Int J Data Sci Anal
DOI 10.1007/s41060-017-0078-6

REGULAR PAPER

Online conformance checking: relating event streams to process models using prefix-alignments

Sebastiaan J. van Zelst[1] · **Alfredo Bolt**[1] · **Marwan Hassani**[1] · **Boudewijn F. van Dongen**[1] · **Wil M. P. van der Aalst**[1]

Received: 26 May 2017 / Accepted: 11 October 2017

Abstract Companies often specify the intended behaviour of their business processes in a process model. Conformance checking techniques allow us to assess to what degree such process models and corresponding process execution data correspond to one another. In recent years, alignments have proven extremely useful for calculating conformance checking statistics. Existing techniques to compute alignments have been developed to be used in an offline, a posteriori setting. However, we are often interested in observing deviations at the moment they occur, rather than days, weeks or even months later. Hence, we need techniques that enable us to perform conformance checking in an online setting. In this paper, we present a novel approach to incrementally compute prefix-alignments, paving the way for real-time online conformance checking. Our experiments show that the reuse of previously computed prefix-alignments enhances memory efficiency, whilst preserving prefix-alignment optimality. Moreover, we show that, in case of computing approximate prefix-alignments, there is a clear trade-off between memory efficiency and approximation error.

✉ Sebastiaan J. van Zelst
s.j.v.zelst@tue.nl

Alfredo Bolt
a.bolt@tue.nl

Marwan Hassani
m.hassani@tue.nl

Boudewijn F. van Dongen
b.f.v.dongen@tue.nl

Wil M. P. van der Aalst
w.m.p.v.d.aalst@tue.nl

[1] Department of Mathematics and Computer Science, Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands

Keywords Process mining · Conformance checking · Event streams

1 Introduction

Today's information systems track, in great detail, the execution of business processes within companies. Often, a company has an idea, or even a formal specification, of how their business process is required to be executed. In other cases, laws, regulations and/or legislations dictate the exact way the process is to be executed. Such process specification is often recorded in a process model, i.e. a behavioural specification. However, in many cases, the actual execution of the process, as recorded by the information system, is not in line with the behaviour described by the corresponding process model. *Conformance checking*, a sub-field of *process mining* [1], aims at assessing to what degree the behaviour described by a process model is in line with behaviour captured in an event log. In particular, the techniques are able to check conformance based on process modelling formalisms that allow for describing *concurrency*, i.e. the possibility to specify order-independent execution of activities.

Early conformance checking techniques, e.g. "token-based replay" [2], often lead to ambiguous and/or unpredictable results. Hence, *alignments* [3] were developed with the specific goal in mind to explain and quantify deviations in a non-ambiguous manner. Alignments have rapidly developed into the de facto standard conformance checking technique. Moreover, alignments serve as a basis for techniques that link event data to process models, e.g. they support performance analysis, decision mining [4], business process model repair [5] and prediction techniques.

Published online: 27 October 2017

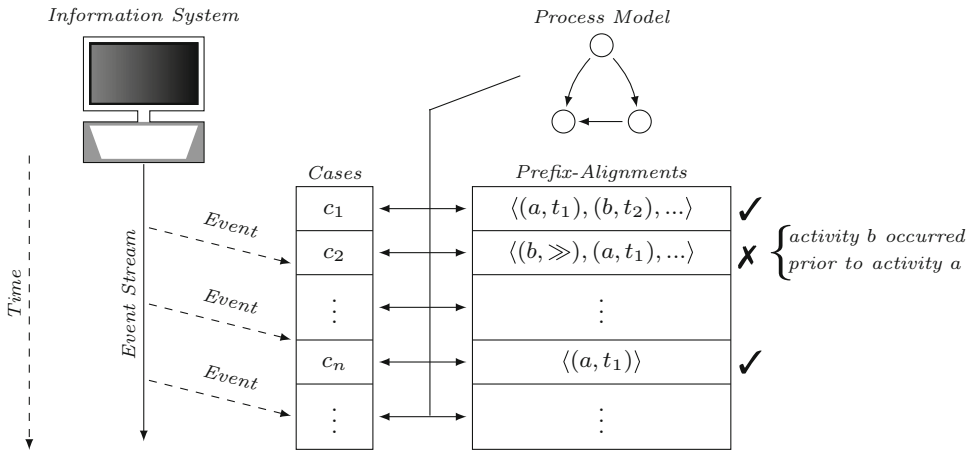


Fig. 1 Schematic overview of online conformance checking

Techniques to compute alignments are only defined in an offline setting. However, early *process-oriented deviation detection* is critical for many organizations in different domains. For example, within hospitals, deviating process executions often lead to higher costs, and/or delays in examination time. Similarly, within highly complex administrative processes, e.g. provision of mortgages, notary processes and unemployment administration, deviant behaviour often leads to excessive process execution time and costs. Upon detection of a deviation, a process owner, or the supporting information system, is able to take adequate actions such as blocking the current process instance, assigning a case manager for additional specialized intervention and restarting the process instance.

In this paper, we present a new approach to compute alignment-based conformance checking statistics in an online setting. Instead of conventionally used *event logs*, i.e. a static data source describing past process execution behaviour, we rely on *event streams*. An event stream is a continuous data stream that describes a potentially unbounded sequence of events. The fundamental difference of event streams w.r.t. event logs is related to the fact that the knowledge of executed events for a case changes over time, i.e. new events related to the same case can occur in the future. Hence, at any point in time, we are unaware whether the sequence of events observed for a certain case is complete or not. As a consequence, we aim at computing *prefix-alignments* rather than *conventional alignments* as they describe the events observed for a case in the best possible way w.r.t. the reference model without requiring explicit termination. In Fig. 1, we present a schematic overview of online conformance checking. We have two main sources of input, i.e. an event stream generated by an information system and a reference process model. Over time, we observe events emitted on the event stream which tell us what activity has been performed in context of what case. For each case we maintain a prefix-alignment. Whenever we receive a new event for a case, we recompute its prefix-alignment. We try to recompute prefix-alignments greedily; however, in some cases we need to resort to solving a shortest path problem. The focus of this paper is mainly towards the efficiency of solving such shortest problem.

Our proposed approach entails an incremental algorithm that allows for computing both *optimal* and *approximate* prefix-alignments. We additionally show that the cost of an optimal prefix-alignment is always an underestimate for the cost of a conventional alignment of any of its possible suffixes. As a consequence, when computing optimal prefix-alignments, our approach underestimates alignment costs for completed cases. This implies that once we detect a deviation from the reference model, we are guaranteed that the behaviour related to the case is not compliant with the reference model. Computing approximate prefix-alignments leads to an increase in memory efficiency, however, at the cost of losing prefix-alignment optimality. We experimentally assess the trade-off between memory efficiency and optimality loss using several artificially generated process models. We additionally assess the applicability of our technique using a real data set originating from a hospital information system. Our experiments show that reusing previously computed prefix-alignments positively impacts the efficiency of computing new prefix-alignments. Moreover, in case of approximation, we observe a clear trade-off between memory usage and prefix alignment optimality loss.

The remainder of this paper is structured as follows: In Sect. 2, we present related work. In Sect. 3, we present background concepts. In Sect. 4, we introduce event streams and motivate the need for computing prefix-alignments. In Sect. 5, we present our incremental algorithm for prefix-alignment computation together with two memory optimization techniques. In Sect. 6, we evaluate the proposed approach in terms of performance and approximation accuracy. In Sect. 7, we provide a discussion of the proposed approach. Section 8 concludes the paper.

2 Related work

A plethora of different process mining techniques exists, ranging from discovery to prediction. However, given the focus of this paper, we limit related work to the field of alignment computation and online process mining. Hence, we refer to [1] for an overview of different process mining techniques.

Early work in conformance checking uses token-based replay [2]. The techniques replay a trace of executed events in a process model (Petri net) and add missing tokens if transitions are not able to fire. After replay, remaining tokens are counted and a conformance statistic is computed based on missing and remaining tokens. Alignments were introduced in [3] and have rapidly developed into the de facto standard for conformance checking. In [6,7], decomposition techniques are proposed together with computing alignments. Using decomposition techniques greatly enhances computation time, i.e. the techniques successfully apply the divide-and-conquer paradigm; however, the techniques provide lower bounds on conformance checking statistics, rather than computing alignments. More recently, general approximation schemes for alignments, i.e. computation of near-optimal alignments, have been proposed in [8].

A relatively limited amount of work has been done in the area of online process mining. In [9], a first design of an online process discovery algorithm was proposed, based on the Heuristic Miner [10]. The approach was improved in [11] by adopting a different internal storage model. In [12], the work was generalized and converted in an architecture that covers a wide range of process discovery algorithms. In [13], an alternative discovery approach is presented for the purpose of discovering declarative process models.

To the best of our knowledge this paper is the first work that covers conformance checking/alignments in online/incremental settings.

3 Background

In this section, we briefly present key process mining concepts such as *event logs*, *workflow nets* and *alignments*. We assume the reader to be familiar with mathematical concepts such as sets, multisets, functions and sequences. We only present notational conventions regarding these concepts, as used in this paper.

Given set X and $y \notin X$ we write X^y for $X \cup \{y\}$. $\mathbb{N}$ denotes the set of natural numbers, $\mathbb{N}_0$ includes 0. A multiset generalizes the concept of a set and allows elements to have a multiplicity exceeding one. Let $X = \{e_1, e_2, \ldots, e_n\}$ be a set, a multiset M over X is a function $M\colon X \to \mathbb{N}_0$. We write a multiset as $M = [e_1^{k_1}, e_2^{k_2}, \ldots, e_m^{k_m}]$ $(m \leq n)$, where for each $i \in \{1, \ldots, m\}$ we have $M(e_i) = k_i$. If $M(e_i) = 0$, we omit e_i from multiset notation, and, if $M(e_i) = 1$ we omit e_i's superscript. We write sequence σ of length n as $\sigma = \langle \sigma(1), \sigma(2), \ldots, \sigma(n) \rangle$, where for $1 \leq i \leq n$, $\sigma(i)$ denotes the ith element of σ. The set of all possible sequences over X is written as X^*. Concatenation of sequences σ_1 and σ_2 is written as $\sigma_1 \cdot \sigma_2$. An empty sequence is written as ϵ. Given a sequence $\sigma \in (X_1 \times X_2 \times \cdots \times X_n)^*$ we define, for $1 \leq i \leq n$, $\pi_i(\sigma) \in X_i^*$ with $\pi_i(\sigma)(j) = \sigma(j)(i), \forall j \in \{1, 2, \ldots, |\sigma|\}$, e.g. given $\sigma \in (X \times Y)^*$, we have $\pi_1(\sigma) \in X^*$ and $\pi_2(\sigma) \in Y^*$. Given $\sigma \in X^*$ and $Y \subseteq X$ we define $\sigma_{\downarrow Y} \in Y^*$ recursively with $\epsilon_{\downarrow Y} = \epsilon$ and $(\langle x \rangle \cdot \sigma')_{\downarrow Y} = \langle x \rangle \cdot \sigma'_{\downarrow Y}$ if $x \in Y$ and $\sigma'_{\downarrow Y}$ if $x \notin Y$.

3.1 Event logs and process models

The execution of business processes within a company generates traces of event data in its supporting information system. Typically we are able to extract such data from the company's information system describing, for specific cases, e.g. an insurance claim, what activities have been performed over time. We often refer to a collection of such data as an *event log*. From a formal perspective, an event log is considered to be a multiset of sequences of executed process activities, or simply *events*. Consider Table 1, which depicts a simplified view of an event log. The event log describes the execution of activities related to a fictional compensation request process for concert tickets. For example, consider all events related to case 13 (represented by *Case-id* 13), i.e. a new *request is registered* by *Luke*, *Harry* subsequently *examines* the request, *Pete checks* the corresponding *ticket*, etc.

For each event recorded in the event log, we have information regarding its *id*, what *activity* it related to, which *resource* executed the activity and at what *time* it was executed. In general, we are able to obtain even more event data, for example what is the corresponding *ticket id*, what *concert* the ticket belongs to, what is the *ticket price*, etc. However, for the sake of simplicity we abstract from such data. In particular, in context of this paper, we are only interested in actual activities performed, instead of all possible data/resource aspects involved in the activity execution, i.e. we focus on the *control-flow perspective*. For example, based on Table 1 we deduce that for case 13, the sequence $\langle$*register request*, *examine*, *check ticket*, *decide*, *pay compensation*$\rangle$ was performed. We assume the execution of activities to be atomic; hence, given the universe of activities $\mathcal{A}$, an event log L is a multiset over sequences of $\mathcal{A}$, i.e. $L\colon \mathcal{A}^* \to \mathbb{N}_0$.

A process model describes the intended behaviour of a process. Although many process modelling formalisms exist, we focus on (labelled) Petri nets [14], which allow us to explicitly model concurrency in a concise and compact manner. In Fig. 2, we depict an example Petri net. The Petri net, like the example event log in Table 1, describes handling of a compensation request. It dictates that the first activity to be

Table 1 Example event log fragment

Event-id	Case-id	Activity	Resource	Time-stamp
...	...	...	...	...
5	*12*	*Decide (d)*	*Boris*	*2017-05-08 09:45*
6	*13*	*Register request (a)*	*Luke*	*2017-05-08 10:12*
7	*12*	*Update records (h)*	*Harry*	*2017-05-08 10:14*
8	*13*	*Examine (b)*	*Harry*	*2017-05-08 10:31*
9	*13*	*Check ticket (c)*	*Pete*	*2017-05-08 10:40*
10	*13*	*Decide (d)*	*Harry*	*2017-05-08 10:49*
11	*13*	*Pay compensation (e)*	*Harry*	*2017-05-08 11:01*
12	*14*	*Register request (a)*	*Boris*	*2017-05-08 11:03*
...	...	...	...	...

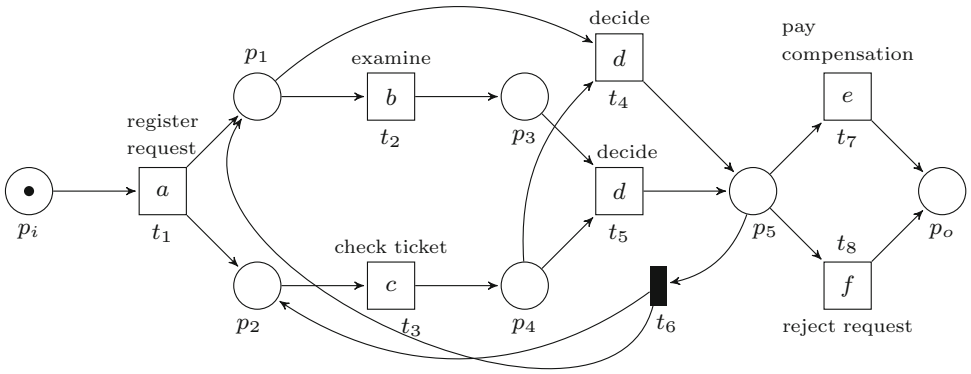


Fig. 2 Example Petri net N_1 (adopted from [1]) with initial marking $[p_i]$

performed should always be *register request*. Subsequently, the *examine* and *check ticket* activities can be performed concurrently. However, we are also allowed to only perform the check ticket activity and subsequently make a decision, i.e. to skip the examination. After a decision is made, we are able to redo the examination and ticket check. However, such decision, i.e. to redo these activities, is not explicitly captured by the system. Eventually, either the *pay compensation* or the *reject request* activity is performed.

A Petri net consists of *places* and *transitions*. Places are used to represent the *state* of the described process, whereas transitions represent possible executable activities, subject to the state. The Petri net in Fig. 2 consists of 7 places (denoted P), i.e. $P = \{p_i, p_1, \ldots, p_5, p_o\}$, visualized as circles. Formally, we represent the state of a Petri net in terms of a *marking* M, which is a multiset of places, i.e. $M : P \to \mathbb{N}_0$. For example, in Fig. 2 place p_i is *marked* with a *token*, visualized by a black dot. Thus, the marking of the Petri net in Fig. 2, as visualized, is $[p_i]$. The Petri net furthermore contains 8 transitions (denoted T), i.e. $\{t_1, \ldots, t_8\}$, visualized as boxes. Transitions allow us to manipulate a Petri net's marking. A transition $t \in T$ is *enabled* if all places p that have an outgoing arc to t contain a token. If a transition is enabled in marking M, we write $M[t\rangle$. An enabled transition is able to *fire*. If we fire a transition t, it consumes a token from each place that has an outgoing arc to t. Subsequently, a token is produced in each place that has an incoming arc from t. For example, in Fig. 2, t_1 is the only enabled transition in marking $[p_0]$, and, if it fires we obtain new marking $[p_1, p_2]$. In marking $[p_1, p_2]$, we are able to fire both t_2 and t_3, in any order. We are thus able to generate sequences of fired transitions, e.g. $\langle t_1, t_2, t_3\rangle$ and $\langle t_1, t_3, t_2\rangle$, which both yield marking $[p_3, p_4]$. Note that, in marking $[p_1, p_2]$, transition t_4 is not (yet) enabled. If we fire t_3, leading to marking $[p_1, p_4]$, transition t_4 is enabled. When executing a sequence $\sigma \in T^*$ of transitions from marking M results in M', we write $M \xrightarrow{\sigma} M'$. We let $\mathcal{M}$ denote the universe of all possible markings.

All transitions, except transition t_6, have a single character *label*, e.g. transition t_1 has label a. Typically, these labels represent actual activities that can be executed in the process described by the Petri net. For convenience, we also added more descriptive names, e.g. *register request*. Transition t_6 is an *invisible transition*, i.e. it is able to manipulate the marking of the Petri net, without being noticed by the outside world. Also, there are two transitions with label d, i.e. t_4 and t_5.

We formally define a Petri net N as a tuple $N = (P, T, F, \lambda)$, where P represents its places, T its transitions, $F \subseteq (P \times T) \cup (T \times P)$ represents the flow relation, i.e. the arcs in Fig. 2. Observe that a place is only connected to a transition and vice versa, i.e. there is never an arc between two places/transitions. Finally, given a set of activity labels

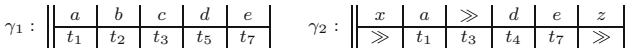


Fig. 3 Example alignments for $\langle a, b, c, d, e\rangle$ and $\langle x, a, d, e, z\rangle$ with N_1

Λ and $\tau \notin \Lambda$, $\lambda\colon T \to \Lambda^{\tau}$ represents the labelling function of N. For example, $\lambda(t_1) = a$ and $\lambda(t_6) = \tau$.

We assume that a reference model of a process is designed by a human business process analyst/designer. We therefore assume that a process model has a certain level of quality, e.g. the Petri net is a *sound workflow net* [15, Definition 7]. We do not introduce the characteristics of such models, however, soundness guarantees that we are always able to reach a designated final marking M_f, from any marking M reachable from designated initial marking M_i. For the purpose of this paper, we assume that the Petri nets we consider also consist of this property, which we deem the *proper termination assumption*.

3.2 Alignments

When we reconsider the example sequence of activities related to case 13, i.e. written short-hand as $\langle a, b, c, d, e\rangle$, we observe that indeed by firing transitions t_1, t_2, t_3, t_5 and t_7, such sequence of activities is produced by N_1 in Fig. 2. If we consider another example trace, i.e. $\langle x, a, d, e, z\rangle$, we observe some problems. For example, activities x and z are not labels of N_1. Furthermore, according to N_1, at least c must be executed in-between a and d.

Alignments allow us to identify and quantify the aforementioned problems and moreover allow us to express these deviations in terms of the reference model. Conceptually, an alignment is a mapping between the execution of transitions in the process model and the activities observed in a *trace* σ in a given event log L. Consider Fig. 3, in which we present alignments for traces $\langle a, b, c, d, e\rangle$ and $\langle x, a, d, e, z\rangle$ w.r.t. Petri net N_1. Alignments are sequences of pairs, e.g. $\gamma_1 = \langle (a, t_1), (b, t_2), \ldots, (e, t_7)\rangle$. Each pair within an alignment is referred to as a *move*. The first element of a move refers to an activity of the trace, whereas the second element refers to a transition. The goal is to create pairs of the form (a, t) s.t. $\lambda(t) = a$, e.g. all moves in γ_1 are of this form. The sequence of activity labels in the alignment needs to equal the input trace. The sequence of transitions in the alignment needs to correspond to a $\sigma \in T^*$ s.t., given a designated initial marking M_i and final marking M_f, we have $M_i \xrightarrow{\sigma} M_f$. For N_1, we have $M_i = [p_i]$ and $M_f = [p_o]$. In some cases, we are not able to construct a move of the form (a, t) s.t. $\lambda(t) = a$. In case of trace $\langle x, a, d, e, z\rangle$, we are not able to map x and z to any transition in N_1 with an equally valued label. Furthermore, we at least need to execute transition t_3 in order to form a sequence of transitions that generates marking M_f from M_i. In some cases we need to fire a transition t with $\lambda(t) = \tau$, for which we again are not able to construct a label-transition mapping. In such cases, we use *skip-symbol* $\gg$ in either the activity or the transition part of a move. For example, consider γ_2 in Fig. 3, which contains three skip symbols. Verify that again, when ignoring skip symbols, the sequence of activity labels equals the input trace, and the sequence of transitions is valid for M_i and M_f. If a move is of the form (a, t), we call this a *synchronous move*, $(a, \gg)$ is an *activity move* and $(\gg, a)$ is a *model move*.

Definition 1 (*Alignment*) Let $\sigma \in \mathcal{A}^*$. Let $N = (P, T, F, \lambda)$ be a Petri net and let M_i, M_f denote $N's$ initial and final marking. Let $\gg \notin \mathcal{A} \cup T \cup \Lambda$ with $\gg \neq \tau$. A sequence $\gamma \in (\mathcal{A}^{\gg} \times T^{\gg})^*$ is an alignment iff:

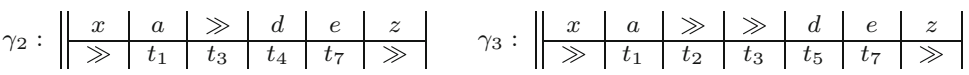
1. $(\pi_1(\gamma))_{\downarrow_{\mathcal{A}}} = \sigma$; activity part (excluding $\gg$'s) equals σ.
2. $M_i \xrightarrow{(\pi_2(\gamma))_{\downarrow_T}} M_f$; transition part (excluding $\gg$'s) in Petri net language.
3. $\forall_{(a,t)\in\gamma}(a \neq \gg \vee t \neq \gg)$; $(\gg, \gg)$ is not valid in an alignment.

We let Γ denote the universe of alignments, and let $\Gamma(N, \sigma, M_i, M_f)$ denote all alignments of N and σ given M_i and M_f.

Given the definition of alignments as presented in Definition 1, several alignments, i.e. sequences of moves adhering to Definition 1, exist for a given trace and Petri net. For example, consider alignment γ_3 depicted in Fig. 4 which, according to Definition 1, is an alignment of $\langle x, a, d, e, z\rangle$ and N_1 as well. The main difference between γ_2 and γ_3, i.e. both aligning trace $\langle x, a, d, e, z\rangle$ with N_1, is the fact that γ_2 binds the execution of t_4 to the observed activity d, whereas γ_3 binds the execution of t_5 to the observed activity d. Clearly, both explanations are possible, however, to be able to bind executed activity d to t_5, and alignment γ_3 requires the explicit execution of transition t_2 as well. Since activity b is not observed in the given trace, we observe the presence of move $(\gg, t_2)$ in γ_3, which is not needed in γ_2. Both alignments are feasible; however, we prefer alignment γ_2 over γ_3 as it minimizes non-synchronous moves, i.e. moves of the form $(a, \gg)$ or $(\gg, t)$.

As exemplified by alignments γ_2 and γ_3, we need means to be able to rank and compare alignments and somehow express our preference of certain alignments w.r.t. others. To this end, we define a cost function over the moves of

Fig. 4 Two possible alignments for $\langle x, a, d, e, z\rangle$ and N_1



an alignment. The cost of an alignment is simply the sum of the costs of its individual moves. Typically synchronous moves are assigned a low, or even 0, cost. The costs of model and activity moves are usually higher than the costs of synchronous moves. Assume we assign cost 0 to synchronous moves and cost 1 to activity/model moves. In this case, the cost of γ_1 is 0. The cost of alignment γ_2 is 3, whereas the cost of alignment γ_3 is 4. Hence, the cost of γ_2 is lower than the cost of γ_3, and we prefer it over γ_2. Formally, we define the cost of a move as a function $\delta\colon \mathcal{A}^{\gg} \times T^{\gg} \to \mathbb{R}_0$. The costs κ^{δ} of a sequence of moves γ, given move cost function δ, are defined as $\kappa^{\delta}(\gamma) = \sum_{i=1}^{|\gamma|} \delta(\gamma(i))$. In general, we are able to use an arbitrary instantiation of δ; however, in the remainder of the paper, we adopt the *unit-cost function*:

1. $\delta(a,t) = 0 \Leftrightarrow a \in \mathcal{A}, t \in T$ and $\lambda(t) = a$ or $a = \gg, t \in T$ and $\lambda(t) = \tau$,
2. $\delta(a,t) = \infty \Leftrightarrow a \in \mathcal{A}, t \in T$ and $\lambda(t) \neq a$,
3. $\delta(a,t) = 1$ otherwise.

Since we assume unit-costs throughout the paper, we omit δ as superscript and simply refer to $\kappa(\gamma)$. We write γ^{opt} to refer to *an optimal alignment*, i.e. $\gamma^{opt} = \textbf{arg min}_{\gamma \in \Gamma(N,\sigma,M_i,M_f)} \kappa(\gamma)$. Consequently, computing an optimal alignment is simply defined as a minimization problem. In [3], it is shown that computing an optimal alignment is equivalent to solving a *shortest path problem on the state space of the synchronous product net of N and σ*. The exact nature of such synchronous product net and an equivalence proof of the two problems is outside the scope of this paper. Hence, we refer to [3] for these definitions and proofs. In this paper, we use the fact that an algorithm to find an optimal alignment exists and we use it as a black box. Finally, it is important to note that multiple optimal alignments exist.

4 Computing prefix-alignments on event streams

We aim at computing conformance checking statistics in an online fashion in order to observe deviations at the moment they occur. Hence, we define the notion of event streams. Subsequently, we motivate why computing conventional alignments on event streams overestimate the potential deviation severity, and therefore, we resort to computing *prefix-alignments*.

4.1 Event streams

Formally, an event stream is a, possibly infinite, sequence of events. An event is a pair consisting of a *case-identifier* and an *activity*. An event describes what activity is performed in context of what process instance (represented by the case-identifier). Event streams differ from event logs in two ways: (1) an event stream is potentially *infinite* and (2) behaviour seen for a case is *incomplete*, i.e. in future new events may be executed in context of a case.

Definition 2 (*Event stream*) Let $\mathcal{C}$ denote the universe of case identifiers. Let $\mathcal{A}$ denote the universe of possible activities. An event stream S is an infinite sequence over $\mathcal{C} \times \mathcal{A}$, i.e. $S \in (\mathcal{C} \times \mathcal{A})^*$.

A pair $(c,a) \in \mathcal{C} \times \mathcal{A}$ represents an event, i.e. activity a was executed in context of case c. $S(1)$ denotes the first event that we receive, whereas $S(i)$ denotes the ith event. Consider stream S_1 in Fig. 5 as an example where we show the emission of activities based on the process model in Fig. 2 using shorthand activity names. Observe that event $(3,d)$ is emitted first ($S_1(1) = (3,d)$), event $(4,a)$ is emitted second, etc. Our knowledge after receiving the third event, i.e. $S_1(3) = (5,a)$, w.r.t. case 5, is different from our knowledge after receiving the fifth event. After the third event, for case 5, we observed $\langle a \rangle$, whereas after the fifth event this is $\langle a,b,c \rangle$.

We aim at computing alignments for the cases emitted onto an event stream, as they allow us to quantify deviations in a clear manner. Assume that we have only seen the first three events, i.e. $(3,d)$, $(4,a)$ and $(5,a)$, of S_1. The only activity seen for case 5 is activity a. An optimal alignment for case 5 is $\langle (a,t_1), (\gg,t_3), (\gg,t_4), (\gg,t_7) \rangle$. After receiving the fourth event, i.e. $(5,b)$, an optimal alignment for case 5 is $\langle (a,t_1), (b,t_2), (\gg,t_3), (\gg,t_5), (\gg,t_7) \rangle$. In both cases, the costs of the alignments are 3. However, after receiving the first nine events on stream S_1 we obtain activity sequence $\langle a,b,c,d,e \rangle$ for case 5 with corresponding optimal alignment $\langle (a,t_1), (b,t_2), (c,t_3), (d,t_5), (e,t_7) \rangle$ with costs 0. Thus, since the knowledge we possess about cases changes over time, computing conventional alignments prior to case completion is expected to lead to an overestimation of the true alignment costs. We do not assume explicit knowledge of case termination w.r.t. the events observed on the stream. Moreover, we aim at detecting potential behaviour during case execution as opposed to computing such figures after case completion. As indicated, doing so with conventional alignments is expected to lead to cost overestimation, and as a consequence, *false positives* from a deviation perspective.

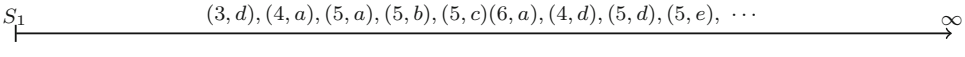


Fig. 5 Example event stream S_1

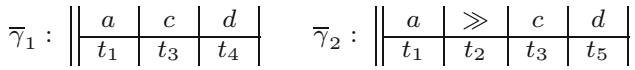


Fig. 6 Two prefix-alignments for $\langle a, c, d\rangle$ and N_1

Therefore, we aim at computing *prefix-alignments*, which are specifically designed to incorporate trace incompleteness.

4.2 Prefix-alignments

Prefix-alignments are a relaxed alternative to conventional alignments [3, Sect. 4.5]. In essence, they relax requirement two of Definition 1 in such way that after executing the $T^{\gg}$-part of the alignment, projected onto T, the final marking M_f *can still be reached*. Formally, we rephrase requirement two to $\exists_{\sigma' \in T^*} \left(M_i \xrightarrow{(\pi_2(\gamma))_{\downarrow T} \cdot \sigma'} M_f \right)$. Consider Fig. 6 in which we depict two example prefix-alignments of incomplete trace $\langle a, c, d\rangle$ and N_1. Observe that, for both alignments we need to append either $\langle t_7\rangle$ or $\langle t_8\rangle$ to obtain marking M_f, and thus, the relaxed requirement is satisfied. Similar to conventional alignments, several prefix-alignments exist that correctly align a prefix and a Petri net. Hence, we again need means to rank and compare prefix-alignments. For example, in Fig. 6 we prefer $\overline{\gamma}_1$ over $\overline{\gamma}_2$, since it only contains synchronous moves, whereas $\overline{\gamma}_2$ contains a model move. Again, we define a cost function for prefix-alignments. Since a prefix-alignment, like a conventional alignment, is a sequences of moves, the cost of a prefix alignment is defined in the exact same manner to the costs of conventional alignments, i.e. it is simply the sum of the costs of its individual moves. Observe that cost function κ is defined over a sequence of moves, and thus, given some prefix-alignment $\overline{\gamma}$, $\kappa(\overline{\gamma})$ is readily defined.

As a consequence, we again have the notion of *optimality*. For example, $\overline{\gamma}_1$ is an optimal prefix-alignment for $\langle a, c, d\rangle$ and N_1. We let $\overline{\Gamma}$ denote the universe of possible prefix-alignments and let $\overline{\Gamma}(N, \sigma, M_i, M_f)$ denote all possible prefix-alignments of σ and N given M_i and M_f.

Interestingly, any optimal prefix-alignment of any prefix of a trace is always underestimating the costs of the optimal alignment of any of its possible suffixes, and thus, of the eventual completed trace.

Proposition 1 (Prefix-alignments underestimate alignment costs) *Let $\sigma \in \mathcal{A}^*$. Let $N = (P, T, F, \lambda)$ be a Petri net with corresponding initial- and final marking M_i, M_f. Let $\gamma \in \Gamma(N, \sigma, M_i, M_f)$ be optimal. If $\overline{\sigma}$ is a prefix of σ and $\overline{\gamma} \in \overline{\Gamma}(N, \overline{\sigma}, M_i, M_f)$ is an optimal prefix-alignment, then $\kappa(\overline{\gamma}) \leq \kappa(\gamma)$.*

Proof (Contradiction) Let us write γ as $\gamma = \gamma' \cdot \gamma''$, s.t. $(\pi_1(\gamma'))_{\downarrow \mathcal{A}} = \overline{\sigma}$. By definition, γ' is a prefix-alignment of $\overline{\sigma}$. In case $\kappa(\overline{\gamma}) > \kappa(\gamma)$, then also $\kappa(\overline{\gamma}) > \kappa(\gamma')$, which contradicts optimality of $\overline{\gamma}$. □

The underestimating property is useful since, in an online setting, once an optimal prefix-alignment has nonzero costs, it guarantees that a deviation from the reference model is present. On the other hand, if a case is not properly terminated and will never terminate, yet the sequence of activities seen so far has a prefix-alignment cost of zero, we do not observe this type of deviation until we compute a corresponding conventional (optimal) alignment.

Any shortest path algorithm to compute conventional alignments, i.e. as briefly discussed in Sect.3.2, is easily altered to compute prefix-alignments. In fact, in line with the relaxation of requirement two of Definition 1, such alteration only consists of adding more states to the set of final states of the search problem. Hence, to compute optimal (prefix-)alignments we are able to use any algorithm designed for finding shortest paths in a graph. However, in [3] the A^* algorithm [16] is proposed and evaluated. In this paper, we simply assume that we are able to use an algorithm $\overline{\alpha}$, i.e.:

$$\overline{\alpha} \colon \mathcal{N} \times \mathcal{A}^* \times \mathcal{M} \times \mathcal{M} \to \overline{\Gamma}$$

Here, we assume $\overline{\alpha}(N, \sigma, M_i, M_f) \in \overline{\Gamma}(N, \sigma, M_i, M_f)$ and it is optimal. Observe that the *proper termination assumption*, w.r.t. the process models considered, guarantees that $\overline{\alpha}$ always find an optimal prefix-alignment.

5 Computing prefix-alignments incrementally

In this section, we present an incremental algorithm for the purpose of online prefix-alignment computation. Subsequently, we present effective parametrization of the algorithm that allows us to reduce memory usage and computation time.

5.1 An incremental framework

We aim at computing a prefix-alignment for each sequence of events seen so far for each case $c \in \mathcal{C}$. In this paper, we primarily focus on the performance of prefix-alignment computation in an incremental setting; we therefore do not consider (the impact of) storing the information seen on the event stream in great detail. Henceforth, we assume the existence of a case administration $D_{\mathcal{C}} \colon \mathcal{C} \times \mathbb{N}_0 \to \overline{\Gamma}$, where, for $i \geq 1$, $D_{\mathcal{C}}(c, i)$ represents the currently known prefix-alignment related to case c after receiving events $S(1), S(2), \ldots, S(i)$. Initially, we have $D_{\mathcal{C}}(c, 0) = \epsilon, \forall c \in \mathcal{C}$. For now, we assume that $D_{\mathcal{C}}$ is able to store all most recent prefix-alignments for all cases. Such assumption, theoretically, requires infinite memory; hence in Sect. 5.3, we briefly

Algorithm 1: `Incremental Prefix-Alignments`

input: $N = (P, T, F, \lambda), M_i, M_f : P \to \mathbb{N}_0, \overline{\alpha} : \mathcal{N} \times \mathcal{A}^* \times \mathcal{M} \times \mathcal{M} \to \overline{\Gamma}, S \in (\mathcal{C} \times \mathcal{A})^*$

begin

1. $i \leftarrow 0$;
2. **while** `true` **do**
3. $\quad i \leftarrow i + 1$;
4. $\quad (c, a) \leftarrow S(i)$;
5. $\quad \overline{\gamma} \leftarrow D_{\mathcal{C}}(c, i-1)$;
6. $\quad$ copy all alignments of $D_{\mathcal{C}}(c', i-1)$ to $D_{\mathcal{C}}(c', i)$ for all $c' \in \mathcal{C}$;
7. $\quad$ let M be the marking of N obtained by $\overline{\gamma}$;
8. $\quad$ **if** $\exists_{t \in T}(\lambda(t) = a)$ **then**
9. $\quad\quad$ **if** $\exists_{t \in T}(\lambda(t) = a \wedge M[t\rangle)$ **then**
10. $\quad\quad\quad$ let t denote such transition;
11. $\quad\quad\quad D_{\mathcal{C}}(c, i) \leftarrow \overline{\gamma} \cdot \langle (a, t) \rangle$;
12. $\quad\quad$ **else**
13. $\quad\quad\quad \sigma \leftarrow (\pi_1(\overline{\gamma}))_{\downarrow_{\mathcal{A}}}$;
14. $\quad\quad\quad D_{\mathcal{C}}(c, i) \leftarrow \overline{\alpha}(N, \sigma \cdot \langle a \rangle, M_i, M_f)$;
15. $\quad$ **else**
16. $\quad\quad D_{\mathcal{C}}(c, i) \leftarrow \overline{\gamma} \cdot \langle (a, \gg) \rangle$;

discuss how to handle this problem, and the corresponding limitations in practice.

To compute prefix-alignments based on the event stream, we conceptually perform the following steps. When we receive an event related to a certain case, we check whether we previously computed a prefix-alignment for that case. In case we are guaranteed that the event refers to an activity move, i.e. because the activity simply has no corresponding label in the reference model, we append such activity move to the prefix-alignment. If this is not the case, we fetch the marking in the reference model, corresponding to the previous prefix-alignment. For example, given prefix alignment $\langle (a, t_1) \rangle$ based on N_1 (Fig. 2), the corresponding marking is $[p_1, p_2]$. If the event is the first event received for the case, we simply obtain marking M_i. In case we are able to directly fire a transition within the obtained marking with the same label as the activity that the event refers to, we append a corresponding synchronous move to the previously computed prefix-alignment. Otherwise we use a shortest path algorithm, of which we present some parametrization in Sect. 5.2, to find a new (optimal) prefix-alignment. In Algorithm 1, we present an algorithmic description of the aforementioned rationale.

The algorithm expects a Petri net, initial- and final marking, an algorithm that computes optimal prefix-alignments and an event stream as an input. Note that, after receiving a new event, the case administration for index $i - 1$ is copied into the ith version, i.e. line 6. This operation is $O(1)$ in practice. Since optimal prefix-alignments underestimate conventional alignment costs (Proposition 1), we are interested to what extent Algorithm 1 guarantees optimality of the prefix-alignments stored in $D_{\mathcal{C}}$.

Theorem 1 (Optimality of Algorithm 1) *We let* $D_{\mathcal{C}} : \mathcal{C} \times \mathbb{N}_0 \to \overline{\Gamma}$, *with* $D_{\mathcal{C}}(c, 0) = \epsilon$, $\forall c \in \mathcal{C}$, *and, assume* $D_{\mathcal{C}}$ *is updated according to Algorithm 1. For any* $c \in \mathcal{C}$, $i \in \mathbb{N}$ *and* $\overline{\gamma} = D_{\mathcal{C}}(c, i)$ *we have* $\overline{\gamma} \in \overline{\Gamma}$ *and* $\overline{\gamma}$ *is optimal for* $(\pi_1(\overline{\gamma}))_{\downarrow_{\mathcal{A}}}$.

Proof (Induction on i)

- *Base Case I*: $i = 0$ All alignments are ϵ.
- *Base Case II*: $i = 1$ Let (c, a) be $S(i)$. We know $D_{\mathcal{C}}(c, i-1) = D_{\mathcal{C}}(c, 0) = \epsilon$. In case we are able to fire some t with $\lambda(t) = a$ in M_0, we obtain alignment $\langle (a, t) \rangle$, which, under the unit-cost function, is optimal.[1] In case $\nexists_{t \in T}(\lambda(t) = a)$ we obtain $\langle (a, \gg) \rangle$ which is trivially an optimal prefix-alignment for trace $\langle a \rangle$. In any other case, we compute $\overline{\alpha}(N, M_i, M_f, \langle a \rangle)$ which is optimal by definition.
- *Induction Hypothesis* Let $i > 1$. For any $c \in \mathcal{C}$, we assume that for $\overline{\gamma} = D_{\mathcal{C}}(c, i)$, we have $\overline{\gamma} \in \overline{\Gamma}$ and $\overline{\gamma}$ is optimal.
- *Inductive Step* We prove that, for any $c \in \mathcal{C}$, for $\overline{\gamma} = D_{\mathcal{C}}(c, i+1)$, we have $\overline{\gamma} \in \overline{\Gamma}$ and $\overline{\gamma}$ is optimal. Let (c, a) be $S(i+1)$. In case $D_{\mathcal{C}}(c, i) = \epsilon$ we know that $\overline{\gamma}$ is optimal (*Base Case* $i = 1$). Let $D_{\mathcal{C}}(c, i) = \overline{\gamma}'$ s.t. $\overline{\gamma}' \neq \epsilon$. In case we are able to fire some t with $\lambda(t) = a$ in M_0 we obtain $\overline{\gamma} = \overline{\gamma}' \cdot \langle (a, t) \rangle$. Since, under unit-cost function, $\delta(a, t) = 0$, if $\overline{\gamma}$ is non-optimal, then also $\overline{\gamma}'$ is non-optimal which contradicts the *IH*. A similar rationale holds in case $\nexists_{t \in T}(\lambda(t) = a)$. In any other case, we compute $\overline{\alpha}(N, M_i, M_f, \sigma \cdot \langle a \rangle)$ which is optimal by definition. □

Theorem 1 proves that Algorithm 1 always computes optimal prefix-alignments for $(\pi_{\mathcal{A}\gg}(\overline{\gamma}))_{\downarrow_{\mathcal{A}}}$, i.e. the sequence of activities currently stored within $D_{\mathcal{C}}$ for some $c \in \mathcal{C}$. Hence, combining this result with Proposition 1, we conclude that whenever the algorithm observes certain alignment costs exceeding 0, the corresponding conventional alignment has at least the same costs, or higher.

5.2 Parametrization

In the previous section, we used $\overline{\alpha}$ completely as a black box and always solved a shortest path problem starting from M_i. In this section, we show that we are able to exploit the previously calculated alignment for a case c in order to prune the search state-space. Moreover, we show means to limit the search by changing its starting point.

5.2.1 Cost upper-bounds

Assume that we receive the ith event (c, a) on the stream and we let $\overline{\gamma}' = D_{\mathcal{C}}(c, i-1)$ and $\overline{\gamma} = D_{\mathcal{C}}(c, i)$. Let us write

[1] Observe that this is true under the *proper termination assumption*.

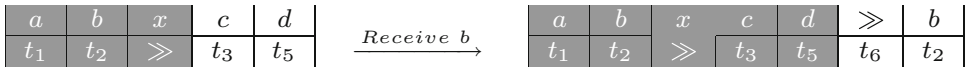


Fig. 7 Partially reverting ($k = 2$) the prefix-alignment of $\langle a, b, x, c, d\rangle$ and N_1 in case of receiving new activity b. The grey coloured moves are not considered when computing the new alignment

the corresponding sequence of activities as $\sigma = \sigma' \cdot \langle a\rangle$. By Proposition 1, we know that $\overline{\gamma}'$ is an optimal prefix-alignment for σ'. It is easy to see that the costs of $\overline{\gamma}'$ together with an activity move on a are an upper bound for the costs of $\overline{\gamma}$, i.e. $\kappa(\overline{\gamma}) \leq \kappa(\overline{\gamma}') + \delta(a, \gg)$. We are able to utilize this knowledge within the shortest path search algorithm $\overline{\alpha}$. Whenever we encounter a path within the search that is (guaranteed to be) exceeding $\kappa(\overline{\gamma}') + \delta(a, \gg)$, we simply ignore it, and all paths extending it.

As indicated, in alignment computation, the A^* algorithm is often used as an instantiation for $\overline{\alpha}$. The A^* algorithm traverses the state space in an implicit manner, i.e. it expects each state it visits to tell which states are their neighbours, and, at what distance. Moreover, it assumes that each state is able to estimate its distance to the closest final state, i.e. each state has a *heuristic* distance estimation to the closest final state. For the purpose of computing (prefix-)alignments, there are two of these heuristic distance functions defined [3, Chapter 4]. The exact characterization of these heuristic functions is out of this paper's scope, i.e. it suffices to know that we are able to, for each marking in the synchronous product net, compute the estimated distance (in terms of alignment costs) to final marking M_f. Moreover, such estimation is always underestimating the true distance. Thus, whenever we encounter a marking M in the state space of which the distance to reach M from M_i, combined with the estimated distance to M_f, exceeds $\kappa(\overline{\gamma}') + \delta(a, \gg)$, we ignore it and all of its possible subsequent markings.

5.2.2 Limiting the search

Again, assume we receive the ith event (c, a) and we let marking M be the marking obtained by executing the transitions of $\overline{\gamma}' = D_{\mathcal{C}}(c, i-1)$. In case there exist transitions t with $\lambda(t) = a$, yet none of these transitions are enabled in M, the basic algorithm simply utilizes $\overline{\alpha}(N, \sigma \cdot \langle a\rangle, M_i, M_f)$. In general, the shortest path algorithm does not need M_i as a start state, i.e. we are able to choose any marking of N as a start state. Hence, we propose to *partially revert* alignment $\overline{\gamma}'$ up to a maximal revert distance k and start the shortest path search from the corresponding marking. Doing so however no longer guarantees optimality as we are no longer searching for a global optimum in the state space.

Consider Fig. 7, where we depict a prefix-alignment for $\langle a, b, x, c, d\rangle$ and N_1 (Fig. 2). Assume we receive a new event that states that activity b follows $\langle a, b, x, c, d\rangle$ and we use a revert window size of $k = 2$. Note that the marking related to the alignment is $[p_5]$. In this marking, no transition with label b is enabled and the algorithm normally calls $\overline{\alpha}(N_1, \langle a, b, x, c, d, b\rangle, [p_i], [p_o])$. However, we revert the alignment two moves, i.e. we revert (d, t_5) and (c, t_3) and call $\overline{\alpha}(N_1, \langle c, d, b\rangle, [p_2, p_3], [p_o])$ instead. The result of this call is $\langle (c, t_3), (d, t_5), (\gg, t_6), (b, t_2)\rangle$, depicted on the right-hand side of Fig. 7. Note that after this call, the window shifts, i.e. the call appended two moves, and thus, (c, t_3) and (d, t_5) are no longer considered upon receiving of new events.

5.3 Administering cases in finite memory

Thus far, we assumed $D_{\mathcal{C}}$ to be of infinite memory, an infeasible assumption in practice. In an online setting, case administration $D_{\mathcal{C}}$ needs to deploy some form of memory management that removes entries based on some case characteristic, e.g. age, relative frequency on the stream, etc. Examples of such mechanisms are reservoirs [17,18], time decay-based data structures [19] and frequency approximation algorithms [20]. These techniques, at some point, remove prefix-alignments related to some, seemingly inactive, case. Note that this no longer guarantees that a prefix-alignment maintained in $D_{\mathcal{C}}$ is always an underestimation for *all activities* emitted on the stream for case c. In fact, Theorem 1 actually does not prove this, as it proves optimality for $(\pi_1(\overline{\gamma}))_{\downarrow_{\mathcal{A}}}$, which under assumption of infinite memory is equivalent to the previous statement. Moreover, if we receive activities related to a case that was previously deleted, we are falsely starting to compute new prefix-alignments for the case, i.e. there was some past behaviour that we no longer possess.

A way to solve this problem is by tracking what cases are removed from case administration. If new events appear related to such case, we simply ignore them. In such way, any element of the case administration truly relates to all behaviour emitted on the stream related to the case. However, again, at some point in time we need to drop cases from the secondary storage component. It is, however, reasonable to assume that the number of distinct cases is orders of magnitudes smaller than the number of events emitted onto the event stream.

Table 2 Parameters used in experiments

Parameter	Type	Values
Use Upper-Bound	`Boolean`	`true`, `false`
Window Size	`Integer`	$\{1, 2, 3, 4, 5, 10, 20, \infty\}$

6 Evaluation

We have evaluated the proposed algorithm, including its parametrization, using the `RapidProM` [21] extension for `RapidMiner`.[2] As a search algorithm, we use a the A^* algorithm provided by `hipster4j` [22]. To evaluate the proposed algorithm, we generated several process models with different characteristics, i.e. different degrees of parallelism, choice and loops. Additionally we evaluated our approach using real event data, related to the treatment of hospital patients suspected of having sepsis. In this experiment, we additionally compare computing prefix-alignments with repeatedly computing conventional alignments on an event stream.

6.1 Experimental set-up

We used a scientific workflow implemented in `RapidProM` which, conceptually, performs the following steps:

1. Generate a (block-structured) workflow net with k labelled transitions, where k is drawn from a triangular distribution with parameters $\{10, 20, 30\}$, for increasing levels of Parallelism, Choice and Loops (from 0 to 50% in steps of 10%) [23].
2. For each workflow net, generate an event log with 1000 cases.
3. For each event log, add increasing levels (from 0 to 50% in steps of 10%) of one type of noise, i.e. *remove activity*, *add activity* or *swap activities*.
4. For each "noisy" event log, do incremental conformance checking against the workflow net it was generated from, using all parameter combinations presented in Table 2.

Observe that within the experiments we mimic event streams by visiting each event in a trace, one by one, e.g. if we have event log $L = [\langle a, b, c\rangle, \langle a, c, b\rangle]$, we generate event stream $\langle (1, a), (1, b), (1, c), (2, a), (2, c), (2, b)\rangle$. Moreover, we align every *trace-variant* once, i.e. if $\langle a, b, c\rangle$ occurs multiple times in the event log, we only align it once. In total, we have generated 18 different models, for which we generate 18 different event logs, each containing 1000 traces, yielding 18.000 noise-free traces. After applying the different types of noise, we obtain a total of 324.000 traces. Clearly, the number of events per trace greatly varies depending on the generated model; however, within our experiments, in total 44.537.728 events were processed (with varying algorithm parametrization). Out of these events, 12.151.510 state-space searches were performed.

6.2 Results

Here, we present the results of the experiments, in line with the parametrization options as described in Sect. 5.2. We first present results related to using cost upper-bounds; later we present the results related to limited search.

6.2.1 Cost upper-bounds

In this section, we present the results related to the performance of using cost upper-bounds. Within these results, we only focus on execution of $\overline{\alpha}$ with M_0 as a start state, i.e. we do not incorporate results related to varying window sizes. In Fig. 8, we present, in terms of the level of introduced noise, the average number of enqueued states, queue size, visited nodes and traversed arcs for each search in the state space.

Clearly, using the upper bound as a cut-off value in state-space traversal greatly enhances the memory efficiency. We observe that, when using the upper bound defined in Sect. 5.2, the average number of states enqueued during the search is less than half compared to not using the upper bound. The average queue size, i.e. average number of states in the queue throughout the search, is much lower in case of using a lower bound. We observe that the search efficiency (Fig. 8c, d) is positively affected by using the upper bound; however, the difference is less severe and in some cases negligible (0% noise level). Thus, using previously computed prefix-alignment values for a case allows effective states-space pruning in the shortest path algorithm.

In Fig. 9, we show the effect of the length of the prefix that needs to be aligned in terms of memory consumption. We only show results for length ≤ 100. Both in case of using and not using the upper bound, we observe a linear increase in number of states queued and average queue size. However, the rate of growth is much lower when using an upper bound. We observe a small region of spikes in both charts around prefix-length 20–25. After investigating the distribution of prefix-length w.r.t. type of process model, i.e. containing loops vs. not containing loops, we observed that most traces exceeding such length are related to the models containing loops. As the other group of models contains relatively more parallelism and/or choices, the complexity of the underlying shortest path search is expected to be slightly more complex, which explains the spikes for relatively short prefix-lengths.

[2] All (raw) experiment results, data and scientific workflows used are available via https://github.com/s-j-v-zelst/research/releases/download/2017_jdsa/experiments.tar.gz.

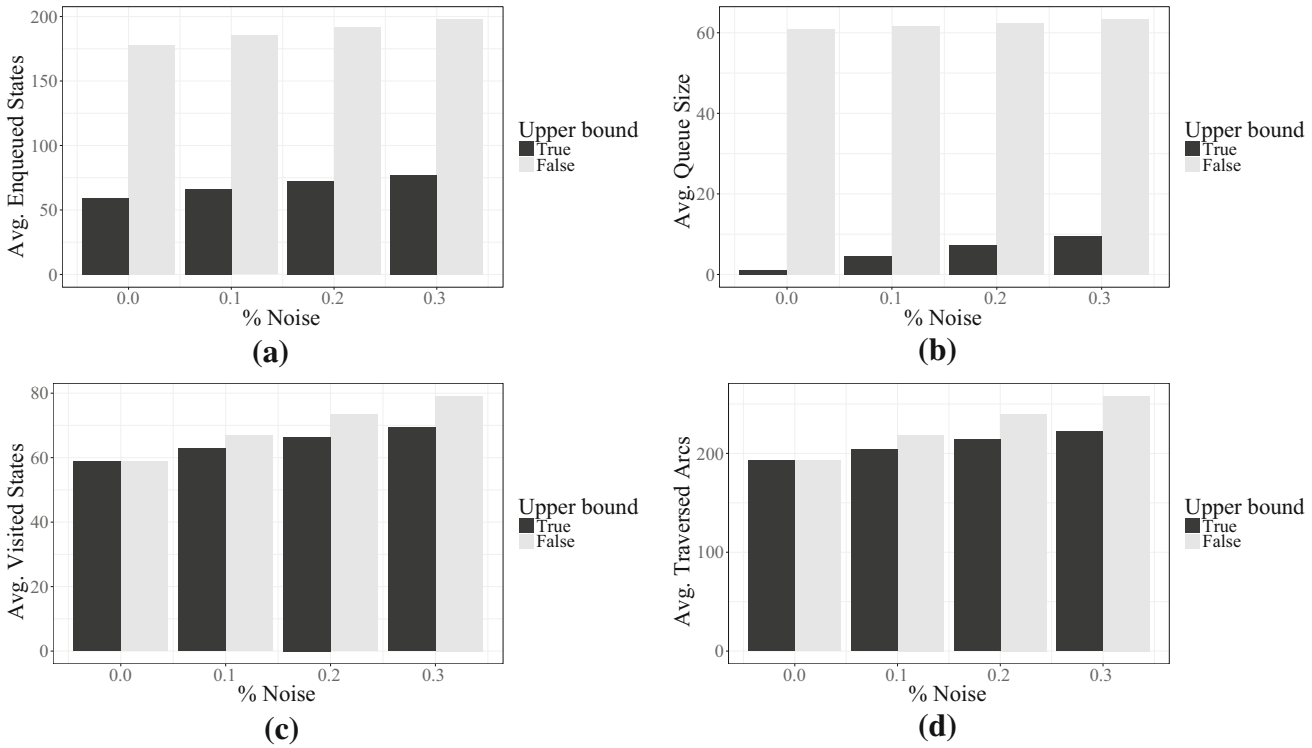


Fig. 8 Performance results of using upper bounds while searching for optimal prefix-alignments. **a** average number of states enqueued, **b** average number queue size, **c** average number of states visited and **d** average number of traversed arcs

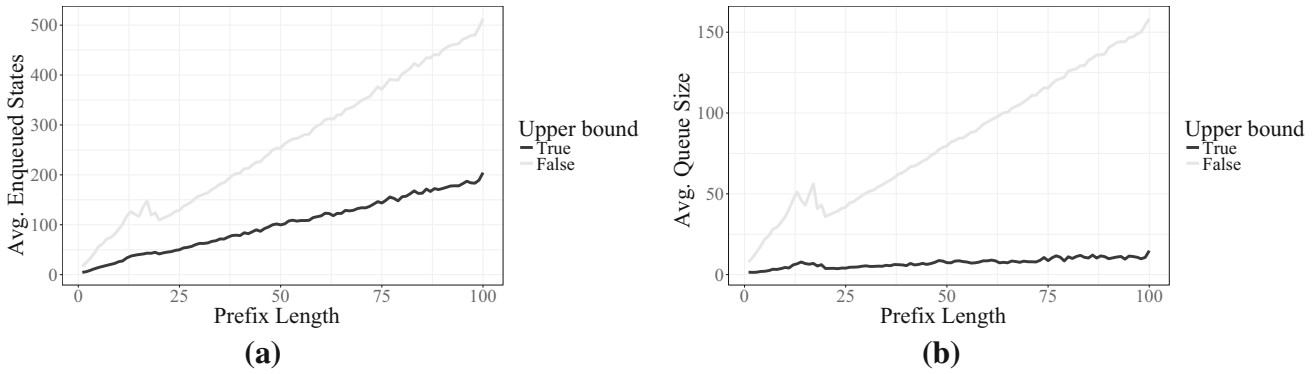


Fig. 9 Memory performance per prefix-length (up to length 100). **a** Average number of states enqueued and **b** average queue size

6.2.2 Reverting alignments

In this section, we present results related to the performance and approximation quality of using revert windows as described in Sect. 5.2.2. In Fig. 10, we present performance results in terms of memory efficiency and approximation error, plotted against noise level. In Fig. 10a, we show the average number of states enqueued when using different revert window sizes. Clearly, the memory usage increases when we increase the window size. Interestingly, this increase seems linear. The approximation error (Fig. 10b) shows an inverse pattern; however, the decrease in approximation error seems nonlinear, when the window size increases. Moreover, in case we set the window size to 5 we observe that the approximation error, within this experiment, is negligible, whereas memory-wise window sizes of 10 and 20 use much more memory while hardly improving the quality of the result.

In Fig. 11, we present performance results in terms of memory efficiency and approximation error, plotted against prefix length. We observe that in terms of enqueued nodes (Fig. 11a), at first a rapid increase appears, after which a steep decline is present. Stabilization around lengths ≥ 25 is again due to the fact that all traces of such length originate from models with loops. The peak and decline behaviour is explained by the fact that the complexity of solving state-

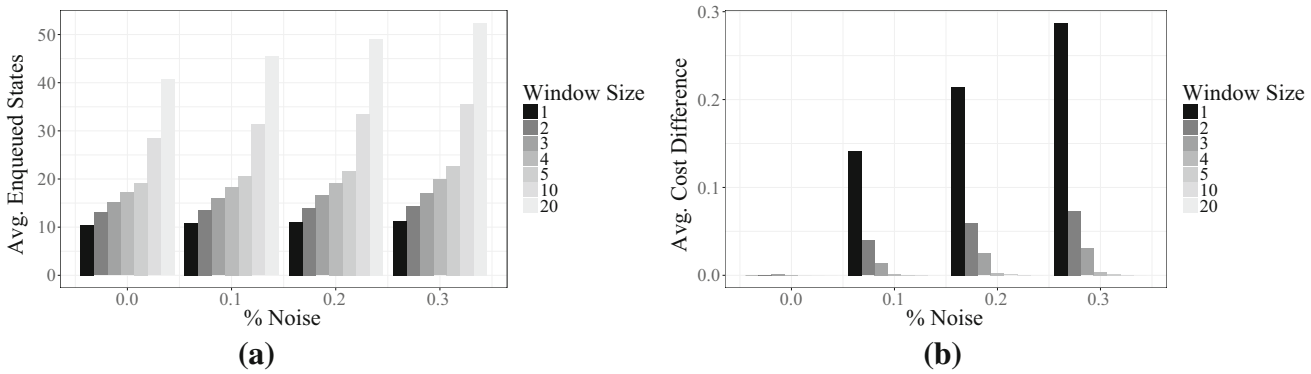


Fig. 10 Memory performance and cost difference w.r.t. optimal prefix-alignments when using different revert window sizes. **a** Average number of states enqueued and **b** average cost difference

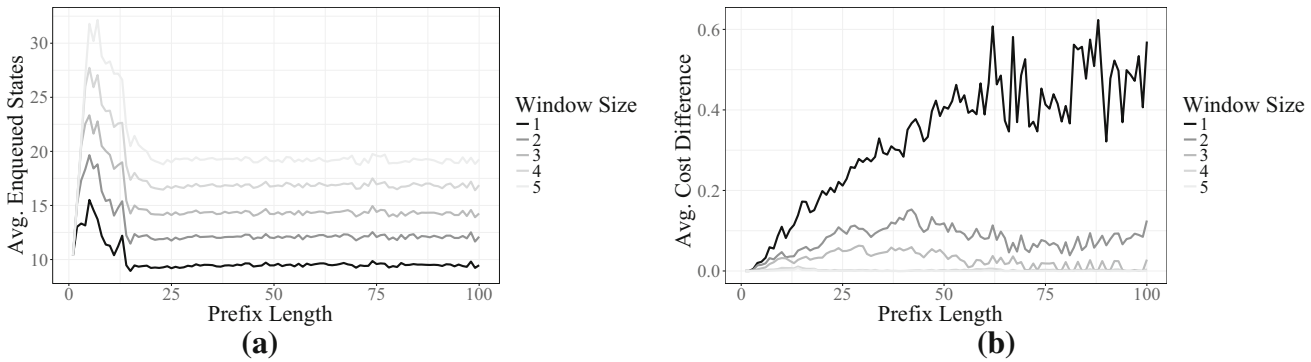


Fig. 11 Memory usage and cost difference w.r.t. optimal prefix-alignments per prefix-length, with different revert window sizes. **a** Average number of states enqueued and **b** average cost difference

space-based search within the models is most likely to be most complex in the middle of the trace. Towards the end of a model's behaviour, we expect less state-space complexity, which explains the decline in the chart around prefix length 10–20.

In Fig. 11b, we observe similar results as observed in Fig. 10b. A window size of 1 is simply too small and it seems that, when the prefix length increases, the costs increase linearly. However, when using a window ≥ 2 we observe asymptotic behaviour in terms of approximation error. Again we observe that using a window of at least size 5 leads to negligible approximation errors.

6.3 Evaluation using real event data

In this section, we discuss the results of applying incremental alignment calculation based on real event data. We focus on the under/overestimation of true eventual conventional alignment cost, as well as the method's performance. As a baseline, we compute conventional alignments every time we receive a new event. We use an event log originating from a Dutch hospital related to the treatment of patients suspected of having sepsis [24]. Since we do not have a reference model, we generated one based on a subset of the data. This generated process model still describes around 90% of the behaviour within the event log (computed using conventional alignments). The data set contains 15.214 events divided over 1.050 cases. Prefix-alignments were computed for 13.775 different events. We plot all results w.r.t. the aligned prefix length as noise percentages, i.e. used in Figs. 8 and 10, are unknown when using real event data. Finally note that the distribution of trace length within the data is heavily skewed and has a long infrequent tail. The majority of the trace's length is below 30, hence, figures for prefix lengths above this value refer to a relatively limited set of cases. Nonetheless, we plot all results for all possible prefix lengths observed.

In Fig. 12, we present results related to computed alignment costs. We show results for using the incremental scheme proposed in this paper with window sizes 5, 10 and 20, and, the baseline ("*Conventional*"). In Fig. 12a, we show the average absolute alignment costs per prefix length. We observe that using a window size of 5 in general leads to higher alignment costs. This is explained by the fact that the relatively little window size does not allow us to revert any choices made in previous alignments, which consequently does not allow us to find an eventual global optimum. Inter-

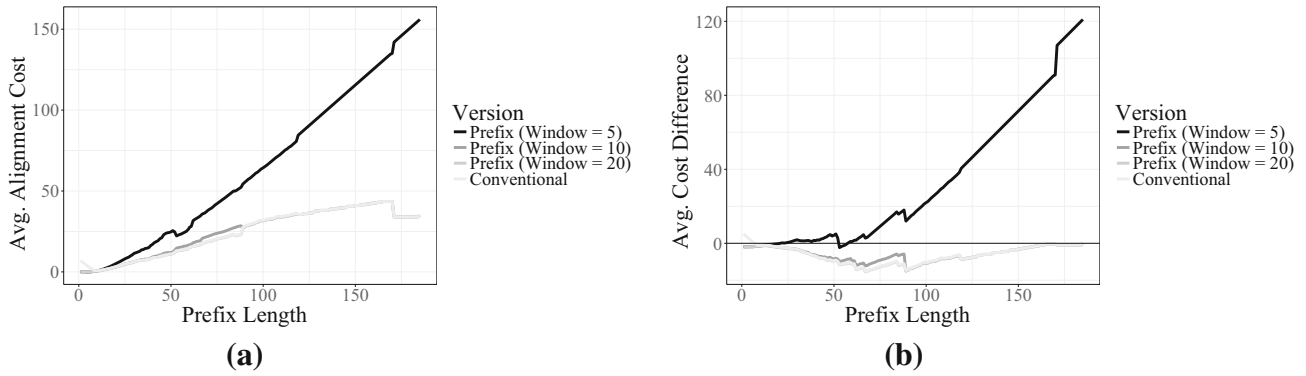


Fig. 12 Average cost results per prefix-length, with different revert window sizes. **a** Average (prefix-) alignment cost and **b** average cost difference w.r.t. eventual optimal conventional alignment

estingly, both window sizes 10 and 20 lead to, on average, comparable alignment costs to simply computing conventional alignments. However, in the beginning of cases, i.e. for small prefixes, as expected, computing conventional alignments leads to higher values. In Fig. 12b, we show the average cost difference w.r.t. the eventual alignment costs, i.e. after case completion. Interestingly, after initially overestimating eventual costs, conventional alignments underestimate the costs of conventional alignments quite severely. This can be explained by the fact that partial traces are aligned by a short path of model moves through the model combined with a limited set of activity moves.

In order to quantify the potential business impact of applying the (prefix-)alignment approach, we derive several different measures of relevance for the three different window sizes and the baseline. These figures are presented in Table 3. To obtain the results as presented, for each received event we define:

- If the difference of the current (prefix-)alignment cost with the eventual alignment cost is zero, and the eventual costs exceed zero, we define a *True Positive*, i.e. we have an exact estimate of non-compliant behaviour.
- If the difference of the current (prefix-)alignment cost with the eventual alignment cost is greater than zero, we define a *False Positive*, i.e. we overestimate non-compliant behaviour.
- If the difference of the current (prefix-)alignment cost with the eventual alignment cost is zero, and the eventual costs equals zero, we define a *True Negate*, i.e. we have an exact estimate of the fact that no deviation occurs.
- If the difference of the current (prefix-)alignment cost with the eventual alignment cost is lower than zero, we define a *False Negative*, i.e. we underestimate non-compliant behaviour.

We acknowledge that alternative definitions of True/False Positives/Negatives are possible. Therefore, the results obtained are specific for the definition provided, as well as the data set of use. We observe that computing conventional alignments, for every event received, leads to better *recall*, i.e. $\frac{TP}{TP+FN}$. This implies that the ratio of correctly observed deviations w.r.t. neglected deviations is better for the conventional approach. However, using the incremental scheme leads to significantly higher *specificity* ($\frac{TN}{TN+FP}$) and *precision* values ($\frac{TP}{TP+FP}$). Specifically for window sizes 10 and 20 we observe very high precision values. This in fact is in line with Proposition 1 and, moreover, shows that the results obtained with these window sizes are close to results for an infinite window size. Finally, we observe that the accuracy of window sizes 10 and 20 is comparable and higher than the alternative approaches, i.e. window size 5 and conventional. However, in terms of $F1$-score, simply calculating conventional alignments outperforms using the incremental scheme as proposed.

In Fig. 13, we show the performance of the different approaches in terms of enqueued states and visited states. Note that the results presented consider the full incremental scheme, i.e. if we are able to execute a synchronous move directly, queued/visited states equals 0. As expected, using a window size of 5 is most efficient. Window sizes 10 and 20 are less efficient yet for longer prefix lengths, they outperform computing conventional alignments. For window size 20, we do observe a peak in terms of computational complexity for prefix lengths of 10–20. Such peak is explained by the relatively inaccurate heuristic used within the A^*-searches performed for prefix-alignment computation. The drops in the chart relate to purely incremental alignment updates. We observe that computational complexity of conventional alignment computation is in general increasing when prefix length increases. The incremental-based approach seems not to suffer from this and shows relatively stabilizing behaviour.

Based on the experiments using real hospital data, we conclude that, for this specific data set, a window size of 10 is appropriate. As opposed to computing conventional

Table 3 Measures of relevance for different window-sized approaches versus computing conventional alignments

Variant Window ∼ 5 Window ∼ 10 Window ∼ 20 Conventional

Variant	*Window ∼ 5*	*Window ∼ 10*	*Window ∼ 20*	*Conventional*
True Positive	1517	2060	2110	3179
False Positive	2076	204	13	5215
True Negative	2962	3340	3377	1424
False Negative	7220	8171	8275	3957
Recall	0.173629392	0.201348842	0.20317766	0.445487668
Specificity	0.587931719	0.942437923	0.996165192	0.214490134
Precision	0.422209852	0.909893993	0.99387659	0.378722897
Negative Predictive Value	0.29090552	0.290157241	0.28982149	0.264634826
False Negative Rate	0.826370608	0.798651158	0.79682234	0.554512332
Fall-out	0.412068281	0.057562077	0.003834808	0.785509866
False Discovery Rate	0.577790148	0.090106007	0.00612341	0.621277103
False Omission Rate	0.70909448	0.709842759	0.71017851	0.735365174
Accuracy	0.325154265	0.392014519	0.398330309	0.33415608
F1-Score	0.246066504	0.329731893	0.337384074	0.409401159

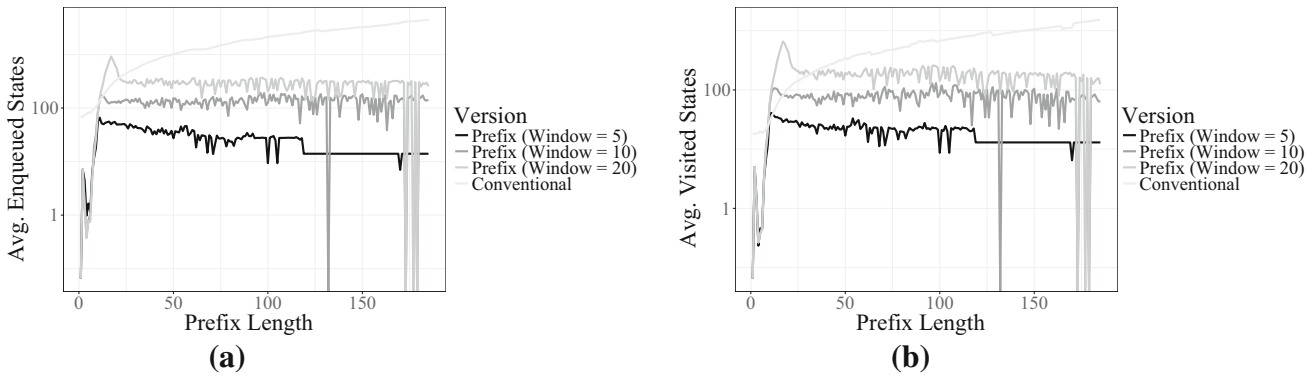


Fig. 13 Performance results based on the hospital data set (logarithmic scales). **a** Average number of states enqueued and **b** Average number of traversed arcs

alignments, it achieves precise results, i.e. whenever a deviation is detected it is reasonable to assume that this is indeed the case. Moreover, it outperforms computing conventional alignments in terms of computational complexity and memory usage.

7 Discussion

The aim of the incremental technique presented in this paper is to compute approximations of alignments, by means of utilizing the concept of prefix-alignments, based on event streams. In particular, we aim at computing these approximations more efficiently w.r.t. simply computing conventional alignments, whilst at the same time limiting the loss in result accuracy.

In general, we conclude that the use of the technique proposed is justified in cases where computational resources are limited, and/or there is a need for high precision, i.e. we aim at high degrees of certainty when we observe a deviation. In cases where computational complexity is not an urgent issue, and/or high recall is more preferable, one can resort to computing conventional alignments. However, recall that conventional alignments initially overestimate alignment costs and thus are not able to properly detect deviations in early stages of a case. Hence, when resorting to using conventional alignments, a warm-up period is advisable.

In the experiments performed using real data, we observe a certain unpredictability w.r.t. memory usage/computational efficiency of prefix-alignment computation, i.e. consider the peeks for window size 20 in Fig. 13. In general, although the search algorithm used in prefix-alignment computation is A^* [16], the practical search performance is, however, equal to the performance of Dijkstra's shortest path algorithm [25]. This is mainly related to the fact that when computing prefix-alignments we need to resort to a rather inaccurate heuristic function. By partially reverting the alignments, combined with applying the upper-bound pruning, we are able to reduce

the search complexity, however, at the cost of losing accuracy. When computing conventional alignments, we are able to resort to a more accurate heuristic which explains the more predictable computational efficiency trends in Fig. 13.

8 Conclusion

In this paper, we proposed an online, event stream-based, conformance checking technique based on the use of prefix-alignments. The algorithm only performs a state-space search to compute a new prefix-alignment if no direct label- or synchronous move is possible. We presented two techniques to increase the search efficiency of the underlying shortest path problems solved. The first technique preserves optimality and allows for effective state-space pruning. The second technique uses an approximation scheme providing a balance between optimality and memory usage. In our evaluation, we primarily focussed on the performance of the underlying shortest path problems solved. Our results show that we are able to effectively prune the state space by using previously computed alignment results. Particularly in terms of memory efficiency, these results are promising. When using our approximation approach, we observe a linear trend in needed memory when increasing window sizes. However, the approximation error seems to decrease more rapidly, i.e. in a nonlinear fashion when increasing window sizes.

Future Work We aim to extend our work as follows. We plan to extend our experiments, using more levels of choice/parallelism/loops, more models per level and larger data sets. Moreover, we plan to perform more experiments using real event data. We also plan to define alternative accuracy measures regarding under/overestimation of conventional alignments to more accurately measure the indicative behaviour of prefix-alignments. Finally, the state of a prefix-alignment, in terms of the underlying reference model, carries some predictive value w.r.t. case termination. Thus, in cases we do not know explicit case termination, it is interesting to study the effect of using prefix-alignments as a case termination predictor.

References

1. van der Aalst, W.M.P.: Process Mining—Data Science in Action, 2nd edn. Springer, Berlin (2016). https://doi.org/10.1007/978-3-662-49851-4
2. 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
3. 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
4. de Leoni, M., van der Aalst, W.M.P.: Data-aware process mining: discovering decisions in processes using alignments. In: Shin, S.Y., Maldonado, J.C. (eds.) Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC'13, Coimbra, Portugal, pp. 1454–1461. ACM, 18–22 March 2013 (2013). https://doi.org/10.1145/2480362.2480633
5. Fahland, D., van der Aalst, W.M.P.: Model repair—aligning process models to reality. Inf. Syst. **47**, 220–243 (2015). https://doi.org/10.1016/j.is.2013.12.007
6. van der Aalst, W.M.P.: Decomposing Petri nets for process mining: a generic approach. Distrib. Parallel Databases **31**(4), 471–507 (2013). https://doi.org/10.1007/s10619-013-7127-5
7. Munoz-Gama, J., Carmona, J., van der Aalst, W.M.P.: Single-entry single-exit decomposed conformance checking. Inf. Syst. **46**, 102–122 (2014). https://doi.org/10.1016/j.is.2014.04.003
8. Taymouri, F., Carmona, J.: A recursive paradigm for aligning observed behavior of large structured process models. In: La Rosa, M., Loos, P., Pastor, O. (eds.) BPM 2016, Riode Janeiro, Brazil, Proceedings,*Lecture Notes in Computer Science*, vol. 9850, pp. 197–214. Springer,18–22 Sept 2016 (2016). https://doi.org/10.1007/978-3-319-45348-4_12.http://dx.doi.org/10.1007/978-3-319-45348-4
9. Burattin, A., Sperduti, A., van der Aalst, W.M.P.: Control-flow discovery from event streams. In: Proceedings of the IEEE Congress on Evolutionary Computation, CEC 2014, Beijing, China, pp. 2420–2427. IEEE, 6–11 July 2014 (2014). https://doi.org/10.1109/CEC.2014.6900341
10. Weijters, A.J.M.M., Ribeiro, J.T.S.: Flexible heuristics miner (FHM). In: Proceedings of the IEEE Symposium on Computational Intelligence and Data Mining, CIDM 2011, Part of the IEEE Symposium Series on Computational Intelligence 2011, Paris, France, pp. 310–317, 11–15 April 2011 (2011). https://doi.org/10.1109/CIDM.2011.5949453
11. Hassani, M., Siccha, S., Richter, F., Seidl, T.: Efficient process discovery from event streams using sequential pattern mining. In: IEEE Symposium Series on Computational Intelligence, SSCI 2015, Cape Town, South Africa, pp. 1366–1373. IEEE, 7–10 Dec 2015 (2015). https://doi.org/10.1109/SSCI.2015.195
12. van Zelst, S.J., van Dongen, B.F., van der Aalst, W.M.P.: Event stream-based process discovery using abstract representations. Knowl. Inf. Syst. (2017). https://doi.org/10.1007/s10115-017-1060-2
13. Burattin, A., Cimitile, M., Maggi, F.M., Sperduti, A.: Online discovery of declarative process models from event streams. IEEE Trans. Serv. Comput. **8**(6), 833–846 (2015). https://doi.org/10.1109/TSC.2015.2459703
14. Murata, T.: Petri nets: properties, analysis and applications. Proc. IEEE **77**(4), 541–580 (1989). https://doi.org/10.1109/5.24143
15. 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
16. Hart, P.E., Nilsson, N.J., Raphael, B.: A formal basis for the heuristic determination of minimum cost paths. IEEE Trans. Syst. Sci. Cyberne. **4**(2), 100–107 (1968). https://doi.org/10.1109/TSSC.1968.300136
17. Aggarwal, C.C.: On biased reservoir sampling in the presence of stream evolution. In: Dayal, U., Whang, K., Lomet, D.B., Alonso, G., Lohman, G.M., Kersten, M.L., Cha, S.K., Kim, Y. (eds.) Proceedings of the 32nd International Conference on Very Large Data

Bases, Seoul, Korea, pp. 607–618. ACM, 12–15 Sept 2006 (2006). http://dl.acm.org/citation.cfm?id=1164180
18. Vitter, J.S.: Random sampling with a reservoir. ACM Trans. Math. Softw. **11**(1), 37–57 (1985). https://doi.org/10.1145/3147.3165
19. Cormode, G., Shkapenyuk, V., Srivastava, D., Xu, B.: Forward decay: a practical time decay model for streaming systems. In: Ioannidis, Y.E., Lee, D.L., Ng, R.T. (eds.) Proceedings of the 25th International Conference on Data Engineering, ICDE 2009, Shanghai, China, pp. 138–149. IEEE Computer Society, 29 March 2009–2 April 2009 (2009). https://doi.org/10.1109/ICDE.2009.65
20. Cormode, G., Hadjieleftheriou, M.: Methods for finding frequent items in data streams. VLDB J. **19**(1), 3–20 (2010). https://doi.org/10.1007/s00778-009-0172-z
21. van der Aalst, W.M.P., Bolt, A., van Zelst, S.J.: RapidProM: mine your processes and not just your data. CoRR (2017). arXiv:1703.03740
22. Rodrguez-Mier, P., Gonzalez-Sieira, A., Mucientes, M., Lama, M., Bugarin, A.: Hipster: an open source java library for heuristic search. In: 2014 9th Iberian Conference on Information Systems and Technologies (CISTI). IEEE (2014). https://doi.org/10.1109/cisti.2014.6876914
23. Jouck, T., Depaire, B.: PTandLogGenerator: a generator for artificial event data. In: Azevedo, L., Cabanillas, C. (eds.) Proceedings of the BPM Demo Track 2016 Co-located with the 14th International Conference on Business Process Management (BPM 2016), Rio de Janeiro, Brazil, *CEUR Workshop Proceedings*, vol. 1789, pp. 23–27. CEUR-WS.org, 21 Sept 2016 (2016). http://ceur-ws.org/Vol-1789/bpm-demo-2016-paper5.pdf
24. Mannhardt, F.: Sepsis cases–event log (2016). https://doi.org/10.4121/uuid:915d2bfb-7e84-49ad-a286-dc35f063a460
25. Dijkstra, E.W.: A note on two problems in connexion with graphs. Numer. Math. **1**(1), 269–271 (1959). https://doi.org/10.1007/BF01386390