Online Process Monitoring Using Incremental State-Space Expansion: An Exact Algorithm

Daniel Schuster[1(✉)] and Sebastiaan J. van Zelst[1,2]

[1] Fraunhofer Institute for Applied Information Technology FIT, Sankt Augustin, Germany
{daniel.schuster,sebastiaan.van.zelst}@fit.fraunhofer.de
[2] RWTH Aachen University, Aachen, Germany
s.j.v.zelst@pads.rwth-aachen.de

Abstract. The execution of (business) processes generates valuable traces of event data in the information systems employed within companies. Recently, approaches for monitoring the correctness of the execution of running processes have been developed in the area of process mining, i.e., online conformance checking. The advantages of monitoring a process' conformity during its execution are clear, i.e., deviations are detected as soon as they occur and countermeasures can immediately be initiated to reduce the possible negative effects caused by process deviations. Existing work in online conformance checking only allows for obtaining approximations of non-conformity, e.g., overestimating the actual severity of the deviation. In this paper, we present an exact, parameter-free, online conformance checking algorithm that computes conformance checking results on the fly. Our algorithm exploits the fact that the conformance checking problem can be reduced to a shortest path problem, by incrementally expanding the search space and reusing previously computed intermediate results. Our experiments show that our algorithm is able to outperform comparable state-of-the-art approximation algorithms.

Keywords: Process mining · Conformance checking · Alignments · Event streams · Incremental heuristic search

1 Introduction

Modern information systems support the execution of different business processes within companies. Valuable traces of *event data*, describing the various steps performed during process execution, are easily extracted from such systems. The field of *process mining* [3] aims to exploit such information, i.e., the event data, to better understand the overall execution of the process. For example, in process mining, several techniques have been developed that allow us to *(i)* automatically discover process models, *(ii)* compute whether the process, as

D. Fahland et al. (Eds.): BPM 2020, LNCS 12168, pp. 147–164, 2020.
https://doi.org/10.1007/978-3-030-58666-9_9

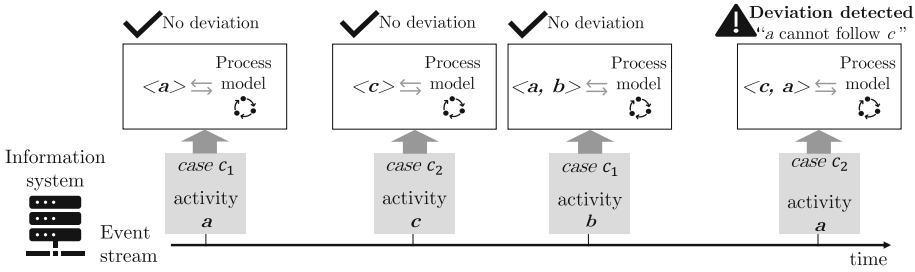


Fig. 1. Overview of online process monitoring. Activities are performed for different process instances, identified by a case-id, over time. Whenever a new activity is executed, the sequence of already executed activities within the given case/process instance is checked for conformance w.r.t. a reference process model

reflected by the data, conforms to a predefined reference model and *(iii)* detect performance deficiencies, e.g., bottleneck detection.

The majority of existing process mining approaches work in an offline setting, i.e., data is captured over time during process execution and process mining analyses are performed a posteriori. However, some of these techniques benefit from an *online application scenario*, i.e., analyzing the process at the moment it is executed. Reconsider *conformance checking*, i.e., computing whether a process' execution conforms to a reference model. Checking conformance in an online setting allows the process owner to detect and counteract non-conformity at the moment it occurs (Fig. 1). Thus, potential negative effects caused by a process deviation can be mitigated or eliminated. This observation inspired the development of novel conformance checking algorithms working in an online setting [7,8,22]. However, such algorithms provide approximations of non-conformity and/or use high-level abstractions of the reference model and the event data, i.e., not allowing us to obtain an exact quantification of non-conformance.

In this paper, we propose a novel, *exact* solution for the online conformance checking problem. We present a *parameter-free* algorithm that computes exact conformance checking results and provides an exact quantification of non-conformance. Our algorithm exploits the fact that the computation of conformance checking results can be reduced to a shortest path problem. In fact, we extend the search space in the course of a process instance execution and compute shortest paths by utilizing previous results every time new behavior is observed. Moreover, we explicitly exploit specific properties of the search space when solving the conformance checking problem. Therefore, the proposed incremental algorithm is specifically designed for online conformance checking and cannot be directly applied to general shortest path problems. The conducted experiments show that the proposed approach outperforms existing approximation algorithms and additionally guarantees exact results.

The remainder of this paper is structured as follows. In Sect. 2, we present related work regarding conformance checking and incremental search algorithms. In Sect. 3, we present preliminaries. In Sect. 4, we present the main algorithm.

In Sect. 5, we prove the correctness of the proposed algorithm. We evaluate the proposed algorithm and present the results of the experiments conducted in Sect. 6. Finally, we conclude the paper in Sect. 7.

2 Related Work

In this section, we first focus on (online) conformance checking techniques. Subsequently, we present related work regarding incremental search algorithms.

Two early techniques, designed to compute conformance statistics, are token-based replay [21] that tries to replay the observed behavior on a reference model and footprint-based comparison [3], in which the event data and the process model are translated into the same abstraction and then compared. As an alternative, *alignments* have been introduced [2,4] that map the observed behavioral sequences to a feasible execution sequence as described by the (reference) process model. Alignments indicate whether behavior is missing and/or whether inappropriate behavior is observed. The problem of finding an alignment was shown to be reducible to the shortest path problem [4].

The aforementioned techniques are designed for offline usage, i.e., they work on static (historical) data. In [22], an approach is presented to monitor ongoing process executions based on an event stream by computing partially completed alignments each time a new event is observed. The approach results in approximate results, i.e., false negatives occur w.r.t. deviation detection. In this paper, we propose an approach that extends and improves [22]. In [7], the authors propose to pre-calculate a transition system that supports replay of the ongoing process. Costs are assigned to the transition system's edges and replaying a deviating process instance leads to larger (non-zero) costs. Finally, [8] proposes to compute conformance of a process execution based on all possible behavioral patterns of the activities of a process. However, the use of such patterns leads to a loss of expressiveness in deviation explanation and localization.

In general, incremental search algorithms find shortest paths for similar search problems by utilizing results from previously executed searches [15]. In [13], the Lifelong Planning A^* algorithm is introduced that is an incremental version of the A^* algorithm. The introduced algorithm repeatedly calculates a shortest path from a fixed start state to a fixed goal state while the edge costs may change over time. In contrast, in our approach, the goal states are constantly changing in each incremental execution, whereas the edge costs remain fixed. Moreover, only new edges and vertices are incrementally added, i.e., the already existing state space is only extended. In [14], the Adaptive A^* algorithm is introduced, which is also an incremental version of the A^* algorithm. The Adaptive A^* algorithm is designed to calculate a shortest path on a given state space from an incrementally changing start state to a fixed set of goal states. In contrast to our approach, the start state is fixed in each incremental execution.

Table 1. Example *event log* fragment

Case	Activity	Resource	Time-stamp
...	...	...	...
13152	Create account (a)	*Wil*	*19-04-08 10:45*
13153	Create account (a)	*Bas*	*19-04-08 11:12*
13154	Request quote (c)	*Daniel*	*19-04-08 11:14*
13155	Request quote (c)	*Daniel*	*19-04-08 11:40*
13152	Submit order (b)	*Wil*	*19-04-08 11:49*
...	...	...	...

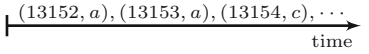


Fig. 2. Schematic example of an event stream

3 Background

In this section, we present basic notations and concepts used within this paper.

Given a set X, a multiset B over X allows us to assign a multiplicity to the elements of X, i.e., $B\colon X \to \mathbb{N}_0$. Given $X = \{x, y, z\}$, the multiset $[x^5, y]$ contains 5 times x, once y and no z. The set of all possible multisets over a set X is denoted by $\mathcal{B}(X)$. We write $x \in_+ B$ if x is contained at least once in multiset B.

A sequence σ of length n, denoted by $|\sigma| = n$, over a base set X assigns an element to each index, i.e., $\sigma\colon \{1, \ldots, n\} \to X$. We write a sequence σ as $\langle \sigma(1), \sigma(2), ..., \sigma(|\sigma|) \rangle$. Concatenation of sequences is written as $\sigma{\cdot}\sigma'$, e.g., $\langle x, y \rangle{\cdot}\langle z \rangle = \langle x, y, z \rangle$. The set of all possible sequences over base set X is denoted by X^*. For element inclusion, we overload the notation for sequences, i.e., given $\sigma \in X^*$ and $x \in X$, we write $x \in \sigma$ if $\exists\, 1 \leq i \leq |\sigma|\, (\sigma(i) = x)$, e.g., $b \in \langle a, b \rangle$.

Let $\sigma \in X^*$ and let $X' \subseteq X$. We recursively define $\sigma_{\downarrow_{X'}} \in X'^*$ with: $\langle\rangle_{\downarrow_{X'}} = \langle\rangle$, $(\langle x \rangle{\cdot}\sigma)_{\downarrow_{X'}} = \langle x \rangle{\cdot}\sigma_{\downarrow_{X'}}$ if $x \in X'$ and $(\langle x \rangle{\cdot}\sigma)_{\downarrow_{X'}} = \sigma_{\downarrow_{X'}}$ if $x \notin X'$. For example, let $X' = \{a, b\}$, $X = \{a, b, c\}$, $\sigma = \langle a, c, b, a, c \rangle \in X^*$ then $\sigma_{\downarrow_{X'}} = \langle a, b, a \rangle$.

Let $t = (x_1, ..., x_n) \in X_1 \times \cdots \times X_n$ be an n-tuple, we let $\pi_1(t) = x_1, \ldots, \pi_n(t) = x_n$ denote the corresponding projection functions that extract a specific component from the tuple, e.g., $\pi_3((a, b, c)) = c$. Correspondingly, given a sequence $\sigma = \langle (x_1^1, \ldots, x_n^1), \ldots, (x_1^m, \ldots, x_n^m) \rangle$ with length m containing n-tuples, we define projection functions $\pi_1^*(\sigma) = \langle x_1^1, \ldots, x_1^m \rangle, \ldots, \pi_n^*(\sigma) = \langle x_n^1, \ldots, x_n^m \rangle$ that extract a specific component from each tuple and concatenate it into a sequence. For instance, $\pi_2^* \left(\langle (a, b), (c, d), (c, b) \rangle \right) = \langle b, d, b \rangle$.

Event Logs. The data used in process mining are *event logs*, e.g., consider Table 1. Each row corresponds to an *event* describing the execution of an activity in the context of an *instance* of the process. For simplicity, we use short-hand activity names, e.g., a for "create account". The events related to *Case-id* 13152 describe the activity sequence $\langle a, b \rangle$.

Event Streams. In this paper, we assume an *event stream* rather than an event log. Conceptually, an event stream is an (infinite) sequence of events. In Fig. 2, we depict an example. For instance, the first event, $(13152, a)$, indicates that for a process instance with case-id 13152 activity a was performed.

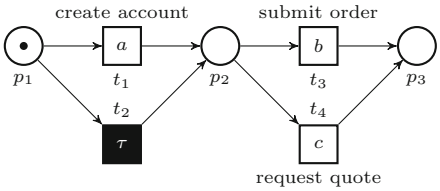


Fig. 3. Example WF-net N_1 with visualized initial marking $[p_1]$ and final marking $[p_3]$ describing a simplified ordering process. First "create account" is optionally executed. Next, either "submit order" or "request quote" is executed

Definition 1 (Event; Event Stream). *Let $\mathcal{C}$ denote the universe of* case identifiers *and $\mathcal{A}$ the universe of* activities. *An event $e \in \mathcal{C} \times \mathcal{A}$ describes the execution of an activity $a \in \mathcal{A}$ in the context of a process instance identified by $c \in \mathcal{C}$. An event stream S is a sequence of events, i.e., $S \in (\mathcal{C} \times \mathcal{A})^*$.*

As indicated in Table 1, real-life events contain additional information, e.g., resource information, and are usually uniquely identifiable by an event id. However, for the purpose of this paper, we are only interested in the executed activity, the case-id of the corresponding process instance and the order of events.

Process Models. Process models allow us to describe the (intended) behavior of a process. In this paper, we focus on *sound Workflow nets* [1]. A Workflow net (WF-net) is a subclass of *Petri nets* [20]. Sound WF-nets, in turn, are a subclass of WF-nets with favorable *behavioral properties*, e.g., no deadlocks and livelocks. Consider Fig. 3, where we depict a sound WF-net. We use WF-nets since many high-level process modeling formalism used in practice, e.g. BPMN [10], are easily translated into WF-nets. Moreover, it is reasonable to assume that an experienced business process designer creates sound process models.

Petri nets consist of a set of *places* P, visualized as circles, and a set of *transitions* T, visualized as rectangular boxes. Places and transitions are connected by arcs which are defined by the set $F = (P \times T) \cup (T \times P)$. Given an element $x \in P \cup T$, we write $x\bullet = \{y \in P \cup T \mid (x, y) \in F\}$ to define all elements y that have an incoming arc from x. Symmetrically, $\bullet x = \{y \in P \cup T \mid (y, x) \in F\}$, e.g., $\bullet p_2 = \{t_1, t_2\}$ (Fig. 3).

The state of a Petri net, i.e., a *marking* M, is defined by a multiset of places, i.e., $M \in \mathcal{B}(P)$. Given a Petri net N with a set of places P and a marking $M \in \mathcal{B}(P)$, a *marked net* is written as (N, M). We denote the initial marking of a Petri net with M_i and the final marking with M_f. We denote a Petri net as $N = (P, T, F, M_i, M_f, \lambda)$. The labeling function $\lambda\colon T \to \mathcal{A} \cup \{\tau\}$ assigns an (possibly invisible, i.e., τ) activity label to each transition, e.g., $\lambda(t_1) = a$ in Fig. 3.

The transitions of a Petri net allow to change the state. Given a marking $M \in \mathcal{B}(P)$, a transition t is *enabled* if $\forall p \in \bullet t\,(M(p) > 0)$. An enabled transition can *fire*, yielding marking $M' \in \mathcal{B}(P)$, where $M'(p) = M(p) + 1$ if $p \in t\bullet \setminus \bullet t$, $M'(p) = M(p) - 1$ if $p \in \bullet t \setminus t\bullet$, otherwise $M'(p) = M(p)$. We write $(N, M)[t\rangle$ if

Fig. 4. Three possible alignments for WF-net N_1 (Fig. 3) and trace $\langle a, b, c \rangle$

t is enabled in M and we write $(N, M) \xrightarrow{t} (N, M')$ to denote that firing transition t in marking M yields marking M'. In Fig. 3, we have $(N_1, [p_1])[t_1\rangle$ as well as $(N_1, [p_1]) \xrightarrow{t_1} (N_1, [p_2])$. If a sequence of transitions $\sigma \in T^*$ leads from marking M to M', we write $(N, M) \xrightarrow{\sigma} (N, M')$. We let $\mathcal{R}(N, M) = \{M' \in \mathcal{B}(P) \mid \exists \sigma \in T^* (N, M) \xrightarrow{\sigma} (N, M')\}$ denote the state space/all reachable markings of N given an initial marking M.

A *WF-net* $N = (P, T, F, [p_i], [p_o], \lambda)$ is a Petri net with a unique source place p_i and a unique sink place p_o, i.e., $M_i = [p_i]$ and $M_f = [p_o]$. Moreover, every element $x \in P \cup T$ is on a path from p_i to p_o.

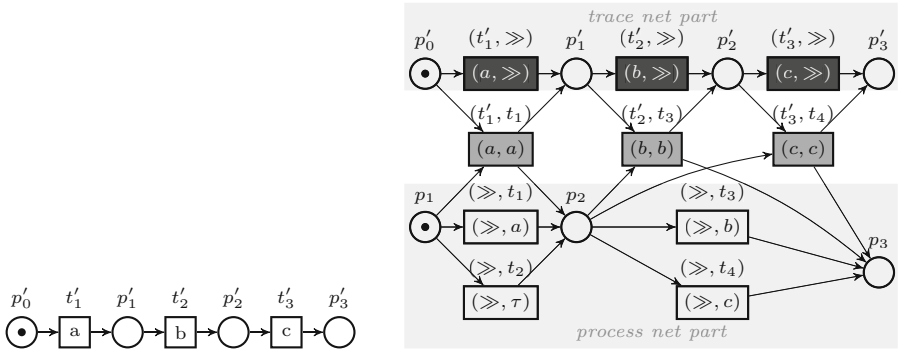
Alignments. To explain traces in an event log w.r.t. a reference model, we use *alignments* [4], which map a trace onto an execution sequence of a model. Exemplary alignments are depicted in Fig. 4. The first row of an alignment (ignoring the skip symbol $\gg$) equals the trace and the second row (ignoring $\gg$) represents a sequence of transitions leading from the initial to the final marking.

We distinguish three types of *moves* in an alignment. A *synchronous move* (light-gray) matches an observed activity to the execution of a transition, where the transition's label must match the activity. *Log moves* (dark-gray) indicate that an activity is not re-playable in the current state of the process model. *Model moves* (white) indicate that the execution of a transition cannot be mapped onto an observed activity. They can be further differentiated into *invisible*- and *visible model moves*. An invisible model move consists of an inherently invisible transition ($\lambda(t) = \tau$). Visible model moves indicate that an activity should have taken place w.r.t the model but was not observed at that time.

In an online setting, an event stream is assumed to be infinite. A new event for a given process instance can occur at any time. Hence, we are interested in explanations of the observed behavior that still allow us to reach the final state in the reference model, i.e., *prefix-alignments*. The first row of a prefix-alignment also corresponds to the trace, but the second row corresponds to a sequence of transitions leading from the initial marking to a marking from which the final marking can still be reached. For a formal definition, we refer to [4].

Since multiple (prefix-)alignments exist, we are interested in an alignment that minimizes the mismatches between the trace and the model. Therefore, we assign costs to moves. We use the *standard cost function*, which assigns cost 0 to synchronous moves and invisible model moves, and cost 1 to log- and visible model moves. A (prefix-)alignment is optimal if it has minimal costs.

To compute an optimal (prefix-)alignment, we search for a *shortest path* in the state-space of the *synchronous product net (SPN)* [4]. An SPN is composed of a trace net and the given WF-net. In Fig. 5a, we depict an example trace net. We refer to [4] for a formal definition of the trace net. In Fig. 5b



(a) Trace net of the trace $\langle a, b, c \rangle$ **(b)** SPN N_1^S of $\langle a, b, c \rangle$ and the WF-net N_1

Fig. 5. Construction of a trace net and a synchronous product net (SPN)

we depict an example SPN. Each transition in the SPN corresponds to a (prefix-)alignment move. Hence, we can assign costs to each transition. Any path in the state-space (sequence of transitions in the SPN) from $[p'_0, p_1]$ to $[p'_3, p_3]$ corresponds to an alignment of N_1 and $\langle a, b, c \rangle$. For the given example, a shortest path with cost 1, which corresponds to the first alignment depicted in Fig. 4, is:

$$(N_1^S, [p'_0, p_1]) \xrightarrow{(t'_1, t_1)} (N_1^S, [p'_1, p_2]) \xrightarrow{(t'_2, \gg)} (N_1^S, [p'_2, p_2]) \xrightarrow{(t'_3, t_4)} (N_1^S, [p'_3, p_3])$$

To compute a *prefix-alignment*, we look for a shortest path from the initial marking to a marking $M \in \mathcal{R}(N_1^S, [p'_0, p_1])$ such that $M(p'_3) = 1$, i.e., the last place of the trace net part is marked. Next, we formally define the SPN.

Definition 2 (Synchronous Product Net (SPN)). *For a given trace σ, the corresponding trace net $N^\sigma = (P^\sigma, T^\sigma, F^\sigma, [p_i^\sigma], [p_o^\sigma], \lambda^\sigma)$ and a WF-net $N = (P, T, F, [p_i], [p_o], \lambda)$ s.t. $P^\sigma \cap P = \emptyset$ and $F^\sigma \cap F = \emptyset$, we define the SPN $N^S = (P^S, T^S, F^S, M_i^S, M_f^S, \lambda^S)$ s.t.:*

- $P^S = P^\sigma \cup P$
- $T^S = (T^\sigma \times \{\gg\}) \cup (\{\gg\} \times T) \cup \{(t', t) \in T^\sigma \times T \mid \lambda(t) = \lambda^\sigma(t') \neq \tau\}$
- $F^S = \{(p, (t', t)) \in P^S \times T^S \mid (p, t') \in F^\sigma \vee (p, t) \in F\} \cup \{((t', t), p) \in T^S \times P^S \mid (t', p) \in F^\sigma \vee (t, p) \in F\}$
- $M_i^S = [p_i^\sigma, p_i]$ *and* $M_f^S = [p_o^\sigma, p_o]$
- $\lambda^S : T^S \to (\mathcal{A} \cup \{\tau\} \cup \{\gg\}) \times (\mathcal{A} \cup \{\tau\} \cup \{\gg\})$ *(assuming* $\gg \notin \mathcal{A} \cup \{\tau\}$*) s.t.:*
  - $\lambda^S(t', \gg) = (\lambda^\sigma(t'), \gg)$ *for* $t' \in T^\sigma$
  - $\lambda^S(\gg, t) = (\gg, \lambda(t))$ *for* $t \in T$
  - $\lambda^S(t', t) = \lambda^S(\lambda^\sigma(t), \lambda(t))$ *for* $t' \in T^\sigma, t \in T$

Next, we briefly introduce the shortest path algorithm $A^\star$ since our proposed algorithm is based on it and it is widely used for alignment computation [4,9].

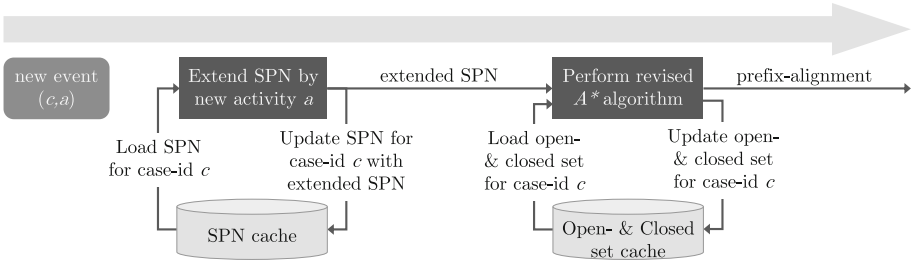


Fig. 6. Overview of the proposed incremental prefix alignment approach

***A*⋆ algorithm.** The $A^\star$ algorithm [12] is an informed search algorithm that computes a shortest path. It efficiently traverses a search-space by exploiting, for a given state, the *estimated remaining distance*, referred to as the heuristic/h-value, to the closest goal state. The algorithm maintains a set of states of the search-space in its so-called open-set O. For each state in O, a path from the initial state to such a state is known and hence, the distance to reach that state, referred to as the g value, is known. A state from O with minimal f-value, i.e., $f = g + h$, is selected for further analysis until a goal state is reached. The selected state itself is moved into the closed set C, which contains fully investigated states for which a shortest path to those states is known. Furthermore, all successor states of the selected state are added to the open set O. Note that the used heuristic must be *admissible* [12]. If the used heuristic also satisfies *consistency* [12], states need not be reopened.

4 Incremental Prefix-Alignment Computation

In this section, we present an exact algorithm to incrementally compute optimal prefix-alignments on an event stream. First, we present an overview of the proposed approach followed by a detailed description of the main algorithm.

4.1 Overview

The core idea of the proposed algorithm is to exploit previously calculated results, i.e., explored parts of the state-space of an SPN. For each process instance, we maintain an SPN, which is extended as new events are observed. After extending the SPN, we "continue" the search for an optimal prefix-alignment.

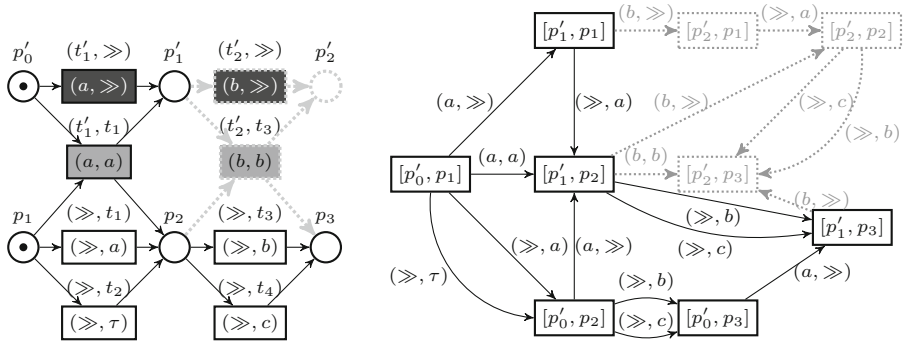
In Fig. 6 we visualize a conceptual overview of our approach. We observe a new event (c, a) on the event stream. We check our SPN cache and if we previously built an SPN for case c, we fetch it from the cache. We then extend the SPN by means of adding activity a to the trace net part. Starting from intermediate results of the previous search, i.e., open & closed set used in the $A^\star$ algorithm, we find a new, optimal prefix-alignment for case c.

Algorithm 1: Incremental Prefix-Alignment Computation

input: $N{=}(P,T,F,[p_i],[p_o],\lambda), S{\in}(\mathcal{C}{\times}\mathcal{A})^*$
begin
1 **forall** $c \in \mathcal{C}$ **do**
  $\mathcal{D}_\sigma(c) \leftarrow \langle\rangle$, $\mathcal{D}_C(c) \leftarrow \emptyset$; // initialize cache
2 $i \leftarrow 1$;
3 **while** *true* **do**
4 $e \leftarrow S(i)$; // get i-th event of event stream
5 $c \leftarrow \pi_1(e)$; // extract case-id from current event
6 $a \leftarrow \pi_2(e)$; // extract activity label from current event
7 $\mathcal{D}_\sigma(c) \leftarrow \mathcal{D}_\sigma(c){\cdot}\langle a\rangle$; // extend trace for case c
8 let $N^S{=}(P^S,T^S,F^S,M_i^S,M_f^S,\lambda^S)$ from N and $\mathcal{D}_\sigma(c)$; // construct/extend synchronous product net
9 let $h : \mathcal{R}(N^S,M_i^S){\rightarrow}\mathbb{R}_{\geq 0}$; // define heuristic function
10 let $d : T^S{\rightarrow}\mathbb{R}_{\geq 0}$; // define standard cost function
11 **if** $|\mathcal{D}_\sigma(c)|{=}1$ **then** // initialization for first run regarding case c
12 $\mathcal{D}_O(c) \leftarrow \{M_i^S\}$; // initialize open set
13 $\mathcal{D}_g(c) \leftarrow M_i^S{\mapsto}0$; // initialize cost-so-far function
14 $\mathcal{D}_p(c) \leftarrow M_i^S{\mapsto}(null,null)$; // initialize predecessor function
15 $\mathcal{D}_{\overline{\gamma}}(c),\mathcal{D}_O(c),\mathcal{D}_C(c),\mathcal{D}_g(c),\mathcal{D}_p(c) \leftarrow A^\star_{\text{inc}}(N^S,\mathcal{D}_\sigma(c),\mathcal{D}_O(c),\mathcal{D}_C(c),\mathcal{D}_g(c),\mathcal{D}_p(c),h,d)$; // execute/continue shortest path search
16 $i \leftarrow i+1$;

In Algorithm 1 we depict the overall algorithm. As input we assume a reference process model, i.e., a WF-net N, and an event stream S. The algorithm processes every event on the stream S in the order in which they occur. First, we extract the case id and the activity label from the current event. Next we either construct the SPN if it is the first activity for the given case or we extend the previously constructed SPN. For the SPN's state space we then define a heuristic function h and the standard cost function d. If we process the first activity for a given case, we initialize the open set O with the SPN's initial marking. Afterwards, we calculate a prefix alignment by calling a modified A^* algorithm, i.e., $A^\star_{\text{inc}}$. We obtain an optimal prefix-alignment $\overline{\gamma}$, open set O, closed set C, cost-so-far function g and the predecessor function p. The function g assigns to already discovered states the currently known cheapest costs and function p assigns the corresponding predecessor state to reach those. We cache the results to reuse them when computing the next prefix-alignment upon receiving a new event for the given case. Afterwards, we process the next event.

Note that, the approach theoretically requires infinite memory since it stores all intermediate results for all occurring cases because in general, we do not know when a case is completed in an online dimension. However, this is a general research challenge in process mining on streaming data, which is not addressed in this paper.



(a) Extension of the SPN **(b)** Resulting extension of the SPN's state space

Fig. 7. Incremental extension of the SPN for the process model N_1 and a trace that was extended by a new activity b, i.e., $\langle a \rangle \cdot \langle b \rangle$

The following sections are structured according to the overview shown in Fig. 6. First, we explain the SPN extension. Subsequently, we present a revised A^* algorithm to incrementally compute optimal prefix-alignments, i.e., $A^\star_{\text{inc}}$. Moreover, we present a heuristic function for the prefix-alignment computation.

4.2 Extending SPNs

Reconsider WF-net N_1 (Fig. 3) and assume that the first activity we observe is a. The corresponding SPN is visualized by means of the solid elements in Fig. 7a and the state space in Fig. 7b. Any state in the state-space containing a token in p'_1 is a suitable goal state of the $A^\star$ algorithm for an optimal prefix-alignment.

Next, for the same process instance, we observe an event describing activity b. The SPN for the process instance now describing trace $\langle a, b \rangle$ as well as its corresponding state-space is expanded. The expansion is visualized in Fig. 7 by means of dashed elements. In this case, any state that contains a token in p'_2 corresponds to a suitable goal state of the optimal prefix-alignment search.

4.3 Incrementally Performing Regular $A^\star$

Here, we present the main algorithm to compute prefix-alignments on the basis of previously executed instances of the $A^\star$ algorithm.

The main idea of our approach is to continue the search on an extended search space. Upon receiving a new event (c, a), we apply the regular $A^\star$ algorithm using the cached open- and closed-set for case identifier c on the corresponding extended SPN. Hence, we incrementally solve shortest path problems on finite, fixed state-spaces by using the regular $A^\star$ algorithm with pre-filled open and closed sets from the previous search. Note that the start state remains the same and only the goal states differ in each incremental step.

Algorithm 2: $A^{\star}_{\text{inc}}$ (modified A^{*} algorithm that computes prefix-alignments from pre-filled open and closed sets)

input: $N^S = (P^S, T^S, F^S, M_i^S, M_f^S, \lambda^S), O, C \subseteq \mathcal{R}(N^S, M_i^S)$,
$g\colon \mathcal{R}(N^S, M_i^S) \to \mathbb{R}_{\geq 0}, p\colon \mathcal{R}(N^S, M_i^S) \to T^S \times \mathcal{R}(N^S, M_i^S)$,
$h\colon \mathcal{R}(N^S, M_i^S) \to \mathbb{R}_{\geq 0}, d\colon T^S \to \mathbb{R}_{\geq 0}$

begin

1 let $p_{|\sigma|}$ be the last place of the trace net part of N^S;
2 **forall** $m \in \mathcal{R}(N^S, M_i^S) \setminus O \cup C$ **do** // initialize undiscovered states
3 $g(m) \leftarrow \infty$;
4 $f(m) \leftarrow \infty$;
5 **forall** $m \in O$ **do**
6 $f(m) = g(m) + h(m)$; // recalculate heuristic and update f-values
7 **while** $O \neq \emptyset$ **do**
8 $m \leftarrow \arg\min_{m \in O} f(m)$; // pop a state with minimal f-value from O
9 **if** $p_{|\sigma|} \in_{+} m$ **then**
10 $\overline{\gamma} \leftarrow$ prefix-alignment that corresponds to the sequence of transitions $(t_1, ..., t_n)$ where $t_n = \pi_1(p(m)), t_{n-1} = \pi_1(\pi_2(p(m)))$, etc. until there is a marking that has no predecessor, i.e., M_i^S;
11 **return** $\overline{\gamma}, O, C, g, p$;
12 $C \leftarrow C \cup \{m\}$;
13 $O \leftarrow O \setminus \{m\}$;
14 **forall** $t \in T^S$ *s.t.* $(N^S, m)[t\rangle(N^S, m')$ **do** // investigate successor states
15 **if** $m' \notin C$ **then**
16 $O \leftarrow O \cup \{m'\}$;
17 **if** $g(m) + d(t) < g(m')$ **then** // a cheaper path to m' was found
18 $g(m') \leftarrow g(m) + d(t)$; // update costs to reach m'
19 $f(m') \leftarrow g(m') + h(m')$; // update f-value of m'
20 $p(m') \leftarrow (t, m)$; // update predecessor of m'

In Algorithm 2, we present an algorithmic description of the $A^{\star}$ approach. The algorithm assumes as input an SPN, the open- and closed-set of the previously executed instance of the $A^{\star}$ algorithm, i.e., for the process instance at hand, a cost-so-far function g, a predecessor function p, a heuristic function h, and a cost function d (standard cost function). First, we initialize all states that have not been discovered yet (line 2). Since the SPN is extended and the goal states are different with respect to the previous run of the algorithm for the same process instance, all h-values are outdated. Hence, we recalculate the heuristic values and update the f-values for all states in the open set (line 6) because we are now looking for a shortest path to a state that has a token in the newly added place in the trace net part of the SPN. Hence, the new goal states were not present in the previous search problem. Note that the g values are not affected by the SPN extension. Thereafter, we pick a state from the open set

with smallest f-value (line 7). First, we check if the state is a goal state, i.e., whether it contains a token in the last place of the trace net part (line 9). If so, we reconstruct the sequence of transitions that leads to the state, and thus, we obtain a prefix-alignment (using predecessor function p). Otherwise, we move the current state from the open- to the closed set and examine all its successor states. If a successor state is already in the closed set, we ignore it. Otherwise, we add it to the open set and update the f-value and the predecessor state stored in p if a cheaper path was found.

Heuristic for Prefix-Alignment Computation. Since the $A^\star$ algorithm uses a heuristic function to efficiently traverse the search space, we present a heuristic for prefix-alignment computation based on an existing heuristic [4] used for conventional alignment computation. Both heuristics can be formulated as an Integer Linear Program (ILP). Note that both heuristics can also be defined as a Linear Program (LP) which leads to faster calculation but less accurate heuristic values.

Let $N^S = (P^S, T^S, F^S, M_i^S, M_f^S, \lambda^S)$ be an SPN of a WF-net $N = (P, T, F, [p_i], [p_o], \lambda)$ and a trace σ with corresponding trace net $N^\sigma = (P^\sigma, T^\sigma, F^\sigma, [p_i^\sigma], [p_o^\sigma], \lambda^\sigma)$. Let $c\colon T^S \rightarrow \mathbb{R}_{\geq 0}$ be a cost function that assigns each (prefix-)alignment move, i.e., transition in the SPN, costs. We define a revised heuristic function for prefix-alignment computation as an ILP:

- *Variables*: $X = \{x_t \mid t \in T^S\}$ and $\forall x_t \in X : x_t \in \mathbb{N}_0$
- *Objective function*: $min \sum_{t \in T^S} x_t \cdot c(t)$
- *Constraints*:
  - Trace net part: $M_f^S(p) = \sum_{t \in {}^\bullet p} x_t - \sum_{t \in p^\bullet} x_t \quad \forall p \in P^S : p \in P^\sigma$
  - Process model part: $0 \leq \sum_{t \in {}^\bullet p} x_t - \sum_{t \in p^\bullet} x_t \quad \forall p \in P^S : p \in P$

The revised heuristic presented is a relaxed version of the existing heuristic used for conventional alignment computation. Admissibility and consistency can be proven in a similar way as for the existing heuristic. We refer to [4,9].

Reducing Heuristic Recalculations. In this section, we describe an approach to reduce the number of heuristic calculations. Reconsider line 6 in Algorithm 2. Before we continue the search on an extended search space, we recalculate the heuristic for all states in the open set. This is needed because the goal states differ in each incremental execution. However, these recalculations are computational expensive. Instead of recalculating the heuristic in advance (Algorithm 2, line 6), we mark all states in the open set that have an outdated heuristic value. Whenever we pop a state from the open set with an outdated heuristic value (line 8), we update its h-value, put it back in the open set and skip the remainder of the while body (from line 9). Thereby, we do not have to recalculate the heuristic value for all states in the open set. This approach is permissible because the goal states are "further away" in each iteration and hence, h-values can only grow.

5 Correctness

In this section, we prove the correctness of the approach. We show that states in the closed set do not get new successors upon extending the SPN. Furthermore,

we show that newly added states never connect to "older" states. Finally, we show that the open set always contains a state which is part of an optimal prefix-alignment of the extended trace.

Lemma 1 (State-space growth is limited to frontier). *Let* $\sigma^{i-1} = \langle a_1, \ldots, a_{i-1} \rangle$, $\sigma^i = \sigma^{i-1} \cdot \langle a_i \rangle$, *and* $\sigma^{i+1} = \sigma^i \cdot \langle a_{i+1} \rangle$. *For a WF-net,* N *let* $N^S_{i-1} = (P^S_{i-1}, T^S_{i-1}, F^S_{i-1}, M^S_{i_{i-1}}, M^S_{f_{i-1}}, \lambda^S_{i-1})$ *be the SPN of* N *and* σ^{i-1}, N^S_i *and* N^S_{i+1} *analogously.*

$$\forall M \in \mathcal{B}(P^S_{i-1}) \forall t \in T^S_{i+1} \left((N^S_{i+1}, M)[t\rangle \Rightarrow t \in T^S_i \right)$$

Proof (By construction of the SPN). Observe that $P^S_{i-1} \subset P^S_i \subset P^S_{i+1}$ and $T^S_{i-1} \subset T^S_i \subset T^S_{i+1}$. Let $p_{|\sigma^i|} \in P^S_{i+1}$ be the i-th place of the trace net part (note that $p_{|\sigma^i|} \notin P^S_{i-1}$) and let $t_{i+1} \in T^S_{i+1} \setminus T^S_i$. By construction of the SPN, we know that $p_{|\sigma^i|} \in {\bullet}t_{i+1}$ and $\forall j \in \{1, \ldots, i-1\} : p_{|\sigma^j|} \notin {\bullet}t_{i+1}$. □

Observe that, when searching for an alignment for σ^i, Algorithm 2 returns whenever place p_{σ^i} is marked. Moreover, the corresponding marking remains in O. Hence, each state in C is "older", i.e., already part of P^S_{i-1}. Thus, Lemma 1 proves that states in the closed set C do not get new successors upon extending the SPN.

Lemma 2 (New states do not connect to old states). *Let* $\sigma^i = \langle a_1, \ldots, a_i \rangle$ *and* $\sigma^{i+1} = \sigma^i \cdot \langle a_{i+1} \rangle$. *For a given WF-net* N, *let* $N^S_i = (P^S_i, T^S_i, F^S_i, M^S_{i_i}, M^S_{f_i}, \lambda^S_i)$ *(analogously* N^S_{i+1}*) be the SPN of* N *and* σ^i.

$$\forall M \in \mathcal{B}(P^S_{i+1}) \setminus \mathcal{B}(P^S_i) \forall M' \in \mathcal{B}(P^S_i) \left(\nexists t \in T^S_{i+1} \left((N^S_{i+1}, M)[t\rangle (N^S_{i+1}, M') \right) \right)$$

Proof (By construction of the SPN). Let $t_{i+1} \in T^S_{i+1} \setminus T^S_i$. Let $p_{|\sigma^{i+1}|} \in P^S_{i+1}$ be the $(i+1)$-th place (the last place) of the trace net part. We know that $p_{|\sigma^{i+1}|} \in t_{i+1}{\bullet}$ and $p_{|\sigma^j|} \notin t_{i+1}{\bullet}$ $\forall j \in \{1, \ldots, i\}$. For all other $t \in T^S_i$ we know that $\nexists M \in \mathcal{B}(P^S_{i+1}) \setminus \mathcal{B}(P^S_i)$ such that $(N^S, M)[t\rangle$. □

From Lemma 1 and 2 we know that states in the closed set are not affected by extending the SPN. Hence, it is feasible to continue the search from the open set and to not reconsider states which are in the closed set.

Lemma 3. (Exists a state in the O-set that is on the shortest path). *Let* $\sigma^i = \langle a_1, ..., a_i \rangle$, $\sigma^{i+1} = \sigma^i \cdot \langle a_{i+1} \rangle$, N^S_i, N^S_{i+1} *the corresponding SPN for a WF-net* N, O^i *and* C^i *be the open- and closed set after the prefix-alignment computation for* σ_i. *Let* $\overline{\gamma}_{i+1}$ *be an optimal prefix-alignment for* σ_{i+1}.

$$\exists j \in \{1, \ldots, |\overline{\gamma}_{i+1}|\}, \overline{\gamma}'_{i+1} = (\overline{\gamma}_{i+1}(1), \ldots, \overline{\gamma}_{i+1}(j)) \text{ s.t.}$$

$$(N^S_{i+1}, M^S_{i+1}) \xrightarrow{\pi^*_2(\overline{\gamma}'_{i+1})_{\downarrow_T}} (N^S_{i+1}, M_O) \text{ and } M_O \in O^i$$

Proof. $\overline{\gamma}^{i+1}$ corresponds to a sequence of markings, i.e., $S = (M^S_{i+1}, \ldots, M', M'', ..., M''')$. Let $X^{i+1} = \mathcal{B}(P^S_{i+1}) \setminus C^i \cup O^i$. It holds that $X^{i+1} \cap O^i = X^{i+1} \cap C^i = O^i \cap C^i = \emptyset$. Note that $M''' \in X^{i+1}$ because $M''' \notin \mathcal{B}(P^S_i)$. Assume $\forall M \in S : M \notin O^i \Rightarrow \forall M \in S : M \in C^i \cup X^{i+1}$. Observe that $M^S_i = M^S_{i+1} \in C^i$ since initially $M^S_i \in O^0$and in the very first iteration M^S_i is selected for expansion because it is not a goal state, Algorithm 2. We know that for any state pair M', M'' it cannot be the case that $M' \in C^i, M'' \in X^{i+1}$. Since we know that at least $M^S_i \in C^i$ and $M''' \in X^{i+1}_c$ there $\exists M', M'' \in S$ such that $M' \in C^i, M'' \in O^i$. □

Hence, it is clear from Lemmas 1–3 that incrementally computing prefix-alignments, continuing the search from the previous open- and closed set, leads to optimal prefix-alignments.

6 Evaluation

We evaluated the algorithm on publicly available real event data from various processes. Here, we present the experimental setup and discuss the results.

6.1 Experimental Setup

The algorithm introduced in [22] serves as a comparison algorithm. We refer to it as Online Conformance Checking (OCC). Upon receiving an event, OCC partially reverts the previously computed prefix-alignments (using a given maximal window size) and uses the corresponding resulting state of the SPN as start state. Hence, the algorithm cannot guarantee optimality, i.e., it does not search for a global optimum. However, OCC can also be used without partially reverting, i.e., using window size ∞. Hence, it naively starts the computation from scratch without reusing any information, however, optimality is then guaranteed. We implemented our proposed algorithm, incremental $A^\star$ (IAS), as well as OCC in the process mining library *PM4Py* [5]. The source code is publicly available[1]. Although, the OCC algorithm was introduced without a heuristic function [22], it is important to note that both algorithms, IAS and OCC, use the previously introduced heuristic in the experiments to improve objectivity.

We use publicly available datasets capturing the execution of real-life processes [6,11,17–19]. To mimic an event stream, we iterate over the traces in the event log and emit each preformed activity as an event. For instance, given the event log $L = [\langle a, b, c\rangle, \langle b, c, d\rangle, \ldots]$, we simulate the event stream $\langle (1, a), (1, b), (1, c), (2, b), (2, c), (2, d), \ldots \rangle$. For all datasets except CCC19 [19] that contains a process model, we discovered reference process models with the Inductive Miner infrequent version (IMf) [16] using a high threshold. This results in process models that do not guarantee full replay fitness. Moreover, the discovered process models contain choices, parallelism and loops.

[1] https://github.com/fit-daniel-schuster/online_process_monitoring_using_incremental_state-space_expansion_an_exact_algorithm.

6.2 Results

In Table 2, we present the results. OCC-Wx represents the OCC algorithm with window size x, OCC with an infinite window size. Moreover, we present the results for the IAS algorithm that does *not* use the approach of reducing heuristic recalculations as presented in Sect. 4.3, we call it IASR. Note that only IAS(R) and OCC guarantee optimality. Furthermore, note that a queued state corresponds to a state added to the open set and a visited state corresponds to a state moved into the closed set. Both measures indicate the search efficiency.

We observe that reducing the number of heuristic re-calculations is valuable and approximately halves the number of solved LPs and hence, reduces the computation time. As expected, we find no significant difference in the other measured dimensions by comparing IAS and IASR. We observe that IAS clearly outperforms all OCC variants regarding search efficiency for all used event logs except for CCC19 where OCC variants with small window sizes have a better search efficiency. This results illustrate the relevance of IAS compared to OCC and OCC-Wx and show the effectiveness of continuing the search on an extended search space by reusing previous results. Regarding false positives, we observe that OCC-Wx variants return non-optimal prefix-alignments for all event logs. As expected, the number of false positives decreases with increasing window size. In return, the calculation effort increases with increasing window size. This highlights the advantage of the IAS' property being parameter-free. In general, it is difficult to determine a window size because the traces, which have an impact on the "right" window size, are not known in an online setting upfront.

Table 2. Results of the conducted experiments for various real-life event logs

Event log	≈ avg. queued states per trace							≈ avg. visited states per trace						
	IASR	IAS	OCC	OCC-W1	OCC-W2	OCC-W5	OCC-W10	IASR	IAS	OCC	OCC-W1	OCC-W2	OCC-W5	OCC-W10
CCC 19 [19]	774	766	14614	312	431	885	1622	756	751	12557	212	283	506	932
Receipt [6]	31	29	65	37	50	82	104	18	17	26	19	23	33	42
Sepsis [17]	73	70	532	102	146	285	450	44	43	232	47	62	103	166
Hospital [18]	21	21	42	32	41	65	71	11	11	15	14	17	23	26
BPIC 19 [11]	28	28	257	41	57	90	107	18	18	154	21	27	40	48
Event log	**# traces with false positives**							**# variants with false positives**						
	IASR	IAS	OCC	OCC-W1	OCC-W2	OCC-W5	OCC-W10	IASR	IAS	OCC	OCC-W1	OCC-W2	OCC-W5	OCC-W10
CCC 19 [19]	0	0	0	7	8	1	1	0	0	0	7	8	1	1
Receipt [6]	0	0	0	8	5	3	1	0	0	0	8	5	3	1
Sepsis [17]	0	0	0	59	60	6	1	0	0	0	58	59	6	1
Hospital [18]	0	0	0	88	88	69	32	0	0	0	49	49	39	19
BPIC 19 [11]	0	0	0	318	259	193	90	0	0	0	272	206	145	75
Event log	**≈ avg. computation time (s) per trace**							**≈ avg. number solved LPs (heuristic functions) per trace**						
	IASR	IAS	OCC	OCC-W1	OCC-W2	OCC-W5	OCC-W10	IASR	IAS	OCC	OCC-W1	OCC-W2	OCC-W5	OCC-W10
CCC 19 [19]	12.2	5.69	35.7	0.74	0.85	1.51	2.61	3345	1889	8443	338	393	658	1066
Receipt [6]	0.12	0.04	0.05	0.04	0.04	0.07	0.09	89.2	42	53	40	50	75	91
Sepsis [17]	0.59	0.28	0.6	0.09	0.11	0.23	0.35	518	226	343	104	138	247	356
Hospital [18]	0.05	0.03	0.03	0.02	0.03	0.04	0.05	63	30	35	34	42	61	66
BPIC 19 [11]	0.4	0.19	0.79	0.06	0.09	0.12	0.14	128	71	136	44	57	81	91

Regarding calculation time, we note that the number of solved LPs has a significant influence. We observe that IAS has often comparable computation time to the OCC-wx versions. Comparing optimality guaranteeing algorithms (IAS & OCC), IAS clearly outperforms OCC in all measured dimensions for all logs.

6.3 Threats to Validity

In this section, we outline the limitations of the experimental setup. First, the artificial generation of an event stream by iterating over the traces occurring in the event log is a simplistic approach. However, this allows us to ignore the general challenge of process mining on streaming data, deciding when a case is complete, since new events can occur at any time on an (infinite) event stream. Hence, we do not consider the impact of multiple cases running in parallel.

The majority of used reference process models are discovered with the IMf algorithm. It should, however, be noted that these discovered models do not contain duplicate labels. Finally, we compared the proposed approach against a single reference, the OCC approach. To the best of our knowledge, however, there are no other algorithms that compute prefix-alignments on event streams.

7 Conclusion

In this paper, we proposed a novel, parameter-free algorithm to efficiently monitor ongoing processes in an online setting by computing a prefix-alignment once a new event occurs. We have shown that the calculation of prefix-alignments on an event stream can be "continued" from previous results on an extended search space with different goal states, while guaranteeing optimality. The proposed approach is designed for prefix-alignment computation since it utilizes specific properties of the search space regarding prefix-alignment computation and therefore, generally not transferable to other shortest path problems. The results show that the proposed algorithm outperforms existing approaches in many dimensions and additionally ensures optimality.

In future work, we plan to implement the proposed approach in real application scenarios and to conduct a case study. Thereby, we want to focus on limited storage capacities, which requires to decide whether a case is considered to be completed to free storage.

References

1. van der Aalst, W.M.P.: The application of Petri Nets to workflow management. J. Circuits Syst. Comput. **8**(1), 21–66 (1998). https://doi.org/10.1142/S0218126698000043
2. van der Aalst, W.M.P., Adriansyah, A., van Dongen, B.F.: Replaying history on process models for conformance checking and performance analysis. Wiley Interdisc. Rew.: Data Min. Knowl. Discov. **2**(2), 182–192 (2012)

3. van der Aalst, W.M.P.: Process Mining - Data Science in Action, 2nd edn. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49851-4
4. Adriansyah, A.: Aligning observed and modeled behavior. Ph.D. thesis, Eindhoven University of Technology, Department of Mathematics and Computer Science (2014). https://doi.org/10.6100/IR770080
5. Berti, A., van Zelst, S.J., van der Aalst, W.: Process mining for Python (PM4Py): Bridging the gap between process-and data science. In: Proceedings of the ICPM Demo Track 2019, Co-located with 1st International Conference on Process Mining (ICPM 2019), Aachen, Germany, 24–26 June 2019. pp. 13–16 (2019). http://ceur-ws.org/Vol-2374/
6. Buijs, J.: Receipt phase of an environmental permit application process ('WABO'), CoSeLoG project. Dataset (2014). https://doi.org/10.4121/uuid:a07386a5-7be3-4367-9535-70bc9e77dbe6
7. Burattin, A.: Online conformance checking for Petri Nets and event streams. In: Proceedings of the BPM Demo Track and BPM Dissertation Award co-located with 15th International Conference on Business Process Modeling (BPM 2017), Barcelona, Spain, 13 September 2017 (2017). http://ceur-ws.org/Vol-1920/BPM_2017_paper_153.pdf
8. Burattin, A., van Zelst, S.J., Armas-Cervantes, A., van Dongen, B.F., Carmona, J.: Online conformance checking using behavioural patterns. In: Weske, M., Montali, M., Weber, I., vom Brocke, J. (eds.) BPM 2018. LNCS, vol. 11080, pp. 250–267. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-98648-7_15
9. Carmona, J., van Dongen, B.F., Solti, A., Weidlich, M.: Conformance Checking - Relating Processes and Models. Springer, Heidelberg (2018). https://doi.org/10.1007/978-3-319-99414-7
10. Dijkman, R.M., Dumas, M., Ouyang, C.: Semantics and analysis of business process models in BPMN. Inf. Softw. Technol. **50**(12), 1281–1294 (2008). https://doi.org/10.1016/j.infsof.2008.02.006
11. van Dongen, B.F.: BPI Challenge 2019. Dataset (2019). https://doi.org/10.4121/uuid:d06aff4b-79f0-45e6-8ec8-e19730c248f1
12. Hart, P.E., Nilsson, N.J., Raphael, B.: A formal basis for the heuristic determination of minimum cost paths. IEEE Trans. Syst. Sci. Cybern. **4**(2), 100–107 (1968). https://doi.org/10.1109/TSSC.1968.300136
13. Koenig, S., Likhachev, M.: Incremental A*. In: Dietterich, T.G., Becker, S., Ghahramani, Z. (eds.) Advances in Neural Information Processing Systems 14 (Neural Information Processing Systems: Natural and Synthetic, NIPS 2001, 3–8 December 2001, Vancouver, British Columbia, Canada), pp. 1539–1546. MIT Press (2001). http://papers.nips.cc/paper/2003-incremental-a
14. Koenig, S., Likhachev, M.: Adaptive A. In: Dignum, F., Dignum, V., Koenig, S., Kraus, S., Singh, M.P., Wooldridge, M.J. (eds.) 4th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2005), 25–29 July 2005, Utrecht, The Netherlands, pp. 1311–1312. ACM (2005). https://doi.org/10.1145/1082473.1082748
15. Koenig, S., Likhachev, M., Liu, Y., Furcy, D.: Incremental heuristic search in AI. AI Mag. **25**(2), 99–112 (2004). http://www.aaai.org/ojs/index.php/aimagazine/article/view/1763
16. Leemans, S.J.J., Fahland, D., van der Aalst, W.M.P.: Discovering block-structured process models from event logs containing infrequent behaviour. In: Lohmann, N., Song, M., Wohed, P. (eds.) BPM 2013. LNBIP, vol. 171, pp. 66–78. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-06257-0_6

17. Mannhardt, F.: Sepsis Cases. Dataset. 4TU.Centre for Research Data (2016). https://doi.org/10.4121/uuid:915d2bfb-7e84-49ad-a286-dc35f063a460
18. Mannhardt, F.: Hospital Billing. Dataset. 4TU.Centre for Research Data (2017). https://doi.org/10.4121/uuid:76c46b83-c930-4798-a1c9-4be94dfeb741
19. Munoz-Gama, J., de la Fuente, R., Sepúlveda, M., Fuentes, R.: Conformance checking challenge 2019. dataset. 4TU.Centre for Research Data (2019). https://doi.org/10.4121/uuid:c923af09-ce93-44c3-ace0-c5508cf103ad
20. Murata, T.: Petri Nets: properties, analysis and applications. Proc. IEEE **77**(4), 541–580 (1989). https://inst.eecs.berkeley.edu/~ee249/fa07/discussions/PetriNets-Murata.pdf
21. Rozinat, A., van der Aalst, W.M.P.: Conformance checking of processes based on monitoring real behavior. Inf. Syst. **33**(1), 64–95 (2008). https://doi.org/10.1016/j.is.2007.07.001
22. van Zelst, S.J., Bolt, A., Hassani, M., van Dongen, B.F., van der Aalst, W.M.P.: Online Conformance checking: relating event streams to process models using prefix-alignments. Int. J. Data Sci. Anal. (2017). https://doi.org/10.1007/s41060-017-0078-6