Efficient Approximate Conformance Checking Using Trie Data Structures

Ahmed Awad
University of Tartu, Tartu, Estonia
ahmed.awad@ut.ee
Cairo University, Giza, Egypt

Kristo Raun
University of Tartu, Tartu, Estonia
kristo.raun@ut.ee

Matthias Weidlich
Humboldt-Universität zu Berlin, Berlin, Germany
matthias.weidlich@hu-berlin.de

***Abstract*—Conformance checking compares a process model and recorded executions of a process, i.e., a log of traces. To this end, state-of-the-art approaches compute an alignment between a trace and an execution sequence of the model. Since the construction of alignments is computationally expensive, approximation schemes have been developed to strike a balance between the efficiency and the accuracy of conformance checking. Specifically, conformance checking may rely only on so-called proxy behavior, a subset of the behavior of the model. However, the question how such proxy behavior shall be represented for efficient alignment computation has been largely neglected.**

In this paper, we contribute a new formulation of the proxy behavior derived from a model for approximate conformance checking. By encoding the proxy behavior using a trie data structure, we obtain a logarithmically reduced search space for alignment computation compared to a set-based representation. We show how our algorithm supports the definition of a budget for alignment computation and also augment it with strategies for meta-heuristic optimization and pruning of the search space. Evaluation experiments with five real-world event logs show that our approach reduces the runtime of alignment construction by two orders of magnitude with a modest estimation error.

I. Introduction

Conformance checking compares the behavior as defined by a model for a specific process with traces of an event log that represents the behavior as recorded during process execution [8]. Conformance checking has diverse applications. It enables compliance and risk assessment [9], supports model-driven performance analysis [28], and serves as a means to evaluate the quality of automatically discovered process models [5].

Common techniques for conformance checking compute an alignment between a trace and an execution sequence of a process model [1]. An alignment relates events of a trace to activity executions defined by the model, such that a cost function over this relation is minimized. State-of-the-art algorithms phrase the construction of such optimal alignments as a search problem and show an exponential time complexity in the size of the trace and the model [8]. This is problematic, especially when alignments need to be constructed repeatedly. When traces evolve continuously due to new events being recorded in online scenarios, or models are subject to change (e.g., in exploratory analysis by a user or in the context of iterative discovery algorithms), the sheer number of alignments to compute may render an analysis intractable.

Against this background, it was suggested to rely on approximate conformance checking, which potentially trades the optimality of the constructed alignments against computational efficiency [2], [22]. Specifically, approximate conformance checking may be realized by sampling traces of the event log [2], [3], by adopting a heuristic algorithm for alignment construction [13], [25], or by considering some *proxy* behavior, i.e., a subset of the behavior of the process model [22], [21]. However, approaches that fall into the latter category and incorporate proxy behavior focus on the computation of bounds of conformance measures. For the computation of actual alignments, they rely on a set-based representation of the proxy behavior. Given that compact representations for sequential data have been researched for decades to enable efficient search [16], there is a notable research gap: *Structures for compact representations of behavior have not yet been utilized for efficient, approximate computation of alignments.*

In this paper, we explore this gap and present an approach for approximate conformance checking based on tries, i.e., tree-based data structures for efficient search in sequential data. In our context, a trie represents the proxy behavior derived from a process model and directly serves as an encoding of the search space for the computation of an alignment. As such, compared to a set-based representation of the proxy behavior used in existing work [22], [21], we achieve a logarithmic reduction of the search space in the best and average cases. More specifically, our contributions are summarized as follows:

(1) We propose a trie data structure for conformance checking. It extends traditional prefix trees with annotations to support the construction of alignments.
(2) We present an algorithm to construct alignments using the trie data structure.
(3) We extend the basic algorithm with a user-defined budget for alignment computation, as well as strategies for meta-heuristic optimization and pruning of the search space.

We evaluate our approach with five real-world event logs. Our results demonstrate that our approach, in most of the cases, finds alignments two orders of magnitude faster than a baseline technique with a mean absolute error close to one.

Next, Section II provides background for our work, before Section III introduces our approach to approximate conformance checking. We present experimental results in Section IV, review related work in Section V, and conclude in Section VI.

II. Background

We first discuss event logs and process models, before summarizing the essence of (approximate) conformance checking.

A. Event Logs

Data about the execution of a process is commonly represented as an *event log* [27]. It contains *traces* that denote single executions of the process, each trace being a sequence of events. An *event*, in turn, represents the execution of an activity of the process. Traces that represent distinct process executions, but are built of events that induce the same sequence of activity executions are said to be of the same *trace variant*.

While events and traces may be assigned further information about the context of process execution, such as timestamps or data payload, a relatively simple model for event logs is sufficient for the context of our work. Specifically, with $\mathcal{A}$ as a universe of activities, we model a trace σ as a finite sequence of activities, $\sigma = \langle a_1, \ldots, a_n \rangle \in \mathcal{A}^*$. We use the notation $\sigma(i)$ for the activity at the i-th position of σ, while σ_i ($_i\sigma$) refer to the suffix (prefix) of σ from (up to) and including position i, $i \geq 1$. An event log is often modeled as a multiset of traces, thereby incorporating the frequency with which a certain trace variant has been recorded. However, as essential conformance results are obtained on the level of a trace variant, information on its frequency is irrelevant in our context. To keep the notation concise, we therefore do not distinguish traces and trace variants, and capture an event log L as a set of traces, $L \subseteq \mathcal{A}^*$. Below, we will use the event log $L = \{\langle a, b, c, e \rangle, \langle a, e \rangle, \langle c, e \rangle\}$ for illustration purposes.

B. Process Models

A process model defines which sequences of activity executions are considered to be valid. As such, its behavior M may also be represented by a set of sequences of activities, $M \subseteq \mathcal{A}^*$. To avoid confusion with the traces of an event log, we refer to the elements of M as execution sequences. However, the above notations for activities at a position ($\pi(i)$), a suffix (π_i), and prefix ($_i\pi$) also apply for an execution sequence $\pi \in M$.

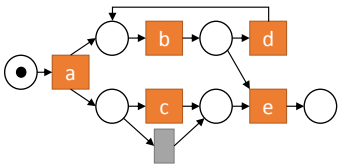


Fig. 1. Example process model for conformance checking

Figure 1 shows a process model, captured as a Petri net. Abstracting from the formal details, execution sequences defined by this model start with activity a, followed by either b, c, or e (the so-called silent transition marked in gray is not observable). Also, activities b and c may be observed in either order and the loop in the model allows for repeated executions of activities b and d. Example execution sequences are $\langle a, b, c, e \rangle$ and $\langle a, c, b, d, b, d, b, e \rangle$. Note that, due to the loop, the set of execution sequences of this model is unbounded.

C. Conformance Checking

Conformance checking compares the behavior recorded in an event log with the behavior specified by a process model [8]. To this end, an alignment between a trace $\sigma \in L$ of the log and an execution sequence $\pi \in M$ the model may be computed [1]. An alignment $\gamma = \langle (x_1, y_1), \ldots, (x_n, y_n) \rangle$ is a sequence of steps, each step $(x, y) \in (\mathcal{A} \cup \{\gg\}) \times (\mathcal{A} \cup \{\gg\})$ linking an activity of the trace, or the skip symbol $\gg$, to an activity of the execution sequence, or the skip symbol, whereas a step $(\gg, \gg)$ is illegal. Here, it must hold that the projection of γ on the first component, ignoring $\gg$, yields σ, and the projection of γ on the second component, ignoring $\gg$, yields π. A step (x, y) is called *synchronous move* if $x, y \in \mathcal{A}$, *log move* if $y = \gg$, a *model move* if $x = \gg$, while the last two are jointly referred to as asynchronous moves.

Assigning costs to steps, a cost-optimal alignment of a trace and an execution sequence can be identified. Often, a cost of one is assigned to asynchronous moves, while synchronous moves for the same activity have zero cost. Then, an optimal alignment minimizes the edit distance between a trace and an execution sequence. Identifying an optimal alignment for a trace and all execution sequences of a model is a search problem for which existing algorithms show an exponential time complexity in the size of the trace and the model [8]. In the remainder, we write $\delta(\gamma)$ for the total cost of an alignment γ.

D. Conformance Checking using Proxy Behavior

To increase the efficiency of conformance checking, we may rely only on a sample of the behavior of a process model [22]. Then, an alignment for a trace is computed no longer based on all execution sequences M of a process model, but using a subset $M' \subset M$ of them, referred to as *proxy behavior*. Ideally, optimal alignments of traces with the proxy behavior M' have equal or only slightly higher costs than the optimal alignments with M. Various strategies for the selection of proxy behavior that aim at this goal are available [21], [22].

In the remainder, we assume the proxy behavior of the process model in Figure 1 to be given as $M' = \{\langle a, b, e \rangle, \langle a, b, c, d, b, d, b, e \rangle, \langle a, b, c, d, b, e \rangle, \langle a, b, c, e \rangle, \langle a, c, b, e \rangle\}$.

III. Efficient Approximate Conformance Checking

In this section, we first give an overview of the approach (Section III-A). Next, we define a trie data structure for conformance checking (Section III-B) and show how it is used for alignment computation (Section III-C). Finally, we present algorithmic optimizations (Section III-D).

A. Approach Overview

An overview of our approach to approximate conformance checking is given in Figure 2. We build upon the idea to increase the efficiency of conformance checking by considering solely a subset of the behavior defined by a process model. That is, we assume that one of the existing approaches, such as [22], to extract executions sequences from the process model is used to construct some proxy behavior. These steps are marked in gray in Figure 2 and are out of scope of this paper.

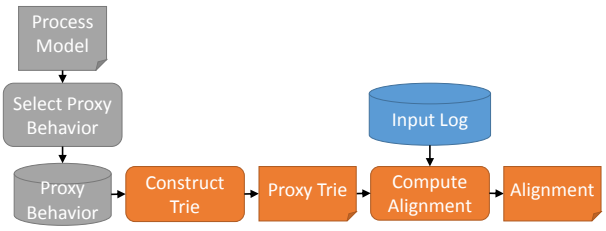


Fig. 2. Overview of our approach

Using the proxy behavior, as illustrated by the steps marked in orange in Figure 2, we first construct a trie data structure for the proxy behavior, referred to as *proxy trie*. It captures the selected proxy behavior in a compact manner and is used directly for the computation of alignments for the traces in a given event log. In essence, the proxy trie denotes an encoding of the search space for the identification of the best-matching execution sequence of the model for a specific trace.

B. Trie Construction

Trie definition. A trie data structure is essentially a prefix tree, which is commonly used in string matching [16]. This use case inspires the use of tries to compute alignments, as the two problems are similar to some extent. However, to better serve our purpose, we do not employ a traditional notion of a trie, but annotate its nodes and edges with some information that is beneficial when using the trie for alignment computation.

Definition 1 (Proxy Trie): Let $M' \subseteq \mathcal{A}^*$ be some proxy behavior of a process model. Then, the *proxy trie* constructed for it is a structure $(N, E, root, l, isEnd, min, max)$ where:

- N is a finite set of nodes. There is one node per prefix ${}_i\pi$ for any execution sequence $\pi \in M'$ as well as one additional node $root \in N$.
- $E \subset N \times N$ is a set of edges, s.t. for all $n \in N$ it holds $|\{n' \mid (n', n) \in E\}| \leq 1$ and (N, E) is a connected graph. There are edges from $root$ to all nodes representing prefixes of length one, and from each node n to node n', if the prefix represented by n' is obtained from the prefix of n by concatenation with a single activity.
- $root \in N$ is the root of the trie, i.e., the only node $n \in N$ for which $|\{n' \mid (n', n) \in E\}| = 0$;
- $l : N \to (\mathcal{A} \cup \{\bot\})$ is a labeling function for nodes. The label is the last activity of the prefix represented by the node, while $root$ is assigned $\bot$.
- $isEnd : N \to \{0, 1\}$ is a Boolean function that indicates end nodes, i.e., whether the prefix represented by the node denotes an execution sequence.
- $min, max : E \to \mathcal{N}$ are functions assigning to an edge the minimum and maximum path length, respectively, to reach an end node, when traversing that edge.

Example. Figure 3 illustrates the notion of a proxy trie using the execution sequences presented earlier. Note that the numbers at each edge encode the minimum length and the maximum length to an end node, respectively. For example, the edge from the root node to its only child node is annotated with $(3, 8)$, meaning that, if we follow that edge, the shortest path

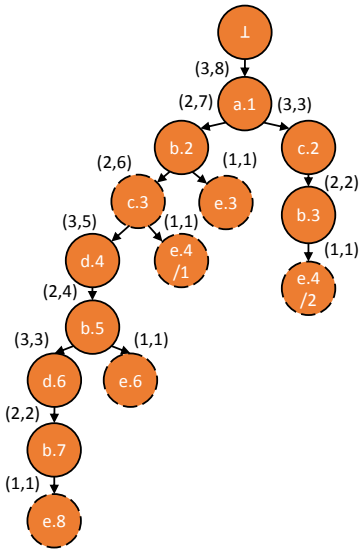


Fig. 3. The trie for the example proxy behavior

to an end of an execution sequence is 3, which corresponds to $\langle a, b, e \rangle$, whereas the longest path of length 8 corresponds to $\langle a, b, c, d, b, d, b, e \rangle$. Moreover, the level of non-root nodes encodes the length of the represented prefix and, hence, the position of the activity defined as a label in the respective execution sequence. Several nodes may share the same label, representing the occurrence of an activity in different prefixes. Nodes with a dashed border, not necessarily leaf nodes of the tree, are end nodes.

Space and Time Complexity. Encoding the proxy behavior using a trie reduces the space needed to store it. All common prefixes among execution sequences are stored only once. In the average case, the trie size is $\mathcal{O}(log_k N)$, where k is the branching factor (fan out) and N is the total number of activities that occur in any execution sequence (i.e., the sum of the length of the execution sequences). The worst case occurs when the execution sequences are distinct. Then, the size of the trie is linear, $\mathcal{O}(N)$. The time needed to construct the trie is also linear, $\mathcal{O}(N)$, as all execution sequences are scanned once.

C. Computing Alignments

Search problem. Computing an alignment of a trace against the proxy behavior, encoded as a trie, can be seen as a variant of trie traversal. The best case occurs when there is a common (pre) infix between the trace and the proxy trie, which represents synchronous moves in a respective alignment. In the case of deviations, we have to explore asynchronous moves, either in the trace or in the proxy trie. This yields a search problem. When a synchronous move is not possible, we have to explore the possibilities represented by a log move or a model move. To this end, we assign a cost to each possible asynchronous move and keep these moves in a priority queue. Then, we follow the move with the least cost, potentially resuming the search from one of the other moves at a later point in time. As such, asynchronous moves induce the states of the search space to find an optimal alignment.

Algorithm 1 Alignment construction by trie traversal

Input: $(N, E, root, l, isEnd, min, max)$, a trie; σ, a trace.
Output: γ, an alignment.

1: $state.n \leftarrow root$
2: $state.\hat{\sigma} \leftarrow \sigma$
3: $state.\hat{\gamma} \leftarrow \langle\rangle$
4: $state.c \leftarrow 0$
5: $candState \leftarrow \perp$ ▷ The best alignment found so far
6: $S \leftarrow \{state\}$
7: **while** $|S| > 0$ **do**
8: $state \leftarrow pickState(S)$
9: **if** $|state.\hat{\sigma}| = 0 \wedge isEnd(state.n)$ **then** ▷ Trie and trace exhausted
10: **if** $candState = \perp \vee \delta(state.\gamma) < \delta(candState.\gamma)$ **then**
11: $candState \leftarrow state$
12: **else if** $|state.\hat{\sigma}| = 0$ **then** ▷ Only trace exhausted
13: ▷ Add model moves to nearest end of trie path
14: $n' \leftarrow state.n.getChildOnShortestPathToEnd()$
15: **while** $n' \neq \perp$ **do**
16: $state.\gamma \leftarrow state.\gamma \cdot \langle(\gg, l(n'))\rangle$
17: $n' \leftarrow n.getChildOnShortestPathToEnd()$
18: $updateCandidateState(state, candState)$
19: **else if** $isEnd(state.n)$ **then** ▷ Only trie exhausted
20: ▷ Add log moves for the rest of the suffix
21: **for** $i = 1; i \leq |state.\hat{\sigma}|; i++$ **do**
22: $state.\gamma \cdot \langle(l(state.\hat{\sigma}(i)), \gg)\rangle$
23: $updateCandidateState(state, candState)$
24: **else**
25: $a \leftarrow state.\hat{\sigma}(1)$
26: $n' \leftarrow state.n.getChildWithActivityLabel(a)$
27: **if** $n' \neq \perp$ **then** ▷ Synchronous move
28: $newState.n \leftarrow n'$
29: $newState.\hat{\sigma} \leftarrow state.\hat{\sigma}_1$
30: $newState.\hat{\gamma} \leftarrow state.\hat{\gamma} \cdot \langle(a, a)\rangle$
31: $newState.c = cost(n', state.\hat{\sigma}_1)$
32: $S \leftarrow S \cup \{newState\}$
33: **else** ▷ Consider both, log and model moves
34: $S \leftarrow S \cup \{createLogMoveState(state)\}$
35: $S \leftarrow S \cup createModelMoveStates(state)$
36: **if** $candState \neq \perp$ **then return** $candState.\gamma$
37: **else return** $\langle\rangle$

The above formulation of the search problem is close to the one adopted by existing techniques for the construction of an optimal alignment, see [29]. However, there are a few notable differences: 1) We do not generate a search space for each trace, or rather trace variant, as it is done when constructing the so-called synchronous product model using traditional search techniques [8]. Rather, the proxy trie is used to structure the search. 2) In case of asynchronous moves, states of the search space are generated and explored from the point of a deviation between the trace and the proxy behavior. 3) Annotations of the trie are used to guide the search and prioritize the states to explore.

In the remainder of this section, we will elaborate on the details of our search strategy for the construction of alignments.

Guiding objective function. The formulation of a good cost function is a key ingredient for efficient search [24]. Our design of the cost function was guided by the following principles:

- Favor depth-first traversal: To find an alignment, we aim to traverse the trie, so that the trace suffix is exhausted. Later, we may consider other paths to find a better alignment.
- Exploit local information: At each node, we may benefit from local information on the minimum/maximum path length to an end node and on the children of the current node in the trie. This information helps to estimate the costs induced by moves. For example, if a synchronous move is not possible, we would prefer a model move in case a child of the current node has the same activity as a label as the currently investigated event in the trace.

Against this background, our guiding cost function computed for a node $n \in N$ of a trie $(N, E, root, l, isEnd, min, max)$ and a trace suffix σ_i is defined as:

$$cost(n, \sigma_i) = \begin{cases} 0 & \text{sync. move} \\ |\sigma_i| + \min\limits_{e=(n,\_)\in E} min(e) & \text{log move} \\ |\sigma_i| + \min\limits_{e=(n,\_)\in E} min(e) & \text{model move} \\ -|\{(n, n') \in E \mid l(n') = \sigma_i(1)\}| & \end{cases} \quad (1)$$

The cost function assigns zero costs to nodes that represent synchronous moves. For log moves, the cost is derived from the length of the trace suffix still to be aligned and the length of the shortest path to an end node in the trie. For model moves, the cost is the same as for log moves, but reduced in case the head of the trace suffix matches the activity assigned as a label to a child of the current node.

Algorithm. Our approach to compute an alignment based on trie traversal is summarized in Alg. 1. It takes a trie and a trace as input and returns an alignment. The alignment construction is a search that employs the following notion of a state:

Definition 2 (State): A state is a tuple $(n, \hat{\gamma}, \hat{\sigma}, c)$, where

- n is a node of the trie;
- $\hat{\gamma}$ is a (partial) alignment;
- $\hat{\sigma}$ is a trace suffix;
- c is the cost of the move represented by the node.

We explain the algorithm by means of examples, as follows.

Example. Let us consider the log $L = \{\langle a, b, c, e\rangle, \langle a, e\rangle, \langle c, e\rangle\}$ introduced earlier. Using the trie from Figure 3, we note that trace $\langle a, b, c, e\rangle$ matches a path in the trie, i.e., the cost is 0.

For trace $\langle a, e\rangle$, event a matches the label of node $a.1$, so a synchronous move is made (line 27 in Alg. 1). The corresponding state is added to the priority queue of states to explore (modelled as a set in Alg. 1 for simplicity), with cost 0 (line 32). Next, we try to match the first event in the suffix $\langle e\rangle$, to a child of node $a.1$. As there is no match, we consider log and model moves. Using Eq. 1, we determine the log move cost as 2. For node $a.1$, there are two possible model moves, i.e., via b to node $b.2$ or via c to node $c.2$. Here, the cost for taking $b.2$ is 1, since there is a child labelled with activity e. The cost for taking $c.2$ is 3. As such, three states are enqueued: $s1 = (a.1, \langle(a, a), (\gg, e)\rangle, \langle\rangle, 2)$, $s2 = (b.2, \langle(a, a), (b, \gg)\rangle, \langle e\rangle, 1)$, and $s3 = (c.2, \langle(a, a), (c, \gg)\rangle, \langle e\rangle, 3)$. Choosing $s2$, a synchronous move (e, e) is found and we reach an end node, which corresponds to the alignment $\langle(a, a), (b, \gg), (e, e)\rangle$ with cost 1. State $s2$ is now a candidate state. Next, $s1$ will be dequeued. Here, we exhausted the trace and need to follow the shortest path in the trie, line 14. So, the alignment will be $\langle(a, a), (\gg, e), (b, \gg), (e, \gg)\rangle$ with cost 2. Yet, as this cost is greater than the cost of the alignment of the candidate state, it is rejected. State $s3$ is handled similarly.

For trace $\langle c, e\rangle$, Alg. 1 examines log and model moves, i.e., $s4 = (\bot, \langle(\gg, c)\rangle, \langle e\rangle, 4)$ and $s5 = (a.1, \langle(a, \gg)\rangle, \langle c, e\rangle, 4)$. Both have the same cost. Assuming that $s5$ is dequeued first, we reach a new state $s6 = (a.1, \langle(a, \gg), (c, c)\rangle, \langle e\rangle, 0)$, which is dequeued next, as it is a synchronous move. From $s6$, no synchronous move can be made, so that model and log moves are checked. Later, the candidate state $(e.4/2, \langle(a, \gg), (c, c), (b, \gg), (e, e)\rangle, \langle\rangle, 0)$ is reached. The remaining states to explore do not result in better alignments.

D. Algorithmic Optimizations

Search budget. Asynchronous moves cause the search space to grow exponentially. At each step, at least two states need to be explored, one log move and as many model moves as there are children for the current node in the trie. To address this problem, we consider a variant of Alg. 1 that incorporates a budget, as it is widely employed in machine learning techniques [11]. The budget provides an upper bound on the number of iterations of the algorithm (line 7-35), before the alignment of the best candidate state found so far is reported.

Meta-heuristic optimization. As the objective function is designed to favor depth-first traversal, the search may get trapped in a local minimum for many iterations. Thus, a second optimization we implement is to alternate between exploitation and exploration [24]. That is, we adapt the function $pickState$ in Alg. 1 (line 8) to periodically explore other parts of the search space, i.e., other states are taken from the priority queue. This alternation is controlled by a parameter $frequency$. Upon exploration, we pick a random unexplored state and evaluate its cost (Eq. 1) and add the respective new states to the priority queue. We later show the positive effect of this optimization.

Pruning. To further control the growth of the search space, we prune infeasible states early. The pruning is applicable when the algorithm reaches a candidate state, in which both, a path of the trie and the trace have been exhausted. A new state is infeasible if in the best case, the cost of the respective alignment is larger than the cost of the alignment of the candidate state. If so, we assume that asynchronous moves will occur only due to a difference in the lengths of the remaining path in the trie and the trace suffix. We capture this property as follows:

Definition 3 (Infeasible State): Let $(N, E, root, l, isEnd, min, max)$ be a trie and $(n_{cs}, \hat{\sigma_{cs}}, \hat{\gamma_{cs}}, c_{cs})$ be a candidate state. A state $(n_s, \hat{\sigma_s}, \hat{\gamma_s}, c_s)$ is infeasible, if

$$\delta(\hat{\gamma_s}) + \min\left(\left||\sigma_s| - \min_{\substack{e=(n_s,\_)\\ \in E}}(min(e))\right|, \left||\sigma_s| - \max_{\substack{e=(n_s,\_)\\ \in E}}(max(e))\right|\right) > \delta(\hat{\gamma_{cs}}).$$

At the respective node, we check the minimum difference between the trace suffix from one side, and either the minimum path or the maximum path to the end of a path in the trie on the other side. This difference has to be in the form of asynchronous moves. If adding this alignment cost to the one at state s exceeds the cost of the best known alignment, the state is not explored further.

IV. Evaluation

Below, we first summarize our evaluation setup, before presenting experimental results obtained with it.

A. Evaluation Setup

Procedure. We compared our approach to the state of the art for approximate conformance checking with proxy behavior [22]. It relies on a set-based presentation of the proxy behavior and employs an edit distance to compute an alignment. To extract the proxy behavior, we use the techniques defined in [22]. First, we discover a Petri net from each log with the inductive miner (noise threshold 0.2). Then, we followed [22] to obtain the proxy log using five techniques: 1) Simulation: The proxy behavior is derived by simulating the process model. 2) Clustering: Using a K-medoid method, the traces are partitioned into K different clusters using control-flow information. Then, from each cluster, a single trace is chosen as a cluster representative. The value of K is dynamic, calculated as 1/10 of the unique number of variants in the event log. 3) Random: Traces are randomly selected for the proxy behavior. 4) Frequency: The traces are selected based on their frequency in the log. 5) Reduced: First, a subset of activities is chosen (activity selection). Then, the proxy behavior is generated from traces that contain these activities. We used the implementation of [22] to generate the proxy behavior.

Under all configurations, we employed the same sample of 100 traces from the input log. All these traces turned out to represent different trace variants. For our trie-based approach, the search *budget* was set to $100K$ and meta-heuristic optimization was done with a *frequency* parameter of 100. Then, we kept changing the configuration until either a low error was achieved or the runtime of the baseline was exceeded. Runtime measurements were averaged over five runs on a system with an Intel Core i7-10510U CPU at 1.80GHz with 16GB RAM on Windows 10 with Oracle Java 1.8.

Metrics. We assess the deviation in alignment cost of our approach, compared to the baseline, using the mean absolute error (MAE). Also, we compare runtimes (in ms). However, the way our approach finds alignments is fundamentally different from the baseline. In our approach, we follow synchronous moves without exploring possibilities of asynchronous ones. This leads to longer prefixes and, respectively, infixes matches between the input trace and the proxy behavior, but might lead to much different suffixes. This, in turn, will affect the alignment cost. We later demonstrate this effect through concrete examples. Finally, we study the deviation distribution [3] across alignments found by our approach and the baseline.

Datasets. Our experiments were based on five real-world datasets. Four of them originate from the BPI challenges: BPI 2012,[1] BPI 2015,[2] BPI 2017,[3] and BPI 2019.[4] Additionally, an event log containing events of sepsis cases was used.[5]

[1] https://doi.org/10.4121/uuid:3926db30-f712-4394-aebc-75976070e91f
[2] https://doi.org/10.4121/uuid:31a308ef-c844-48da-948c-305d167a0ec1
[3] https://doi.org/10.4121/uuid:5f3067df-f10b-45da-b98b-86ae4c7a310b
[4] https://doi.org/10.4121/uuid:d06aff4b-79f0-45e6-8ec8-e19730c248f1
[5] https://doi.org/10.4121/uuid:915d2bfb-7e84-49ad-a286-dc35f063a460

	Proxy behav.	#Traces	Min. trace	Max. trace	Avg. trace	#Events	Trie size	Reduct.	Trie constr. (ms)
BPI 2012	Sim.	10K	4	29	19	194373	114355	41%	2165
	Clust.	3930	6	175	38	166513	56929	66%	1199
	Random	3950	3	175	38	164560	55446	66%	1022
	Freq.	436	3	127	36	15981	6837	57%	138
	Reduced	4367	3	175	38	182084	60373	67%	1196
	Sampled	100	3	127	38				
BPI 2015	Sim.	10K	15	41	28	289963	237351	18%	2562
	Clust.	140	3	81	44	6272	5073	19%	80
	Random	140	3	81	45	6302	5106	19%	66
	Freq.	15	34	71	52	783	677	14%	9
	Reduced	156	34	71	52	7014	5643	20%	93
	Sampled	100	21	71	50				
BPI 2017	Sim.	10K	6	41	25	261953	153691	41%	3793
	Clust.	14338	10	180	48	699751	259656	63%	9132
	Random	14422	10	180	48	698997	257857	63%	8750
	Freq.	15931	12	180	45	71818	32373	55%	1256
	Reduced	15931	10	180	48	768942	276914	64%	9555
	Sampled	100	10	44	180				
BPI 2019	Sim.	10K	2	28	7	85712	16621	81%	1616
	Clust.	10782	2	990	17	301550	188941	37%	2661
	Random	10848	1	990	11	302048	190647	37%	1850
	Freq.	1197	1	923	14	36909	28521	23%	212
	Reduced	11974	1	990	13	338226	213316	37%	2532
	Sampled	100	1	990	13				
SEPSIS	Sim.	10K	1	11	3	56723	19399	66%	1406
	Clust.	85	3	59	8	1082	544	50%	58
	Random	766	3	185	8	12482	6104	51%	236
	Freq.	84	3	24	8	1046	522	50%	23
	Reduced	847	3	185	8	13745	6605	48%	1216
	Sampled	100	3	59	8				

TABLE I

PROXY BEHAVIOR AND RESPECTIVE TRIES: SIZE AND CONSTRUCTION TIME, FOR THE DATASETS USING THE DIFFERENT GENERATION ALGORITHMS

Table I describes the generated proxy behavior and the sampled logs. Also, we report the number of nodes in the respective trie (size) and the reduction compared to the set-based representation of the proxy behavior used by the baseline.

Implementation. We implemented our approach to approximate conformance checking in Java. The source code along with data and experimental results is available on Github.[6] For the baseline, we used the implementation presented in [22]. However, the source code is included in our repository, as we had to isolate the parts of the code that are relevant to our experiments. To enable a fair comparison, for the baseline approach, we terminate the computation for a trace once a perfect alignment with an execution sequence in the proxy behavior was found. To ensure repeatability of the experiments, we fixed a seed for the random number generator that is invoked in the meta-heuristic optimization.

B. Experimental Results

Table II summarizes the runtime and MAE of our approach in comparison to the baseline. For each log, we report the performance for the different proxy behavior generation methods. For our approach, we report at most three different configurations of the budget and frequency to alternate between exploitation and exploration. The default configuration allows a maximum of $100K$ iterations and a jump to exploration every 100 iterations. Next, we change the alternation frequency and/or the budget. We stop in case we get a MAE under 5 or we exceed the runtime of the baseline approach. The best results per setting are highlighted in bold font.

In most of the cases, the default configuration is at least one order of magnitude faster than the baseline with a MAE close to 0. Upon investigation, there is a large overlap between the behavior in the sampled log and the proxy behavior. In this case, the strength of logarithmic search space reduction of using tries pays off and our approach is able to compute the alignment much faster. This is the case for the BPI 2012, 2019, 2017, and the SEPSIS logs.

[6] https://github.com/DataSystemsGroupUT/ConformanceCheckingUsingTries/

	Proxy behav.	Edit Dist. Runtime	Trie Conf. 1 Error	Trie Conf. 1 Runtime	Trie Conf. 2 Error	Trie Conf. 2 Runtime	Trie Conf. 3 Error	Trie Conf. 3 Runtime
BPI 2012	Sim.	354887	**100K/100**					
			0.98	**24750**				
	Clust.	337212	100K/100		100K/29		1.5M/97	
			4.3	5526	3.8	5785	3	331199
	Random	44748	100K/100		**100K/29**			
			0.6	531	**0.4**	**503**		
	Freq.	5319	**100K/100**					
			0.88	**1886**				
	Reduced	25045	**100K/100**					
			0	**32**				
BPI 2015	Sim.	136903	100K/100		500K/100		8M/1K	
			9.56	1936	9.3	8216	8.68	132757
	Clust.	3088	100K/100		1M/100k/29			
			12.75	2141	12.06	2785		
	Random	701	**100K/100**					
			0	**38**				
	Freq.	476	100K/100		30K/19		6M/19	
			8.1	209	7.93	353	1	858992
	Reduced	743	**100K/100**					
			0	**17**				
BPI 2017	Sim.	468457	**100K/100**					
			0	**15876**				
	Clust.	1169135	100K/100		100K/29		5M/97	
			6.89	3120	6.88	3915	5.6	437403
	Random	250833	**100K/100**					
			0.35	**868**				
	Freq.	29298	**100K/100**					
			1.09	**817**				
	Reduced	272785	**100K/100**					
			0	**1572**				
BPI 2019	Sim.	632717	**100K/100**					
			0	**10065**				
	Clust.	2100000	100K/100		100K/29		3M/13	
			9	4682	8	14995	3.28	1199590
	Random	720000	100K/100		**100K/29**			
			1.3	533	**0.7**	**792**		
	Freq.	27940	100K/100		100K/29		**2M/13**	
			0.3	390	0.18	446	**0**	**15292**
	Reduced	48312	**100K/100**					
			0	**15**				
SEPSIS	Sim.	16370	100K/100		**100K/29**			
			0.09	6162	**0.05**	**4905**		
	Clust.	346	**100K/100**					
			0	**24**				
	Random	1756	**100K/100**					
			0.01	**44**				
	Freq.	348	**100K/100**					
			0.2	**329**				
	Reduced	1292	**100K/100**					
			0	**7**				

TABLE II

RUNTIME AND MAE OF THE TRIE-BASED APPROACH AGAINST THE EDIT-DISTANCE-BASED APPROACH FOR ALIGNMENT COMPUTATION

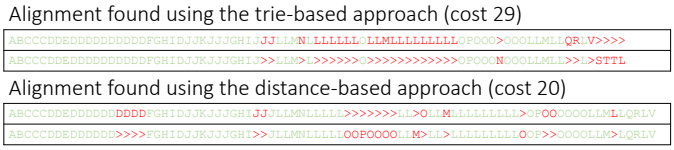


Fig. 4. Alignments found by the different algorithms

In some cases, the default configuration comes with a relatively high MAE. For instance, using simulation or clustering on the BPI 2015 log results in a MAE of around 10 and 13, respectively. The error is slightly enhanced by increasing the budget and allowing more exploration, e.g., using $8M/1K$, for simulated proxy behavior of BPI 2015.

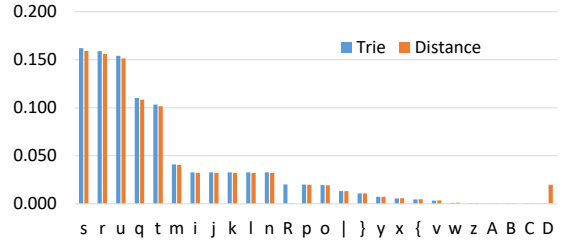
A similar, relatively high MAE occurs with BPI 2017 and BPI 2019 using simulation and clustering. However, the error improvement is much better. For the case of BPI 2017, using clustering, the error goes down to 5.6. In our experiments, we achieved lower errors, but for a much higher budget, so that the runtime exceeded the one of the baseline. In the case of BPI 2019, we can achieve a MAE below 1 with four orders of magnitude lower runtime than the baseline.

Our approach, in some cases, will construct alignments that are more expensive than those found by the baseline due to the emphasis on computing longer prefixes and infixes. For example, Figure 4 shows alignments found by our approach and the baseline for the same input trace. Clearly, the trie-based approach finds longer prefixes, which might lead to alignments with higher costs. However, these longer prefixes might also be more meaningful from a business perspective.

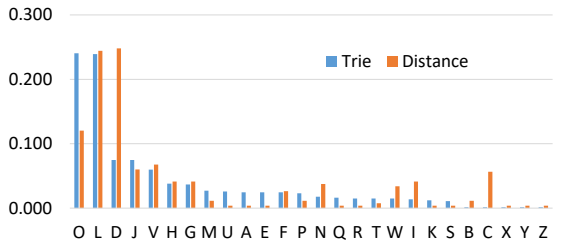
To further analyze the performance of the two approaches, we report the deviation distribution across activities that are involved in asynchronous moves. In Figure 5(a), we show the deviation distribution with an MAE of 0 (Table II) for proxy behavior obtained by simulation over BPI 2019. Here, the two approaches provide the same deviation distribution, except for a single outlier related to activity D.

Figure 5(b), in turn, shows differences in the distributions for proxy behavior obtained by clustering on BPI 2017. This can be attributed to the relatively large MAE of 7 (Table II). However, even in that case, several activities that are often involved in deviations can be identified accurately.

Both our approach and the baseline [22] are approximate conformance checking techniques that are based on some proxy behavior of the process model. Yet, the baseline technique will find an optimal alignment w.r.t this proxy behavior and, therefore, serves as a lower bound for the approximation error of our approach compared to the optimal alignment cost with the complete model behavior. To quantify the overall approximation error, we used the Rapid Miner tool to discover a Petri net (using heuristic miner) for the BPI 2015 log. The miner guarantees a fitting model, so that all traces of our sample have an alignment cost of zero, we used the ILP log replayer plugin of Rapid Miner. We compare these results against the alignment costs obtained with the approaches for approximate conformance checking based on proxy behavior.



(a) Proxy log obtained by simulation on BPI 2019 with MAE of 0



(b) Proxy log obtained by clustering on BPI 2017 with MAE of 7

Fig. 5. Deviation analysis results

Figure 6 illustrates the MAE for these approaches against the optimal alignments for the whole model behavior. Both approximation approaches are sensitive to the sampling of proxy behavior. While our approach increases the error for simulated, clustered, and frequency-based proxy behavior, it yields the same error for random and reduced proxy behavior, which also induces the smallest total error.

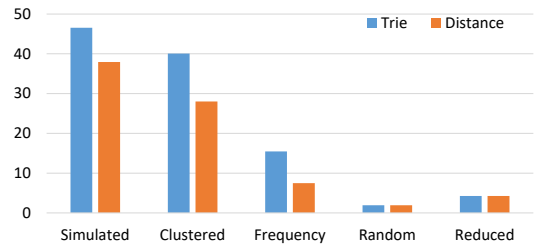


Fig. 6. MAE of our approach and the baseline compared to optimal alignments.

V. Related Work

Conformance checking may be based on trace replay [20], rule checking [32], or the computation of alignments [1]. Due to their ability to provide a complete characterization of conformance deviations, alignments serve as the basis for most contemporary techniques for conformance checking. To cope with the complexity of alignment construction, various optimizations have been suggested, including strategies to guide the search [29], [23], decomposition schemes [15], [13], or specific encodings of the search problem [4], [10]. Data structures for a compact representation of behavior have, so far, been largely neglected. A notable exception is conformance checking using a Deterministic Acyclic Finite State Automaton (DAFSA) [19], [18]. While the structure resembles the notion of a trie, it is employed to represent the behavior of the complete

log *and* the model to speed up the computation of *all* optimal alignments. Our work employs the tries for the proxy behavior of a model, *independent* of a log, to compute *one* alignment.

Heuristic approaches to approximate conformance checking, again, rely on decomposition schemes [13] and specific encodings [17]. It was also suggested to consider relaxations of the order of events [25] or estimate bounds from existing conformance results [2], [3]. However, a reduction of the size of the considered behavior provides another angle for efficient conformance checking, e.g., by sampling traces from a log [2], [3] or by sampling execution sequences from a process model [22], [21]. We follow the latter approach and extract proxy behavior from a model. Unlike existing approaches, however, we employ a compact representation of the proxy behavior. As demonstrated, this reduces the runtime by up to two orders of magnitude with a modest estimation error.

Techniques for approximate conformance checking are also particularly relevant for online scenarios, where data is available as an event stream. Here, existing work exploits a tailored token replay technique [31] or computes prefix alignments to handle events as they arrive [30]. Other work pre-computes possible conformance issues for online processing [6] or checks patterns derived from a model [7], [33]. As we demonstrated above, our approach to approximate conformance checking based on trie data structures can directly be lifted to such online scenarios.

VI. Conclusion & Future Work

In this paper, we have presented an efficient approach using tries data structures to compute alignments of logs with reference processes. The source of efficiency is the compact representation of the proxy behavior of the process as well as the utilization of tries to reduce the search space. Moreover, we applied meta-heuristics techniques to speed up the reach of better alignments. Experimental evaluation shows promising results compared to other string-based alignment computation.

In future work, we plan to study the effect of using different guiding cost functions and lift our approach to partially ordered events [26] and stochastic conformance [14].

Acknowledgement. Ahmed Awad's work is funded by the European Regional Development Funds via the Mobilitas Plus programme (grant MOBTT75).

References

[1] A. Adriansyah, B. F. van Dongen, and W. M. P. van der Aalst. Conformance checking using cost-based fitness analysis. In *EDOC*, pp. 55–64. IEEE, 2011.

[2] M. Bauer, H. van der Aa, and M. Weidlich. Estimating process conformance by trace sampling and result approximation. In *BPM*, *LNCS* 11675, pp. 179–197. Springer, 2019.

[3] M. Bauer, H. van der Aa, and M. Weidlich. Sampling and approximation techniques for efficient process conformance checking. *Inf. Syst.*, 2020.

[4] M. Boltenhagen, T. Chatain, and J. Carmona. Optimized SAT encoding of conformance checking artefacts. *Computing*, 103(1):29–50, 2021.

[5] J. C. A. M. Buijs, B. F. van Dongen, and W. M. P. van der Aalst. A genetic algorithm for discovering process trees. In *CEC*, pp. 1–8. 2012.

[6] A. Burattin and J. Carmona. A framework for online conformance checking. In *BPM Workshops*, *LNBIP* 308, pp. 165–177. Springer, 2017.

[7] A. Burattin, S. J. van Zelst, A. Armas-Cervantes, B. F. van Dongen, and J. Carmona. Online conformance checking using behavioural patterns. In *BPM*, *LNCS* 11080, pp. 250–267. Springer, 2018.

[8] J. Carmona, B. F. van Dongen, A. Solti, and M. Weidlich. *Conformance Checking - Relating Processes and Models*. Springer, 2018.

[9] F. Caron, J. Vanthienen, and B. Baesens. Comprehensive rule-based compliance checking and risk management with process mining. *DSS*, 54(3):1357–1369, 2013.

[10] M. de Leoni and A. Marrella. How planning techniques can help process mining: The conformance-checking case. In *Italian Symp. on Adv. Database Systems*, *CEUR* 2037, page 283. CEUR-WS.org, 2017.

[11] M. Feurer, A. Klein, K. Eggensperger, J. T. Springenberg, M. Blum, and F. Hutter. Efficient and robust automated machine learning. In *NIPS*, pp. 2755–2763. MIT Press, 2015.

[12] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Syst. Science and Cybernet.*, 4(2):100–107, 1968.

[13] W. L. J. Lee, H. M. W. Verbeek, J. Munoz-Gama, W. M. P. van der Aalst, and M. Sepúlveda. Recomposing conformance: Closing the circle on decomposed alignment-based conformance checking in process mining. *Inf. Sci.*, 466:55–91, 2018.

[14] S. J. J. Leemans and A. Polyvyanyy. Stochastic-aware conformance checking: An entropy-based approach. In *CAiSE*, *LNCS* 12127, pp. 217–233. Springer, 2020.

[15] J. Munoz-Gama, J. Carmona, and W. M. P. van der Aalst. Single-entry single-exit decomposed conformance checking. *Inf. Syst.*, 46:102–122, 2014.

[16] G. Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88, 2001.

[17] J. Peeperkorn, S. vanden Broucke, and J. D. Weerdt. Conformance checking using activity and trace embeddings. In *BPM Forum*, *LNBIP* 392, pp. 105–121. Springer, 2020.

[18] D. Reißner, A. Armas-Cervantes, R. Conforti, M. Dumas, D. Fahland, and M. L. Rosa. Scalable alignment of process models and event logs: An approach based on automata and s-components. *Inf. Syst.*, 94:101561, 2020.

[19] D. Reißner, R. Conforti, M. Dumas, M. L. Rosa, and A. Armas-Cervantes. Scalable conformance checking of business processes. In *OTM*, *LNCS* 10573, pp. 607–627. Springer, 2017.

[20] 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.

[21] M. F. Sani, J. J. G. Gonzalez, S. J. van Zelst, and W. M. P. van der Aalst. Conformance checking approximation using simulation. In *ICPM*, pp. 105–112. IEEE, 2020.

[22] M. F. Sani, S. J. van Zelst, and W. M. P. van der Aalst. Conformance checking approximation using subset selection and edit distance. In *CAiSE*, *LNCS* 12127, pp. 234–251. Springer, 2020.

[23] A. Syamsiyah and B. F. van Dongen. Improving alignment computation using model-based preprocessing. In *ICPM*, pp. 73–80. IEEE, 2019.

[24] E. Talbi. *Metaheuristics - From Design to Implementation*. Wiley, 2009.

[25] F. Taymouri and J. Carmona. A recursive paradigm for aligning observed behavior of large structured process models. In *BPM*, *LNCS* 9850, pp. 197–214. Springer, 2016.

[26] H. van der Aa, H. Leopold, and M. Weidlich. Partial order resolution of event logs for process conformance checking. *DSS*, 136:113347, 2020.

[27] W. M. P. van der Aalst. *Process Mining - Data Science in Action, Second Edition*. Springer, 2016.

[28] W. M. P. van der Aalst, A. Adriansyah, and B. F. van Dongen. Replaying history on process models for conformance checking and performance analysis. *DMKD*, 2(2):182–192, 2012.

[29] B. van Dongen. Efficiently computing alignments - using the extended marking equation. *BPM*, *LNCS* 11080, pp. 197–214. Springer, 2018.

[30] S. J. van Zelst, A. Bolt, M. Hassani, B. F. van Dongen, and W. M. P. van der Aalst. Online conformance checking: relating event streams to process models using prefix-alignments. *Int. J. Data Sci. Anal.*, 8(3):269–284, 2019.

[31] S. K. L. M. vanden Broucke, J. Munoz-Gama, J. Carmona, B. Baesens, and J. Vanthienen. Event-based real-time decomposed conformance analysis. In *OTM*, *LNCS* 8841, pp. 345–363. Springer, 2014.

[32] M. Weidlich, A. Polyvyanyy, N. Desai, J. Mendling, and M. Weske. Process compliance analysis based on behavioural profiles. *Inf. Syst.*, 36(7):1009–1025, 2011.

[33] M. Weidlich, H. Ziekow, J. Mendling, O. Günther, M. Weske, and N. Desai. Event-based monitoring of process execution violations. In *BPM*, *LNCS* 6896, pp. 182–198. Springer, 2011.