability, Fig. 3 uses the letters next to the each of the tasks in Fig. 2 as task labels. In this example there is only one final state f_D, and all traces in the log are paths from s to f_D. For instance, the trace $\langle B,D,E\rangle$ is represented by the path $\langle (s,B,n_1),(n_1,D,n_2),(n_2,E,f_D)\rangle$. In this example, the prefixes of state n_2 and the suffixes of state n_1 are common for all the traces.

3.3 From a process model to a FSM

Process models are normative descriptions of business processes and define the expected behavior of the process. Over the years, several business process modelling languages have been proposed, such as Petri nets, BPMN and EPC. In the context of this work, business processes are modelled as (labelled) Petri nets.

Definition 5 (Labelled Petri net). *A (labelled) Petri net is the tuple $\mathscr{N} = (P,T,F,\lambda)$, where P and T are disjoint sets of places and transitions, respectively, $F \subseteq (P \times T) \cup (T \times P)$ is the flow relation, and $\lambda : T \to L \cup \{\tau\}$ is a labelling function mapping transitions to the set of task labels L and to a special label τ.*

Note that τ is a special label and it is used to represent invisible transitions, i.e. actions not recorded in the event log when executed. Places and transitions are conjointly referred to as nodes. A node x is in the preset of a node y if there is a transition from x to y and, conversely, a node z is in the postset of y if there is a transition from y to z. Then, the preset of a node y is the set $\bullet y = \{x \in P \cup T | (x,y) \in F\}$ and the postset of y is the set $y\bullet = \{z \in P \cup T | (y,z) \in F\}$. A marking m is a multiset of places representing a state during the execution of a system. A transition $t \in T$ is *enabled* at a marking m iff $\bullet t \subseteq m$. An enabled transition t can *fire* and yield a new marking $m' = m - \bullet t + t\bullet$. The reachability graph [21] of a Petri net $\mathscr{N}$ with an initial marking m_0 contains all possible markings of $\mathscr{N}$ – denoted as M. Intuitively, a reachability graph is a deterministic FSM where states denote markings, and arcs denote the transitions fired to go from one marking to another. The complexity of constructing a reachability graph is at worst exponential on the size of the Petri net [19], i.e. $O(2^{|P \cup T|})$.

Definition 6 (Reachability graph). *The* reachability graph *of a Petri net $\mathscr{N}$ is a deterministic finite state machine $\mathscr{R} = (M, \mathrm{A}_{\mathscr{R}}, m_0, M_f)$, where M is the set of reachable markings, $\mathrm{A}_{\mathscr{R}}$ is the set of arcs $\mathrm{A}_{\mathscr{R}} = \{(m_1, \lambda(t), m_2) \in M \times L \times M \mid m_2 = m_1 - \bullet t + t\bullet\}$ and $M_f = \{m \in M \mid \nexists t \in T$, such that $\bullet t \subseteq m\}$.*

Algorithm 1: Remove Tau Transitions

```
input: Reachability Graph $\mathscr{R}$
$\sigma \leftarrow \langle m_0 \rangle$;
$\Omega \leftarrow \{m_0\}$;
while $\sigma \neq \langle\rangle$ do
    $m \leftarrow$ head $\sigma$ [1];
    $\sigma \leftarrow$ tail $\sigma$ [2];
    $\Psi \leftarrow \{a = (m_1, l, m) \in \blacktriangleright m \mid l = \tau \wedge m \notin M_f\}$;
    for $a \in \Psi$ do replaceTau$(a, m, \{m\})$ ;
    $\mathrm{A}_{\mathscr{R}} \leftarrow \mathrm{A}_{\mathscr{R}} \setminus \Psi$;
    for $(m, l, m_2) \in m\blacktriangleright \mid m_2 \notin \Omega$ do
        $\sigma \leftarrow \sigma \oplus m_2$;
        $\Omega \leftarrow \Omega \cup \{m_2\}$;
$\Xi \leftarrow \{m \in M \mid (\blacktriangleright m = \varnothing \wedge m \neq m_0) \vee (m\blacktriangleright = \varnothing \wedge m \notin M_f)\}$;
while $\Xi \neq \varnothing$ do
    for $m \in \Xi$ do $\mathrm{A} \leftarrow \mathrm{A} \setminus (\blacktriangleright m \cup m\blacktriangleright)$ ;
    $M \leftarrow M \setminus \Xi$;
    $\Xi \leftarrow \{m \in M \mid (\blacktriangleright m = \varnothing \wedge m \neq m_0) \vee (m\blacktriangleright = \varnothing \wedge m \notin M_f)\}$;
return $\mathscr{R}$;
Function replaceTau$((m_1, \tau, m_t) \in \mathrm{A}, m \in M, \Theta \in 2^M)$
    for $(m, l, m_2) \in m\blacktriangleright$ do
        if $l \neq \tau \vee m_2 \in M_f$ then $\mathrm{A}_{\mathscr{R}} \leftarrow \mathrm{A}_{\mathscr{R}} \cup \{(m_1, l, m_2)\}$ ;
        else if $m_2 \notin \Theta$ then
            $\Theta \leftarrow \Theta \cup \{m_2\}$;
            replaceTau$((m_1, \tau, m_t), m_2, \Theta)$;
```

A large amount of τ-transitions in a Petri net can lead to large reachability graphs. In principle, we assume that the Petri nets have a minimal number of τ-transitions, e.g.,

[1] *head* in Z notation [14] to obtain the first element of a sequence.
[2] *tail* in Z notation [14] to obtain a subsequence after the first element of a sequence.

resulting from the application of reduction rules in [24]. However, oftentimes some τ-transitions cannot be removed because they represent the "skip" or parallel execution of transitions. In this regard, we propose a further τ-reduction over the reachability graph that does not modify the underlying behavior. Algorithm 1 shows the top-down approach for the proposed reduction. Intuitively, for each arc $a = (m_1, \tau, m_2)$ referring to a τ-transition, the algorithm replaces a with $a' = (m_1, l, m_3)$ for each outgoing arc of m_2, such that $(m_2, l, m_3) \in \mathrm{A}_{\mathscr{R}}$. This replacement is repeated until all arcs referring to τ-transitions are removed. If all incoming arcs of a state m are replaced, then m and its outgoing arcs are removed. The algorithm refrains from removing τ transitions targeting final markings to ensure proper completion. Fig. 4 shows the τ-less reachability graph of the loan application process aside. Observe that the arc $[p5, p4] \rightarrow [p6]$ is replaced by $[p5, p4] \rightarrow [p7]$ with label D, and the state $[p3, p2]$ is removed.

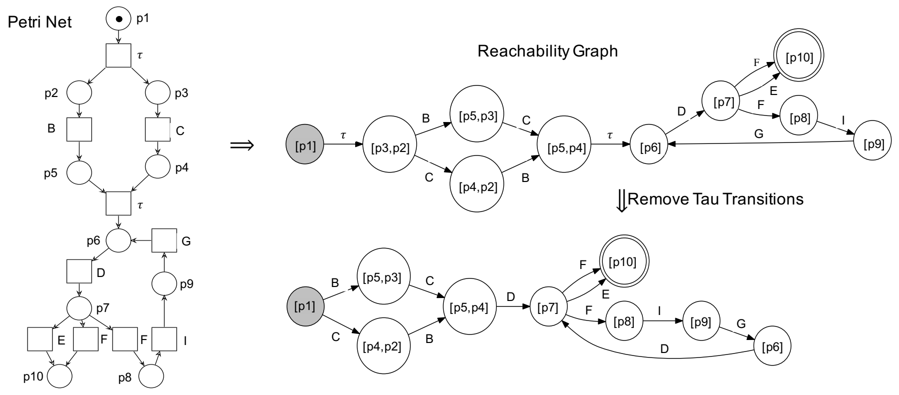

Fig. 4. Petri net obtained from the BPMN model in Fig. 2, and its tau-less reachability graph.

3.4 Error-correcting synchronised product

The computation of similar and deviant behavior between an event log and a process model is based on an error-correcting synchronized product (a.k.a. PSP) [5]. Intuitively, the traces represented in the DAFSA are "aligned" with the executions of the model by means of three operations: (1) synchronized move (*match*), the process model and the event log can execute the same task/event w.r.t. label; (2) log operation (*lhide*), an event observed in the log cannot occur in the model; and (3) model operation (*rhide*), a task in the model can occur, but the corresponding event is missing in the log.

Both a trace in a log and an execution represented in a reachability graph are totally ordered sets of events (sequences). Then, an alignment aims at *matching* events from both sequences that represent the same tasks w.r.t. their labels, such that the order between the matched events is preserved. For example, given a trace in a log $\langle B, D, E \rangle$ and an execution in a model $\langle D, B, E \rangle$, it is possible to match the events with label E, and either the events with label B or the events with label D, but not both. An event that is not matched has to be hidden using the operation *lhide* if it belongs to the log, or *rhide* if it belongs to an execution in the model.

In our context, the alignments are computed over a pair of finite state machines, a DAFSA and a reachability graph, therefore the three operations: *match*, *lhide* and *rhide*, are applied over the arcs of both FSMs. An operation applied over a pair of arcs (one in the DAFSA and one in the reachability graph) is called a *synchronization*. Note that *lhide* and *rhide* are applied only over one arc, thus we use $\perp$ to denote the absence of the other element in the triplet.

Definition 7 (Synchronization). *Let* $\mathrm{A}_{\mathscr{D}}$ *and* $\mathrm{A}_{\mathscr{R}}$ *be the arcs in the DAFSA and in the reachability graph, respectively. A synchronization* β *is a triplet* $\beta \subseteq op \times \mathrm{A}_{\mathscr{D}} \times \mathrm{A}_{\mathscr{R}}$*, where* $op \in \{match, lhide, rhide\}$*. The set of all synchronizations is denoted as* S*.*

All possible alignments between the traces represented in a DAFSA and the executions represented in a reachability graph can be computed inductively as follows. The construction starts by pairing the initial states of both FSMs and then applying the three defined operations over the events that can occur in the DAFSA and in the reachability graph – each application of the operations (synchronization) yield a new pairing of states. Note that the alignments between (partial) traces and executions are implicitly computed as sequences of synchronizations. Then, an alignment is defined as follows.

Definition 8 (Alignment). *Given a set of synchronizations* S*, an alignment is defined as* $\varepsilon = \langle \beta_1, ..., \beta_n \rangle$ *with* $\beta_i \in S, 1 \leq i \leq n$*. All the possible alignments are denoted as* $\mathscr{C}$*.*

Given an alignment $\varepsilon = \langle \beta_1, \beta_2, \ldots, \beta_m \rangle$, we use $\hat{\varepsilon}$ to denote the aligned trace in the log, i.e., $\hat{\varepsilon} = \langle l_1, l_2, \ldots, l_n \rangle$, such that for any l_i, l_j, where $1 \leq i < j \leq n$, there exist $\beta_x = (op_x, (b_s, l_i, b_t), a_2)$ and $\beta_y = (op_y, (b_v, l_j, b_w), a_3)$ in ε, where $1 \leq x < y \leq m$, $op_x \in \{match, lhide\}$ and $op_y \in \{match, lhide\}$. In case there exists β_x but no β_y, $\hat{\varepsilon} = \langle l_x \rangle$, and in case there exists no β_x, $\hat{\varepsilon}$ is the empty sequence $\langle \rangle$. Thus, $\hat{\varepsilon}$ is a sequence of log task labels, that have been aligned in ε with *match* or *lhide* operations.

All alignments can be collected in a finite state machine called PSP [5]. Every state in the PSP is a triplet (n, m, ε), where n is a state in the DAFSA, m is a state in the reachability graph and ε is the (partial) alignment of the events occurred at n and m; every arc is a synchronization; the pairing of the initial states is the initial state; and the finial states are those with no outgoing arcs.

Definition 9 (PSP). *Given a DAFSA* $\mathscr{D}$ *and a reachability graph* $\mathscr{R}$*, their PSP* $\mathscr{P}$ *is a finite state machine* $\mathscr{P} = (N_{\mathscr{P}}, \mathrm{A}_{\mathscr{P}}, s_{\mathscr{P}}, R_{\mathscr{P}})$*, where* $N_{\mathscr{P}} \subseteq N_{\mathscr{D}} \times M \times \mathscr{C}$ *is the set of nodes,* $\mathrm{A}_{\mathscr{P}} = N_{\mathscr{P}} \times S \times N_{\mathscr{P}}$ *is the set of arcs,* $s_{\mathscr{P}} = (s_{\mathscr{D}}, m_0, \langle \rangle) \in N_{\mathscr{P}}$ *is the initial node, and* $R_{\mathscr{P}} = \{f \in N_{\mathscr{P}} \mid f\blacktriangleright = \varnothing\}$ *is the set of final nodes.*

The PSP contains all possible alignments, however we are interested in those containing the minimum amount of hides for each trace in the log. These alignments are called *optimal*. The computation of all optimal alignments can become infeasible when the search space is too large. Thus, we use an A^* algorithm [16] to consider the most promising paths in the PSP first, i.e., those minimizing the number of hides. Given an event log $\mathscr{L}$, the resulting PSP is complete and minimal, since it contains only the optimal alignments for every trace $c \in \mathscr{L}$. The cost function for our A^* algorithm is $\rho(x, c) = g(x) + h(x, c)$, where x is a node in the PSP and c is a trace in the log.

Algorithm 2: Construct the PSP

```
input: Event Log $\mathscr{L}$, DAFSA $\mathscr{D}$, Reachability Graph $\mathscr{R}$
for $c \in \mathscr{L}$ do
    $\sigma \leftarrow \{(s_{\mathscr{P}}, \rho(s_{\mathscr{P}}, c))\}$;
    $\rho_{max} \leftarrow \mid c \mid +$ minModelSkips;
    while $\sigma \neq \varnothing$ do
        choose a tuple $(n_{act} = (n_{\mathscr{D}}, m, \varepsilon), \rho) \in \sigma$, such that $\nexists (n'_{\mathscr{P}}, \rho') \in \sigma : \rho > \rho'$;
        $\sigma \leftarrow \sigma \setminus \{(n_{act}, \rho)\}$;
        if $n_{\mathscr{D}} \in R_{\mathscr{D}} \wedge m \in R_{\mathscr{R}} \wedge \hat{\varepsilon} = c$ then
            if $\rho(n_{act}, c) < \rho_{max}$ then
                $\rho_{max} \leftarrow \rho(n_{act}, c)$;
                $Opt \leftarrow \varnothing$;
                $\sigma \leftarrow \{(n, \rho(n,c)) \in \sigma \mid \rho(n,c) \leq \rho_{max}\}$
            $Opt \leftarrow Opt \cup \{n_{act}\}$;
        else
            $n_{new} \leftarrow \varnothing$;
            for $\alpha_{\mathscr{D}} = (n_{\mathscr{D}}, l_{\mathscr{D}}, n_t) \in n_{\mathscr{D}}\blacktriangleright \mid l_{\mathscr{D}} = c(|\{\beta = (op, a_{\mathscr{D}}, a_{\mathscr{R}}) \in \varepsilon \mid op \neq rhide\}| + 1)^4$ do
                $n_{new} \leftarrow n_{new} \cup \{(n_t, m, \varepsilon \oplus (lhide, \alpha_{\mathscr{D}}, \bot))\}$;
                for $\alpha_{\mathscr{R}} = (m, l_{\mathscr{R}}, m_t) \in m\blacktriangleright \mid l_{\mathscr{R}} = l_{\mathscr{D}}$ do
                    $n_{new} \leftarrow n_{new} \cup \{(n_t, m_t, \varepsilon \oplus (match, \alpha_{\mathscr{D}}, \alpha_{\mathscr{R}}))\}$
            for $\alpha_{\mathscr{R}} = (m, l_{\mathscr{R}}, m_t) \in m\blacktriangleright$ do  $n_{new} \leftarrow n_{new} \cup \{n_{\mathscr{D}}, m_t, \varepsilon \oplus (rhide, \bot, \alpha_{\mathscr{R}}))\}$ ;
            $\sigma \leftarrow \sigma \cup \{(n_{next}, \rho(n_{next}, c)) \mid n_{next} \in n_{new} \wedge \rho(n_{next}, c) \leq \rho_{max}\}$;
    for $f \in Opt$ do  $InsertIntoPSP(f, c, \mathscr{P})$ ;
return $\mathscr{P}$;
```

The current cost function of a state $x = (n, m, \varepsilon)$ is $g(x) = |\{(op, a_1, a_2) \in \varepsilon \mid op \neq match\} \setminus \{(rhide, \bot, (b_s, l, b_t)) \in \varepsilon \mid l = \tau\}|$, i.e., the number of hide operations in an alignment without the operations over the τs. The heuristics function $h(x,c) = min\{\left|F_{Log}(x,c) \setminus f_{Model}\right| + \left|f_{Model} \setminus F_{Log}(x,c)\right|\}$, such that $f_{Model} \in F_{Model}(x)$, gives an optimistic approximation of the least amount of hide operations required to match the remaining labels in a trace c. In this formula $F_{Log}(x,c)$ represents the future task labels of a trace, such that given $x = (n, m, \varepsilon)$, then $F_{Log}(x,c) = MultiSet(c) \setminus MultiSet(\hat{\varepsilon})$, i.e., the multiset representation of c minus the labels of the trace matched or hidden so far.[3] The future labels in the model $F_{Model}(n)$ are computed with a bottom-up traversal on the strongly connected components of the reachability graph, where the multisets of task labels are collected and stored in each node of the graph. Observe that h assumes that all events with the same label in F_{Log} and f_{Model} are matched, this is clearly an optimistic approximation, since some of the those matches might not be possible; then the optimistic approximation computed by h signifies an admissible heuristics for the A^*-search, which guarantees the optimality of the computed alignments.

Algorithm 2 shows the procedure to build the PSP, where an A^* search is applied to find the optimal alignments for each trace in a log. The algorithm chooses a node with minimal cost ρ, such that if it represents the alignment of a complete trace and the pairing of two final states (one in the DAFSA and one in the reachability graph), then it is marked as an optimal alignment. Otherwise, the search continues by applying *lhide*, *rhide* and *match*. As shown in [15], the complexity for constructing the PSP is in the

[3] *MultiSet* retrieves the multiset representing the labels in a trace or the labels of a set of arcs.

[4] $c(i)$ is the operator in Z notation [14] to obtain the ith element in a sequence.

Algorithm 3: Construct the PSP with Prefix- and Suffix Memoization

```
▷ replace line 2 with the following block:
    ▷ Reuse common prefix alignments
        for $i = 1 \to |c|$ do $\sigma \leftarrow \sigma \cup \{(n_{next}, \rho(n_{next}, c)) \mid n_{next} \in \mathit{PrefixTable}(c \mathit{\ for\ } i)\}$[5];
        if $\sigma = \varnothing$ then $\sigma \leftarrow \sigma \cup \{(s_{\mathscr{P}}, \rho(s_{\mathscr{P}}, c)\}$ ;
▷ replace line 14 with the following block:
    ▷ Reuse common suffix alignments
        $\mathit{suff}_{act} \leftarrow c \mathit{\ after\ } |\{\beta = (op, a_{\mathscr{D}}, a_{\mathscr{R}}) \in \varepsilon \mid op \neq \mathit{rhide}\}|$[6];
        $n_{new} \leftarrow \{(f_{\mathscr{D}}, f_{\mathscr{R}}, \varepsilon \oplus g_{\mathit{suff}}) \mid (f_{\mathscr{D}}, f_{\mathscr{R}}, g_{\mathit{suff}}) \in \mathit{SuffixTable}(n_{\mathscr{D}}, m, \mathit{suff}_{act})\}$;
        $\sigma \leftarrow \sigma \cup \{(n_{next}, \rho(n_{next}, c)) \mid n_{next} \in n_{new}\}$;
        if $n_{new} \neq \varnothing$ then continue ;
```

order of $O(3^{|N_{\mathscr{D}}| \cdot |M|})$ where $N_{\mathscr{D}}$ is the set of states in the DAFSA and M is the set of reachable markings of the Petri net.

In order to cope with the complexity of the computation of the PSP, we propose an optimization based on two memoization tables: prefix and suffix memoization tables. Both tables store a set of partial trace alignments for common prefixes and suffixes that have been aligned previously. The tables are constructed incrementally by identifying common prefixes/suffixes after the alignment of each trace and storing the corresponding partial trace alignments. The integration of these tables requires the modification of Algorithm 2, as shown in Algorithm 3. For each trace c, the algorithm starts by checking if there is a common prefix for c in the prefix memoization table. If this is the case, the A^* starts from the nodes after all partial trace alignments for this common prefix instead of the initial node. In the case of common suffix memoization, the algorithm checks at each iteration whether the current pair of nodes and the current suffix is stored in the suffix memoization table. If this is the case, the algorithm appends nodes to the A^* search for each pair of memoized final nodes and appends all partial suffix alignments to the current alignment instead of continuing the regular search procedure. By reusing the information stored in these tables, the search space for the A^* is reduced.

The approach illustrated so far produces a PSP containing all optimal alignments. Nevertheless, if only one optimal alignment is required, then the algorithm can be easily modified to stop as soon as the first alignment is found. Overall, the complexity of the proposed approach consists of the construction of the DAFSA, the construction of the reachability graph and the computation of the PSP, therefore it is exponential in the worst case, i.e. $O(|L| \cdot \log n + 2^{|P \cup T|} + 3^{|N_{\mathscr{D}}| \cdot |M|})$. The technique presented in this paper does not intend to lower the complexity class for the problem of trace alignment, but rather to implement a more efficient solution within the same complexity class.

Figure 5 shows the PSP obtained by synchronizing the DAFSA of the loan application process in Fig. 3 and the τ-less reachability graph of Fig. 4, we remind the reader that a PSP represents the synchronization of the whole log. To understand its construction let us consider the sample trace $\langle B, D, E \rangle$. Starting from the source node we have $g(n) = 0$, $F_{Log}(n,c) = \{B^1, D^1, E^1\}$, and $F_{Model}(n) = \{B^1, C^1, D^1, E^1\}$. The A^* will

[5] *for* in Z notation [14] to obtain the first i elements of a sequence.

[6] *after* in Z notation [14] to obtain the elements after the first i*th* elements of a sequence.

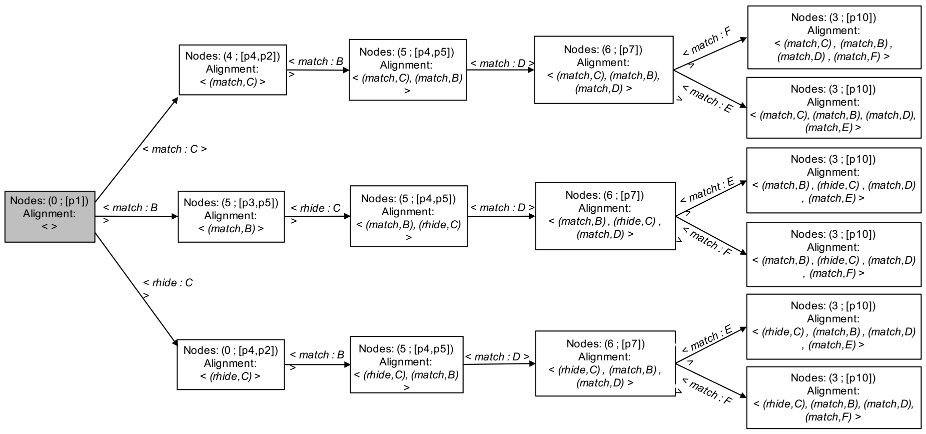

Fig. 5. The PSP for our loan application process example.

compute the cost of performing the following possible synchronizations: $(match, B)$, $(lhide, B)$ $(rhide, B)$, and $(rhide, C)$. Out of these four possibilities it will only explore $(match, B)$[7] and $(rhide, C)$ which have a cost of one. Both $(rhide, B)$[8] and $(lhide, B)$ will never be explored since they have a cost of three and there exist nodes with a lower cost. The A^* will continue exploring the possible synchronizations until all optimal alignments are discovered, which are found in nodes f_3 and f_5 for the trace $\langle B, D, E\rangle$.

3.5 Extracting Behavioral Mismatch Statements

In the previous section, we presented a scalable approach to discover a complete set of optimal alignments between an event log and a Petri net. While it is general practice to assess these alignments one-by-one or to aggregate them into a single metric [2], García-Bañuelos et al. [15] showed that practitioners prefer to reason in terms of natural language statements when investigating behavioral issues.

García-Bañuelos et al. [15] defined nine mismatch patterns over the PSP for the generation of natural language statements, which characterize behavior present in the log and missing in the model and vice versa. Out of these nine mismatch patterns we only support the seven patterns related to *unfitting behavior*. The detection of patterns related to *additional model behavior* is out of the scope of this paper.

Differences related to *unfitting behavior* can be divided into relation mismatch patterns and event mismatch patterns. On the one hand, relation mismatch patterns comprise cases when a pair of events in the log has a different behavior relation (sequence, concurrency, conflict) than the corresponding events in the model. E.g., it is possible to obtain statements such as: *In the log, after "A", "B" occurs before "C", while in the model they are concurrent* or *In the model, after "A", "B" occurs before "C", while*

[7] In case of $(match, B)$ we have a current cost of zero since it is a match (i.e. $g(n) = 0$), and a future cost of one (i.e. $h(n,c) = \left|\{D^1, E^1\} \setminus \{C^1, D^1, E^1\}\right| + \left|\{C^1, D^1, E^1\} \setminus \{D^1, E^1\}\right| = 1$).

[8] In case of $(rhide, B)$ we have a current cost of one since it is a hide (i.e. $g(n) = 1$), and a future cost of two (i.e. $h(n,c) = \left|\{B^1, D^1, E^1\} \setminus \{C^1, D^1, E^1\}\right| + \left|\{C^1, D^1, E^1\} \setminus \{B^1, D^1, E^1\}\right| = 2$).

in the log they are mutually exclusive". On the other hand, event mismatch patterns characterize all other cases of unfitting behavior; e.g., *In the log, after "A", "B" is optional* or *In the model, "A" occurs after "B" instead of "C"*. In the running example, we return the statement *In the model, "C" occurs after "B" and before "D"*. A detailed description of each pattern and their verbalization can be found in [15].

Given that our technique uses the same PSP as in [15], we adapt their algorithm for the generation of *behavioral mismatch statements*. Similar to the original approach, we rely on an oracle for the computation of concurrency relations between events in a log. We use the local concurrency oracle presented in [6], however other oracles can be used, e.g, $\alpha+$ relations [13]. The approach in [6] helps to alleviate the generalisation of the behavior while computing potential concurrent behavior from a log. Roughly speaking, the local concurrency oracle delimits the scope of a concurrency relation between pairs of events to a pair of execution states. Thus, during the generation of statements, the oracle requires a pair of events, as well as an execution state, and outputs *true* if the given events can occur concurrently at that particular state, or *false* otherwise.

4 Evaluation

The presented approach was implemented as a standalone tool.[9] Given a log in XES or MXML format and a model in BPMN or PNML format, the tool returns a list of one or all-optimal alignments, and the list of behavioral mismatch statements. Other structures, such as the DAFSA, reachability graph and PSP, can be also retrieved.

The conducted set of experiments measure the quality and time performance of our approach in comparison with the trace alignment approach. Our approach was compared against the ProM [33] plugin "Replay a Log on Petri Net for Conformance Checking"[10] [3] for one-optimal trace alignment, and against "Replay a Log on Petri Net for All Optimal Alignments" [3] for the case of all-optimal. This latter plugin relies on different baseline algorithms, but only the "Tree-based state space replay for all optimal alignments" algorithm actually aims at generating all-optimal alignments; however it also returns non-optimal results. Therefore, non-optimal results were filtered out in a post-processing step, i.e., those with bigger cost than the optimal computed by the one-optimal trace alignment (this step was not included in the performance measure). The behavioral alignment approach based on event structures was not included in the evaluation, since this approach showed to be generally slower than trace alignment [15].

The performance was measured in terms of execution time (ms) and quality of the results (number of optimal alignments). The alignments are considered optimal when they have the same cost as one optimal trace alignment. Given that the computation of all-optimal trace alignments oftentimes ran for hours before running out of memory, we use two bounds in the experiments: a time bound of 5 minutes (after 10 minutes the alignment will also continue for 12 hours [23]) and a state exploration bound of 100,000 states. Hence, we report on all optimal alignments found for each approach until one of the bounds is reached or until termination. The experiments were conducted on a 6-core Xeon E5-1650 3.50Ghz with 128GB of RAM running JVM 8.

[9] Available from `http://apromore.org/tools`

[10] "A* Cost-based Fitness Express with ILP, assuming at most 32,767 tokens in each place".

4.1 Datasets

The experiments use three model-log pairs. The first pair is a (publicly available) dataset of a real-life Italian road fines management process (hereafter RTFMP) [12], its normative description is presented in [20], whereas its model is presented in [15].

The second dataset is the real-life log "closed problems" of the BPI Challenge 2013 (hereafter BPIC13 cp.) [28] This log originates from an IT incident and problem management system used at Volvo. From this log, the model was discovered using Structured Miner [8]. The log was preprocessed with a noise filter [9]. The resulting model is sound (a requirement for both approaches[11]). In this case the model generated by the Structured Miner has high precision, in contrast to the model discovered by other techniques, such as Inductive Miner [17] that generates an over-generalized model, causing state space explosion when computing all alignments.

The third dataset is the SAP R/3 collection [10], a repository of 604 EPCs documenting the reference model to customize the R/3 ERP product. The models were converted into Petri nets and those with behavioral issues (i.e. unsound models) were filtered out. The event logs were generated from the remaining models using the ProM plugin "Generate Event Log from Petri Net" [30]. This plugin produces unique traces for each possible execution in the model. Next, we filtered out all logs with less than ten unique traces, since such small logs are not useful to measure scalability. This resulted in 120 pairs of real-life models and logs. Additional event logs were created with different levels of noise $(2.5\%, 5\%, 7.5\%, 10\%)$. For that, we duplicated each unique trace in the logs tenfold to maintain the original behavior and used the noise generator tool in [9] to create the noisy logs. This tool inserts events into randomly chosen traces, such that new directly-follows dependencies are created until the noise threshold is reached. The reason for inserting noise is because otherwise there is a perfect fit between the log and the model, and hence the output of the conformance checking is empty, which does not help to test for scalability.

Table 1 provides descriptive statistics of the datasets. The size of the models and of their reachability graphs ($\mathscr{R}$) correspond to the number of places and transitions, and nodes and arcs, respectively. For the SAP R/3 collection, we report the average and standard deviation for the event logs and models, for each noise level. The last column reports on the time required for constructing the reachability graph plus that for removing tau transitions for the given models.

Dataset	**Events**	**Unique events**	**Traces**	**Unique traces**	**Model size**	$\mathscr{R}$ **size**	$\mathscr{R}$ **time** (ms)
RTFMP	561,470	12	150,370	231	35	33	16
BPIC13 cp.	6,660	5	1,487	183	28	9	96
SAP R/3 2.5%	37,580 (±116,515)	15 (±5)	2,795 (±7,897)	1,062 (±3,192)	49 (±16)	128 (±79)	4 (±3)
SAP R/3 5%	38,569 (±119,581)	15 (±5)	2,795 (±7,897)	1,551 (±4,830)	49 (±16)	128 (±79)	4 (±3)
SAP R/3 7.5%	39,612 (±122,813)	15 (±5)	2,795 (±7,897)	2,075 (±6,147)	49 (±16)	128 (±79)	4 (±3)
SAP R/3 10%	40,712 (±126,225)	15 (±5)	2,795 (±7,897)	2,342 (±6,966)	49 (±16)	128 (±79)	4 (±3)

Table 1. Descriptive statistics of the event logs and models.

[11] Strictly speaking, trace alignment requires easy-soundness while our approach requires safeness. However both requirements are satisfied by soundness.

4.2 Results

Table 2 reports the number of optimal alignments and execution times for each conformance checking approach (for the SAP R/3 datasets, we report on the average and the upper bound of the 95% confidence interval for these measurements). To ensure comparability of the results we only count the alignments for our approach (shortened as DAFSA in the table) with the same cost as trace alignment, i.e. the same number of asynchronous moves. However, our approach did not detect any additional non-optimal alignments.

In the case of one-optimal, our approach always returned the same number of alignments as trace alignment. Both approaches are expected to find one-optimal alignment per unique trace of an event log, thus the number of alignments and the number of unique traces is the same. However, there is no intuitive expectation for all-optimal alignments. In this regard, our approach found many more optimal alignments than trace alignment within the same state space and time bounds. For example, on the RTFMP log, our approach found 467 alignments instead of 338 returned by trace alignment. This difference increases substantially in the other datasets: in logs with high noise levels (SAP R/3 7.5 and 10%), our approach returned up to five times the number of all-optimal alignments than trace alignment. This is due to the reuse of partial trace alignments (prefix and suffix memoization). Our approach scaled well to the number of unique traces and to the amount of unfitting behavior observed in the logs. Additionally, (all-optimal) trace alignment suffers from reporting non-optimal results that have to be filtered in a preprocessing step[12] (unfiltered results are reported in square brackets). Thus, our approach was capable of finding a more complete set of alignments.

Comparing the execution times for the all-optimal variants, our approach outperforms the tree-based trace alignment approach by 1-2 orders of magnitude. For example, our approach took 125 milliseconds to compute all alignments for the RTFMP dataset, as opposed to 52 seconds for trace alignment. Additionally, trace alignment times out in 207 out of 480 cases for the SAP dataset, while our approach only timed out in two cases (trace alignment also timed out in these two cases). Our one-optimal variant performs 1.5 to nearly 40 times faster (trace alignment timed out in 2 cases for the SAP datasets, while our approach never timed out). Only in the BPIC13 cp. dataset the trace alignment outperformed our approach by nearly a factor two. The process model in this dataset contains a large state space due to the presence of nested loops, which can lead to a combinatorial state space to be explored by the A^* algorithm. Thus, when the estimation of our heuristics is imprecise due to complex loop structures, the memory and time requirements increase. Conversely, trace alignment uses a more accurate heuristic function, which leads to outperforms our approach in the BPIC13 cp. dataset. In short, the execution times positively correlate with the number of unique traces in a log, both approaches, DAFSA and trace alignment, apply an A^* algorithm for each unique trace. Our approach scales better in the case of more complex SAP R/3 logs, which exhibit a very high number of unique traces compared to the RTFMP and BPIC13 cp., as per Table 1. Our approach calculates all optimal alignments in less than ten seconds, while

[12] An alignment was filtered if it had a higher cost than that computed by one-optimal alignment or if it represented the swap of the label of an invisible task with that of a visible one.

	Optimal alignments (upper bound of 95% confidence interval)				**Execution time (ms)** (upper bound of 95% confidence interval)			
	All optimal		**One optimal**		**All optimal**		**One optimal**	
Dataset	**DAFSA**	**Trace align. [#unfiltered]**	**DAFSA**	**Trace align.**	**DAFSA**	**Trace align.**	**DAFSA**	**Trace align.**
RTFMP	467	338 [1,898,182]	231	231	**125**	52,041	**56**	1,844
BPIC13 cp.	28,656	22,259 [1,904,057]	183	183	**5,360**	50,160	453	**260**
SAP R/3 2.5%	4,253 (22,675)	1,233 [1,067,533] (6,470 [1,929,629])	1,062 (7,319)	1,062 (7,319)	**1,102** (7,778)	127,013 (300,000)	**814** (6,132)	1,800 (12,891)
SAP R/3 5%	7,672 (41,133)	1,751 [1,224,079] (9,178 [2,199,248])	1,551 (11,019)	1,551 (11,019)	**2,832** (28,040)	150,017 (300,000)	**1,718** (18,696)	3,415 (25,857)
SAP R/3 7.5%	11,652 (61,504)	2,154 [1,283,583] (14,207 [3,039,240])	2,075 (14,122)	2,075 (14,122)	**3,208** (19,502)	163,593 (300,000)	**2,083** (12,912)	4,967 (58,961)
SAP R/3 10%	15,754 (84,167)	2,809 [1,286,568] (22,883 [3,302,068])	2,342 (15,996)	2,342 (15,996)	**8,204** (75,643)	173,438 (300,000)	**3,480** (25,371)	7,365 (66,003)

Table 2. Evaluation results.

trace alignment reaches the time bound of five minutes on average in every second model-log pair for the same dataset.

In our approach, a trade-off between the execution time and number of alignments is observed from the comparison of the results of one-optimal versus all-optimal. It is more obvious when the amount of unfitting behavior increases. E.g., in the logs SAP R/3 with 10% noise level, our approach took, on average, five seconds longer for computing all-optimal alignments than one-optimal, but returns ten times more alignments.

The extraction of behavioral mismatch statements shows that a large number of alignments can be represented by a significantly smaller, yet interpretable, number of statements. E.g., 3,295 all-optimal alignments in the BPIC13 cp. dataset can be summarized with only 14 statements, and reduced to eight statements in the case of one-optimal alignment. In the RTFMP dataset, 120 statements were computed from 467 all-optimal alignments, whereas only 69 statements were computed for the one-optimal variant. Some example statements are:

– *In the log, after "Insert Fine Notification", "Payment" occurs before task "Add penalty", while in the model they are mutually exclusive.*
– *In the log, after "Add penalty", "Payment" is substituted by "Send Appeal to Prefecture".*

5 Conclusion

We showed that the problem of conformance checking can be mapped to that of computing a minimal error-correcting product between an automaton representing the event log (its minimal DAFSA) and an automaton representing the process model (its reachability graph). The resulting product automaton can be used to produce sets of optimal alignments between each trace in the log and a corresponding trace in the model, or even statements capturing behavioral relations (e.g. conflict relations) observed in a state of the DAFSA but not captured in the corresponding state in the model.

The use of a DAFSA to represent the event log allows us to benefit from both prefix and suffix compression of the traces in the log. This is a distinctive feature of the proposal with respect to trace alignment, which computes an alignment between each trace in the log and the model, without any reuse across traces. Due to this distinctive feature, our approach addresses some of the scalability issues of existing conformance checking techniques allowing more interactivity in redesigning process models with conformance issues. The approach can be employed to assess the quality of automated process model discovery techniques, as well as used as the cornerstone technique for process model repair.

The empirical results show that the execution times of our approach are one to two orders of magnitude faster than those of the all-optimal trace alignment method. When restricted to the problem of computing one alignment per trace (one-optimal alignment), our approach generally but not always outperforms the baseline [2]. This is attributable to the fact that the latter uses a tight heuristic function, whereas the heuristic function we use over-approximates in some cases. Designing a better heuristic function is a direction for future work. The ideas in [2] are not directly transposable as the heuristic function in our approach needs to compute a bound starting from a state of the whole log (the DAFSA), while [2] does so for one trace at a time.

In this paper, we focused on the problem of identifying unfitting log behavior. A possible avenue for future work is to extend the approach to detect additional model behavior, by adapting the ideas proposed in [15] in the context of event structures.

Finally, the empirical evaluation, while based on synthetic and real-life models and logs, is limited in that it only covers models with sizes of up to 50 tasks and logs with up to ca. 2.5K distinct traces. Conducting a more thorough evaluation with even larger process models and event logs is another avenue for future work.

Acknowledgments. This research is partly funded by the Australian Research Council (grant DP150103356) and the Estonian Research Council (grant IUT20-55).

References

1. A. Adriansyah, J. Muñoz-Gama, J. Carmona, B.F. van Dongen, and W.M.P. van der Aalst. Measuring precision of modeled behavior. *ISeB*, 13(1):37–67, 2015.
2. A. Adriansyah, B.F. van Dongen, and W.M.P. van der Aalst. Conformance checking using cost-based fitness analysis. In *Proc. of EDOC*, pages 55–64. IEEE, 2011.
3. A. Adriansyah, B.F. van Dongen, and W.M.P. van der Aalst. Memory-efficient alignment of observed and modeled behavior. *BPM Center Report*, 2013.
4. A.K. Alves de Medeiros. *Genetic Process Mining*. PhD thesis, TU/e, 2006.
5. A. Armas-Cervantes, P. Baldan, M. Dumas, and L. García-Bañuelos. Diagnosing behavioral differences between business process models: An approach based on event structures. *Information Systems*, 56, 2016.
6. A. Armas-Cervantes, M. Dumas, and M. La Rosa. Discovering local concurrency relations in business process event logs. eprint # 102438, QUT, 2016.
7. A. Armas-Cervantes, M. La Rosa, M. Dumas Menjivar, L. García-Bañuelos, and N.R. van Beest. Interactive and incremental business process model repair. eprint # 106611, QUT, 2017.
8. A. Augusto, R. Conforti, M. Dumas, M. La Rosa, and G. Bruno. Automated discovery of structured process models: Discover structured vs. discover and structure. In *Proc. of ER*, LNCS 9974. Springer, 2016.
9. R. Conforti, M. La Rosa, and A.H.M. ter Hofstede. Filtering out infrequent behavior from business process event logs. *IEEE TKDE*, 29(2):300–314, 2016.
10. T. Curran and G. Keller. *SAP R/3 Business Blueprint: Understanding the Business Process Reference Model*. Upper Saddle River, 1997.

11. J. Daciuk, S. Mihov, B.W. Watson, and R.E. Watson. Incremental construction of minimal acyclic finite-state automata. *Computational linguistics*, 26(1):3–16, 2000.
12. M. de Leoni and F. Mannhardt. Road traffic fine management process, 2015.
13. Ana Karla A de Medeiros, Wil MP van der Aalst, and AJMM Weijters. Workflow mining: Current status and future directions. In *Proc. of OTM*, pages 389–406. Springer, 2003.
14. A. Diller. *Z: An introduction to formal methods*. John Wiley & Sons, Inc., 1990.
15. L. García-Bañuelos, N. van Beest, M. Dumas, M. La Rosa, and W. Mertens. Complete and interpretable conformance checking of business processes. *IEEE TSE*, 43, 2017. In press.
16. P.E. Hart, N.J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE TSSC*, 4(2):100–107, 1968.
17. S. Leemans, D. Fahland, and W. van der Aalst. Discovering block-structured process models from event logs - a constructive approach. In *Proc. of Petri Nets*, LNCS. Springer, 2013.
18. Sander JJ Leemans, Dirk Fahland, and Wil MP van der Aalst. Scalable process discovery and conformance checking. *Software & Systems Modeling*, pages 1–33, 2016.
19. Richard Lipton. The reachability problem requires exponential space. *Research Report 62, Department of Computer Science, Yale University, New Haven, Connecticut*, 1976.
20. F. Mannhardt, M. de Leoni, H.A. Reijers, and W.M.P. van der Aalst. Balanced multi-perspective checking of process conformance. *Computing*, 98:407–437, 2016.
21. E.W. Mayr. An algorithm for the general petri net reachability problem. *SIAM Journal on computing*, 13(3):441–460, 1984.
22. J. Muñoz-Gama and J. Carmona. A fresh look at precision in process conformance. In *Proc. of BPM*, pages 211–226. Springer, 2010.
23. J. Muñoz-Gama, J. Carmona, and W.M.P. van der Aalst. Single-entry single-exit decomposed conformance checking. *Inf. Syst.*, 46:102–122, December 2014.
24. Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.
25. A. Polyvyanyy, W.M.P. Van Der Aalst, A.H.M. Ter Hofstede, and M.T. Wynn. Impact-driven process model repair. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 25(4):28, 2016.
26. A. Rozinat and W.M.P. van der Aalst. Conformance checking of processes based on monitoring real behavior. *Inf. Syst.*, 33(1):64–95, 2008.
27. W. Song, X. Xia, H.A. Jacobsen, P. Zhang, and H. Hu. Efficient alignment between event logs and process models. *IEEE Transactions on Services Computing*, 10(1):136–149, 2017.
28. W. Steeman. Bpi challenge 2013, closed problems, 2013.
29. B. van Dongen, J. Carmona, T. Chatain, and F. Taymouri. Aligning modeled and observed behavior: a compromise between complexity and quality. In *Proc. of CAiSE*. Springer, 2017.
30. S. vanden Broucke, J. De Weerdt, J. Vanthienen, and B. Baesens. An improved process event log artificial negative event generator. Technical Report KBI_1216, KU Leuven, 2012.
31. S.K.L.M. vanden Broucke, J. De Weerdt, J. Vanthienen, and B. Baesens. Determining process model precision and generalization with weighted artificial negative events. *IEEE TKDE*, 26(8):1877–1889, 2014.
32. S.K.L.M. vanden Broucke, J. Munoz-Gama, J. Carmona, B. Baesens, and J. Vanthienen. Event-based real-time decomposed conformance analysis. In *Proc. of OTM*, pages 345–363. Springer, 2014.
33. HMW Verbeek, JCAM Buijs, BF Van Dongen, and Wil MP van der Aalst. Prom 6: The process mining toolkit. *Proc. of BPM Demonstration Track*, 615:34–39, 2010.