I Will Survive: An Event-driven Conformance Checking Approach Over Process Streams

Kristo Raun, Riccardo Tommasini, Ahmed Awad

I Will Survive: An Event-driven Conformance Checking Approach Over Process Streams

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

Riccardo Tommassini
LIRIS Lab, INSA de Lyon, France
University of Tartu, Estonia
riccardo.tommasini@liris.cnrs.fr

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

ABSTRACT

Online conformance checking deals with finding discrepancies between real-life and modeled behavior on data streams. The current state-of-the-art output of online conformance checking is a prefix-alignment, which is used for pinpointing the exact deviations in terms of the trace and the model while accommodating a trace's unknown termination in an online setting. Current methods for producing prefix-alignments are computationally expensive and hinder the applicability in real-life settings.

This paper introduces a new approximate algorithm – I Will Survive (IWS). The algorithm utilizes the trie data structure to improve the calculation speed, while remaining memory-efficient. Comparative analysis on real-life and synthetic datasets shows that the IWS algorithm can achieve an order of magnitude faster execution time while having a smaller error cost, compared to the current state of the art. In extreme cases, the IWS finds prefix-alignments roughly three orders of magnitude faster than previous approximate methods. The IWS algorithm includes a discounted decay time setting for more efficient memory usage and a look-ahead limit for improving computation time. Finally, the algorithm is stress tested for performance using a simulation of high-traffic event streams.

CCS CONCEPTS

• **Theory of computation**; • **Information systems → Data mining**; • **Software and its engineering**;

KEYWORDS

online conformance checking, event-based business process management, prefix-alignments, data streams

ACM Reference Format:
Kristo Raun, Riccardo Tommassini, and Ahmed Awad. 2023. I Will Survive: An Event-driven Conformance Checking Approach Over Process Streams. In *The 17th ACM International Conference on Distributed and Event-based Systems (DEBS '23), June 27–30, 2023, Neuchatel, Switzerland*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3583678.3596887

DEBS '23, June 27–30, 2023, Neuchatel, Switzerland

ACM ISBN 979-8-4007-0122-1/23/06…$15.00
https://doi.org/10.1145/3583678.3596887

1 INTRODUCTION

Process mining [23] is a data-driven approach for analyzing process execution data. The process execution data is commonly collected in event logs. In its simplest form, an event log is a sequence of events characterized by a case identifier, indicating the unique process instance, the label of the executed activity, and a timestamp (Table 2). The sequence of events having the same case identifier is called a trace.

An important aspect of business systems is the ability to detect anomalies and report them in a human-readable form [17]. Conformance checking [9] is the sub-area of process mining that attempts to discover and quantify deviations in business process executions. Conformance checking assumes the prior knowledge of how the world should work – i.e., we have a process model – and examples of how the world is working – i.e., we have process traces. We then compare the traces to the model to analyze the conformance of the process. The state-of-the-art output from conformance checking, in terms of explainability, is an alignment [20]. Importantly, anomalies and non-conformance may not necessarily indicate wrongly executed processes. Deviations may also be a sign of possible process enhancement. Regardless, it is important to be able to find such discrepancies between modeled and actual behavior.

Conformance checking originates in the static setting, where event logs are collected from the business systems and analyzed offline. However, organizations have thousands of ongoing process executions at the same time. Therefore, the analysis of past data quickly loses its value, as deviations usually need to be discovered and acted upon in a timely manner. Such observations have paved the way for **online conformance checking**, where conformance checking is done on infinite event streams rather than logs. Reporting about deviations follows an event-driven fashion to allow process analysts to take action as early as possible. While the underlying goal is the same – finding discrepancies between modeled and real-life behavior – the termination of a single process execution is unknown in an online setting, given that the event stream is unbounded. Thus, computationally efficient algorithms are necessary to keep up with the incoming data.

Problem Statement. While efficient algorithms exist for online conformance checking, they do not use alignments as their output [8], i.e., they do not provide as output a mapping between the event streams and the process model. At the same time, methods using prefix-alignments have been introduced for conformance checking [18, 26], but their computational complexity hinders their applicability in real-life streaming settings.

This paper attempts to bridge this gap by introducing a new efficient algorithm for online conformance checking. The algorithm outputs prefix-alignments with comparable error costs to the state

Concept	Notation	Set notation
Petri net source	i	-
Petri net sink	o	-
Petri net transition	t	-
Silent transition	τ	-
Model behavior	-	M
Proxy behavior	-	M'
Execution sequence on model	π	-
Prefix execution sequence	${}_i\pi$	-
Event	evt	-
Case identifier	$case$	$\mathcal{U}_{case}$
Event activity	act	$\mathcal{U}_{act}$
Trace	σ	-
Activity at i-th position	$\sigma(i)$	-
Trace suffix	$\hat{\sigma}$	-
Event log	-	L
Proxy log	-	L'
Trie	-	T
A node in a trie	n	N
An edge in a trie	e	E
Trie labelling function	l	-
Trie branching factor	bf	-
Alignment	γ	-
Prefix-alignment	$\hat{\gamma}$	-
Skip symbol in an alignment	$\gg$	-
Cost of an alignment	$\delta(\gamma)$	-
A state in the algorithm	s	S
Decay time	dt	-
Discounting factor	df	-
State buffer	-	B
Look-ahead limit	lim	-

Table 1: Notations summary

of the art while improving the computation time by a noticeable extent. The paper is structured as follows: in Section 2 a theoretical background is given. Section 3 introduces the approach and the algorithm. Section 4 compares the algorithm to the state of the art in terms of cost deviations and execution time. A stress test under a fast-paced stream is performed to validate the algorithm's applicability in real-life settings. Finally, Section 5 summarizes the work and provides venues for future research.

2 BACKGROUND

In this section, we introduce the main components necessary for understanding the content of the paper. We describe process models, event logs, conformance checking, and its adaptation to the streaming context. Table 1 summarizes the notations used in the rest of the paper.

2.1 Process Models and Event Logs

A process model defines which sequences of activity executions are considered to be valid. There are many notations to model business processes varying in their richness and formal semantics. For the purposes of this paper, we utilize a special case of a Petri net called a Workflow net [22] (WF-net) where there is a single source place i, and a single sink place o, and any other node, i.e.,

Case identifier	Activity	Timestamp
1	a	2022-08-01 15:00
1	b	2022-08-01 15:02
2	a	2022-08-01 15:03
2	b	2022-08-01 15:06
1	c	2022-08-01 15:06

Table 2: A simple event log showing the case identifier, executed activity, and execution timestamp.

either a place or a transition, is on a path from the source place to the sink place. In other words, adding a transition t to the net with one arc from o to t and another arc from t to i, the resulting Petri net forms a single strongly connected component. WF-nets follow the standard semantics of transitions enablement and firing as ordinary Place/Transition Petri nets [24].

The WF-net in Figure 1, a simplistic order fulfillment process, serves as our running example having five labeled transitions: a, b, c, d, e. Silent transitions (τ transitions) cannot be observed during the execution of a process model and are colored in grey. In Figure 1, the τ transition allows to skip c, but there is no labeled activity associated with skipping c that could be shown on the model.

When WF-nets are enacted, one can observe sequences of labeled transitions based on firing sequences. As such, the model behavior M may also be represented by a set of sequences of activities. M is infinite when the model has loops because a loop can unfold an unlimited number of times. The sequence of fired transitions is called an execution sequence. An execution sequence $\pi \in M$ starts from a transition enabled by the source place, i, marking and ends with a transition that marks the sink place, o, after its firing. A prefix of an execution sequence ${}_i\pi$ indicates the execution until the i-th position in M. An instance of an execution sequence is shown as $\pi = \langle a, b, c, e \rangle$, representing the execution path followed by executing, a, b, c, and e transitions from the WF-net in Figure 1. Another execution sequence $\pi = \langle a, b, d, c, b, d, b, d, e \rangle$ has the loop around transitions b and d executed two times.

Ideally, process models are deployed to process execution engines which ensures faithful executions [11]. In practice, most process execution is unmanaged by a process engine [23]. Rather, they are

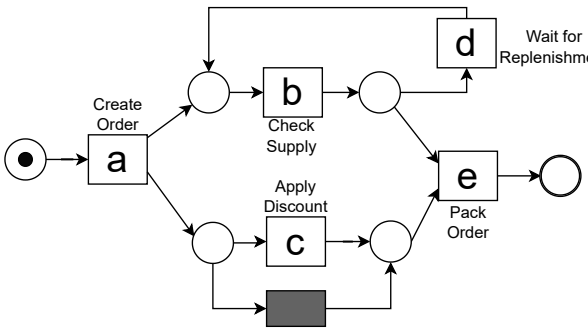

Figure 1: A small example process model with parallelism, skip activity and a loop.

supported by individual information systems that lack an end-to-end tracking of process instances.

One discrete data unit in the execution of a process within information systems is commonly referred to as an **event** (Definition 2.1), while a multiset of events is represented as an **event log** [23] (Definition 2.2.

Definition 2.1 (Event). An event evt is a tuple $evt = (case, act, time) \in \mathcal{U}_{case} \times \mathcal{U}_{act} \times \mathcal{U}_{time}$ with $case$ referring to the case identifier (caseID), act referring to the executed activity and $time$ denoting the event timestamp.

Definition 2.2 (Event log). An event log L is a multiset of events $L \in \mathcal{B}(\mathcal{U}_{case} \times \mathcal{U}_{act} \times \mathcal{U}_{time})$

Definition 2.3 (Trace). A trace $\sigma = \langle act_1, \ldots, act_n \rangle \in \mathcal{U}_{act}$ is a finite sequence of activities with a common caseID. We use the notation $\sigma(i)$ for the activity at the i-th position of σ.

A log contains **traces** (Definition 2.3), a sequence of events, each denoting a single execution of the process. Traces representing distinct process executions built of events that induce the same sequence of activity executions are said to be of the same *trace variant*. A relatively simple model for event logs is sufficient for the context of this paper, i.e., an *event* consists of a *caseID* for assigning an event to a particular process instance, an *activity* label, and a non-decreasing *timestamp*.

A proxy log, L', is an event log that represents a finite subset of behavior – proxy behavior (M') – allowed by the model. For example, the model in Figure 1 allows for infinite behavior due to the model containing a loop. An example of proxy behavior could be limiting looping to a single traversal of activity d. Eliciting such restrictions allows us to generate a proxy log (Table 3) that contains traces describing a finite subset of the model behavior.

2.2 Conformance Checking

Conformance checking compares the behavior recorded in an event log L with the behavior specified by a process model M [9]. Typically, it relies on the concept of alignments [20]. Alignments map the moves in the log (actual behavior) and possible moves in the model. Generally, alignments provide good diagnostics, as it is easy to interpret expected behavior and deviations, e.g., skipping an activity or conducting an activity not expected by the model [21].

Formally, an alignment $\gamma = \langle (x_1, y_1), \ldots, (x_n, y_n) \rangle$ is a sequence of steps, each step $(x, y) \in (\mathcal{U}_{act} \cup \{\gg\}) \times (\mathcal{U}_{act} \cup \{\gg\})$ linking an activity of the trace, or the skip symbol $\gg$, to an activity of the execution sequence, or the skip symbol, whereas a step $(\gg, \gg)$ is illegal. Each activity in the trace and the model are paired in a *move*. Here, it must hold that the projection of γ on the first component, ignoring $\gg$, yields σ, and the projection of γ on the second component, ignoring $\gg$, yields π. A step (x, y) is called *synchronous move* if $x, y \in \mathcal{U}_{act}$, *log move* if $y = \gg$, a *model move* if $x = \gg$, while the last two are jointly referred to as asynchronous moves.

$case$	$\sigma(1)$	$\sigma(2)$	$\sigma(3)$	$\sigma(4)$	$\sigma(5)$	$\sigma(6)$
$case_1$	a	b	c	d	b	e
$case_2$	a	b	c	e		
$case_3$	a	b	d	b	e	
$case_4$	a	b	d	b	c	e
$case_5$	a	b	d	c	b	e
$case_6$	a	b	e			
$case_7$	a	c	b	d	b	e
$case_8$	a	c	b	e		

Table 3: Running example: proxy log

As multiple alignments are possible, a cost is associated with steps for decision-making. In this paper, a cost of one is assigned to asynchronous moves, while synchronous moves have zero cost. Moves on τ transitions can never be observed in the trace; thus, they also have a cost of zero. In the remainder, we write $\delta(\gamma)$ for the total cost of an alignment γ.

An optimal alignment minimizes the edit distance between a trace and an execution sequence. Identifying a cost-optimal (minimal) alignment for a trace and all execution sequences of a model is computationally expensive [9].

Over the years, various alignment-based techniques and outputs have been investigated. Anti-alignments [10], for example, quantify the extreme deviations from the model, allowing for the detection of imprecise models. Some recent techniques [5, 16] have encoded alignment calculations as an SAT problem. Various cost functions have been investigated – some techniques have investigated not obtaining the optimal alignment, but obtaining the maximum amount of synchronous moves [3, 16]. A cost function penalizing earlier deviations of a trace more than later deviations was introduced in [4]. While new methods [15] are emerging, the most common way of finding an optimal alignment requires building a so-called synchronous product net and using the A* search algorithm to find the shortest path through the net [20]. When building the synchronous product net, the search space may grow exponentially. Thus, calculating *optimal* alignments is still considered computationally challenging in real-life settings.

For our running example, let us assume that we have observed a trace $\sigma = \langle a, b, b, c \rangle$. Figure 2 shows two alignments between this trace and the process model. The row marked with σ shows the complete *trace* that has been seen. The row with π shows the corresponding moves in the *model*. For the first alignment, the second execution of b is considered erroneous, and thus a log move is made. Alternatively, the second alignment shows that a model move on d would entail the same cost – in this case, we would assume that the activity d was either skipped or not recorded properly. For both of the alignments, a synchronous move on c, and a model move on e need to be executed for the execution sequence to conclude.

While both of the alignments in Figure 2 are *optimal*, an alignment can also be *suboptimal*, i.e., its associated cost is non-minimal. Approximate algorithms commonly produce alignments that may be suboptimal. Such algorithms quantify the distance from optimality via an *error*. For a more thorough background on conformance checking, we refer to [9].

2.3 Related Works

The general framework for conformance checking on top of event streams – *online conformance checking* – was introduced in [7]. The

approach for conformance checking can calculate conformance in near real-time.

The work in [8] used behavioral patterns for calculating conformance in a streaming setting. Most notably, the method outputs completeness and confidence metrics in addition to conformance. These metrics give additional insights to the user in terms of the reliability of the conformance. Also, the behavioral methods do not penalize *warm starting* scenarios. That is, cases where a process execution has been started before the conformance checking begins. More recently, [13] extended the behavioral approach, basing their method on Hidden Markov Models, alternating between state estimations and calculating conformance. While the methods in this direction are very fast in computation time, they are less informative in diagnosing the causes of deviations. Generally, these methods can be considered trace-level metrics, indicating whether something is wrong and how trustworthy the assessment is. The outcome of utilizing these methods is an alert rather than an alignment, and thus it is hard to pinpoint what exactly is the non-conforming part between the trace and the model.

Another research path has focused on prefix-alignments [1], which were first introduced for process event streams in [26]. In a streaming setting, conformance-checking frameworks usually observe a subsequence of the trace. Indeed, the trace execution may not yet have concluded, and it is unknown how the execution sequence might play out. A *complete* alignment would overestimate the conformance cost in such cases. A **prefix-alignment** $\hat{\gamma}$ is a variation of the alignment where complete path traversal to the model's sink is unnecessary. Returning to the trace $\sigma = \langle a, b, b, c \rangle$, one can deduce that the final event e may still occur, and thus there exists no deviation in terms of the activity e. In this case, there exist two equally optimal prefix-alignments. Prefix-alignments ($\hat{\gamma}$) of trace σ are shown in Figure 2. In a streaming setting, a prefix-alignment is calculated event-by-event, finding a match between the arrived event and allowed model behavior.

The algorithm in [26] uses a window-size parameter to trade-off between the computation time and alignment optimality. An infinite window size allows for calculating optimal prefix-alignments but has the slowest execution time. A window size of one is the fastest, but the alignments produced may be suboptimal. In [18], the authors improved upon their work by introducing an incremental A* algorithm, which can calculate optimal prefix-alignments with a smaller memory footprint. However, computationally, the newer

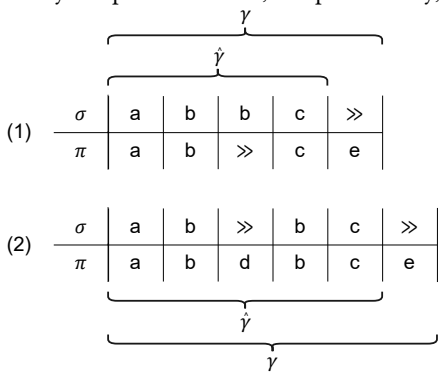

Figure 2: Running example: two optimal alignments and prefix-alignments.

method remained noticeably slower than the initial algorithm with a window size of one. Recent work has seen proposals for various memory-efficient approaches for calculating prefix-alignments in a streaming setting [28]. But in general, due to the reliance on computing synchronous product nets and then doing shortest path traversal, the prefix-alignment methods exhibit a heavy computation load and remain impractical for most real-life scenarios (Section 4).

3 APPROACH

In this section, we introduce our approach named I Will Survive (IWS). In particular, Section 3.1 provides the formal foundation of the IWS data model that consists of event streams, a state buffer, and a trie to represent the proxy log; Section 3.2 drills down into the algorithmic details of IWS, as depicted by *Algorithm* in the pipeline shown in Figure 3.

3.1 Data Model

Our approach IWS assumes the existence of an event log or an event stream and a process model. The process model is simulated in step (1) into a proxy log. From the proxy log, step (2) constructs a trie T. A trie is a particular type of tree, commonly referred to as a prefix tree, where all the children of a node have a common prefix. For the trie construction, we need a finite set of traces, i.e. a proxy log [2]. Definition 3.1 gives a formal definition of the trie in the context of this work. The trie is computed offline in the pre-processing step and is considered immutable as the algorithm executes. That is, the underlying process is not expected to change during the conformance checking.

Utilizing a proxy log such as in Table 3 leads to the construction of the trie shown in Figure 4. The trie is a more concise representation of the proxy log, e.g., all the eight traces start with the activity a, which is represented as a single node in the trie.

The trie is given as input to the algorithm in step (3) when the algorithm is initialized. In step (4), the algorithm expects an event coming from an event log or event stream and consisting of a caseID and an event activity. The algorithm checks for conformance and stores a list of states – a *state buffer* – for each caseID in step (5). Finally, two optional steps are step (6), for fetching the latest prefix-alignment for a case, and step (7), for calculating and fetching a complete alignment, that is permissible by the state buffer. Notably, as the algorithm holds a list of states with prefix-alignments, it is possible to use different methods, such as [2] or [12], in step (7) for finding the complete alignment from a prefix-alignment. However, for the purposes of this paper, fetching complete alignments is out of scope as we aim to produce prefix-alignments in the context of event streams.

In a streaming setting, the events are expected to be processed one by one in the temporal order. Furthermore, it is common that multiple cases are ongoing simultaneously, meaning that events coming in belong to different cases. The algorithm needs to keep track of the seen cases and their states while performing optimizations for low memory consumption. For handling these demands, the definitions for a state, decay time, state buffer, and look-ahead limit are introduced.

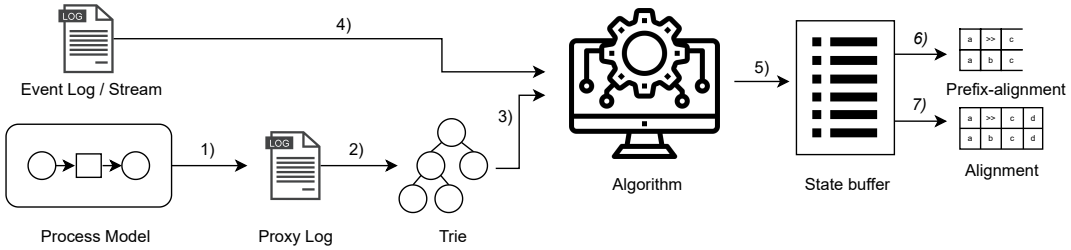

Figure 3: Approach overview

Definition 3.1 (Trie). Let $M' \subseteq M$ be some proxy behavior of a process model. Then, the *trie* constructed for it is a structure $T = (N, E, root, l)$ where:

- N is a finite set of nodes. There is one node per prefix ${}_i\pi$ for any execution sequence $\pi \in M'$ as well as one additional node $root \in N$.
- $E \subset N \times N$ is a set of edges, s.t. for all $n \in N$ it holds $|\{n' \mid (n', n) \in E\}| \leq 1$ and (N, E) is a connected graph. There are edges from $root$ to all nodes representing prefixes of length one, and from each node n to node n', if the prefix represented by n' is obtained from the prefix of n by concatenation with a single activity.
- $root \in N$ is the root of the trie, i.e., the only node $n \in N$ for which $|\{n' \mid (n', n) \in E\}| = 0$;
- $l : N \rightarrow (\mathcal{U}_{act} \cup \{\perp\})$ is a labeling function for nodes. The label is the activity of the prefix represented by the node, while $root$ is assigned $\perp$.

Definition 3.2 (State). A state s is a tuple $(n, \hat{\gamma}, \hat{\sigma}, \delta(\hat{\gamma}), dt)$, where n is the current node in the trie, $\hat{\gamma}$ is the prefix-alignment up to this node, $\hat{\sigma}$ is the trace suffix, $\delta(\hat{\gamma})$ is the total cost of the current state, and dt is the associated decay time of the state.

Definition 3.3 (Decay time). Let $\mathbb{N} = \{1, 2, 3, \ldots, \bowtie\}$ where $\bowtie$ is a large natural number. Then, *decay time* $dt \in \mathbb{N}$. $s.dt$ is the *decay time* associated with a particular *state*. With every arrival of a new event within the scope of state s, $s.dt = s.dt - 1$. Let S be the set of states kept in memory. If $s.dt < 1$, then $S = S \setminus \{s\}$.

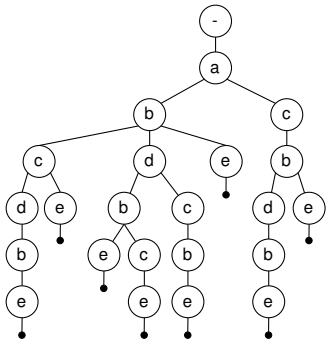

Figure 4: Running example: trie

Definition 3.4 (State buffer). Let S be the set of states associated with a $case \in \mathcal{U}_{case}$, while $\mathcal{P}(S)$ is the powerset of all the sets of states. The *state buffer* B is then a mapping $B : \mathcal{U}_{case} \mapsto \mathcal{P}(S)$.

The state holds the information necessary for the algorithm to compute the conformance. For the running example, let us assume that we have seen the trace $\sigma = \langle a, b, b, c \rangle$. The most recent optimal states s would thus have the current node $n = c$ where the path from the root is abc and $abdbc$, respectively, as this is the model path in the prefix-alignments $\hat{\gamma}$ displayed in Figure 2. The suffix $\hat{\sigma} = \emptyset$ for both states, since $\hat{\gamma}$ contains the latest seen event c and no event currently remains to be processed. The total cost $\delta(\hat{\gamma}) = 1$. The decay time dt value is determined by a hyperparameter, as discussed next.

We distinguish between two modes for initializing dt: **Fixed decay time** denotes a pre-determined integer for each new state. For example, all new states are initialized with $s.dt := 5$. This is effectively a window size parameter. **Discounted decay time** relies on the presumption that deviations near the beginning of a trace are more costly than deviations near the end of a trace [4]. The equation for calculating the discounted decay time is given in Equation 1.

$$Max(\lfloor(\overline{T_{leaf}} - i) * df\rfloor, min_{dt}) \tag{1}$$

The hyperparameters are the *discounting factor* df and a *minimum decay time* min_{dt}. The average length from the root of the trie to each of the leaf nodes is marked by $\overline{T_{leaf}}$, and the current length of the trace is indicated by i as in $\sigma(i)$, where i indicates the i-th event of σ.

To illustrate, the default values set for the algorithm in this paper are $df = 0.3$ and $min_{dt} = 3$. If $\overline{T_{leaf}} = 100$, then for $i = 1$, i.e. the first event of a trace, $dt = Max(\lfloor(100 - 1) * 0.3\rfloor, 3) = 30$. For $i = 50$, $dt = 15$. For $i > 86$, $dt = 3$, as dt will effectively remain at the value set for min_{dt}.

The *State Buffer* is updated with the arrival of every new event *evt* for *case*. The current states of the *caseID* are appended with the new event activity. That is $\forall s \in B(case)\ s.\hat{\sigma} = s.\hat{\sigma} \cup \{act\}$. From each State $s \in S$, the associated costs for adding act are calculated. New states with the least cost are added to the *state buffer*.

Table 4 and Figure 5 show an example. Activities a, b, b, c arrive for a case and the states are calculated based on the trie from Figure 4; a is the first activity for this case, thus the root state with id 0 is added to the *state buffer*; a is a child of the trie's root, so the state with a synchronous move (a, a) is also added to the *state buffer* with state id 1 (step a in Figure 5).

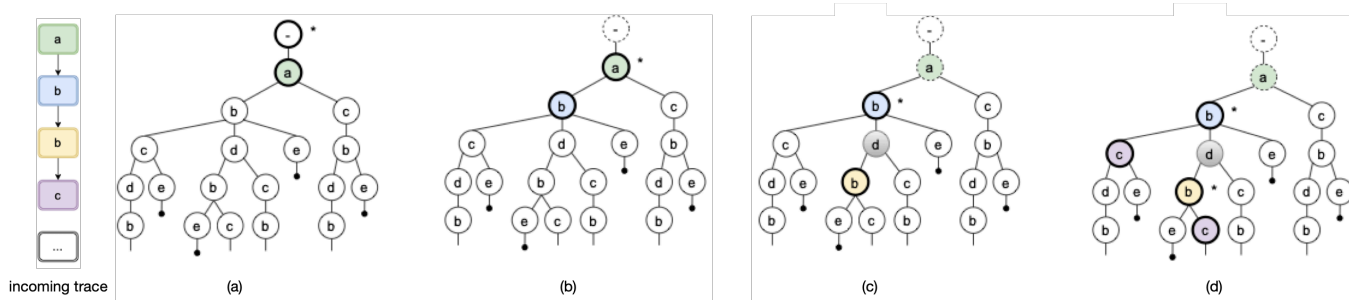

Figure 5: Colored trie nodes follow the color of trace events. Grey trie node points to a model move. Thick black border means the state with alignment ending at this node is still in the state buffer. Dashed border means that the corresponding state has been removed from the buffer. Asterisk to the right of a trie node means it is a member of state with non-empty suffix.

Even though only new states with the least cost are included, preserving a *state buffer* puts strain on the memory, as with each new event arrival, we need to store at least one, but possibly many new states in the buffer. Thus, the *Decay Time* is decremented on each new event arrival for the associated *caseID*.

Lastly, the algorithm includes the *look-ahead limit* for speeding up calculation time in case model moves are needed.

Definition 3.5 (Look-ahead limit). Let $|\hat{\sigma}|$ be the size of the trace suffix, and $s.n.level$ the level of the current state's node in the trie. Then, the *look-ahead limit* $lim = |\hat{\sigma}| + s.n.level + 1$.

The *look-ahead limit* is used for handling model moves, which are more complex in a streaming setting, as the algorithm has to assume the model move is at least as useful as making a log move. To limit a potentially costly traversal, a model move should be realized iff we get a full substring match to $\hat{\sigma}$ in the paths below $s.n$ such that the first matching node is at most at the level lim.

Table 4 shows that, in our running example, when receiving the second b, the state $id = 2$ cannot make a synchronous move, as it is at node $n = b$ where ab is the current path in the trie. $|\hat{\sigma}| = 1$, as this is the second b that is not processed by state $id = 2$. $s.n.level = 2$, as the node is 2 steps from the *root* node. The look-ahead limit for state $id = 2$ is thus $lim = 1 + 2 + 1 = 4$. This indicates that the algorithm should attempt to make model moves iff it gets a substring match on event b at most 4 steps from the *root*. From the Figure 6 it can be deduced that the path $abdb$ is the only viable model move path to get a synchronous move on the second b.

Arriving event	State id	n	$\hat{\gamma}$	$\hat{\sigma}$	$\delta(\hat{\gamma})$	dt
a	0	-	-	$\langle a \rangle$	0	2
	1	a	(a,a)	-	0	2
b	0	-	-	$\langle a, b \rangle$	0	1
	1	a	(a,a)	$\langle b \rangle$	0	1
	2	${}_a b$	(a,a)(b,b)	-	0	2
b	2	${}_a b$	(a,a)(b,b)	$\langle b \rangle$	0	1
	3	${}_a b$	(a,a)(b,b)(b,≫)	-	1	2
	4	${}_{abd} b$	(a,a)(b,b)(≫,d)(b,b)	-	1	2
c	3	${}_a b$	(a,a)(b,b)(b,≫)	$\langle c \rangle$	1	1
	4	${}_{abd} b$	(a,a)(b,b)(≫,d)(b,b)	$\langle c \rangle$	1	1
	5	${}_{ab} c$	(a,a)(b,b)(b,≫)(c,c)	-	1	2
	6	${}_{abd} b$	(a,a)(b,b)(≫,d)(b,b)(c,c)	-	1	2

Table 4: Running example: state buffer. The prefix path of the node is written in subscript. A fixed decay time of 2 is used for a simple example.

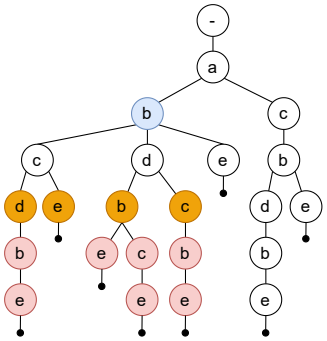

Figure 6: Running example: look-ahead limit for $id = 2$ when the second b arrives. The current node is in blue, the look-ahead limit in orange, and out-of-scope nodes in red.

3.2 Algorithms

The pseudo-code for the IWS algorithm is listed in Algorithm 1. The algorithm takes as input the event *evt* and the trie T.

First, the algorithm initializes some placeholder empty sets of states (Lines 1-3). If the *caseID* is in the state buffer, then the states associated with the *caseID* are fetched and assigned to the set S; otherwise, the initial state with the root node of the trie is added to S (Lines 4-7). In the running example (Table 4), this is when state id 0 is generated.

Then, the algorithm iterates over all the states in the state buffer and attempts to make a synchronous move based on the event activity (Lines 8-9). Utilizing the trie, the synchronous move check is straightforward — the event activity should be a child of the current node of the state. New states are generated for each state where a synchronous move is possible (Line 10). As an example, this occurs for both events a and the first b in the running example.

If no synchronous moves were possible, the states are looped over once more to generate non-synchronous moves and the affiliated

Algorithm 1 I Will Survive

Input: evt, T
1: $S \leftarrow \emptyset$
2: $S_{sync} \leftarrow \emptyset$
3: $S_{nonsync} \leftarrow \emptyset$
4: **if** $evt.case \in B$ **then**
5: $S \leftarrow B(evt.case)$
6: **else**
7: $S \leftarrow \{rootstate\}$
8: **for each** $s \in S$ **do**
9: **if** $s.syncPossible(evt.act)$ **then**
10: $S_{sync} \leftarrow generateSynchronousStates(s, evt.act)$
11: **if** $|S_{sync}| = 0$ **then**
12: **for each** $s \in S$ **do**
13: $S_{nonsync} \leftarrow S_{nonsync} \cup handleLogMove(s, evt.act)$
14: $S_{nonsync} \leftarrow S_{nonsync} \cup handleModelMoves(s, evt.act)$
15: $S_{nonsync} \leftarrow applyCostFilter(S_{nonsync})$
16: $S \leftarrow housekeep(S) \cup S_{sync} \cup S_{nonsync}$
17: $B.S \leftarrow S$
18: *Return* $\hat{\sigma}$

Algorithm 2 Handle Model Moves

Input: s, act
1: $\hat{\sigma}_{check} \leftarrow s.\hat{\sigma} + act$
2: $lim \leftarrow |\hat{\sigma}_{check}| + s.n.level$
3: $S_{model} \leftarrow \emptyset$
4: $N_{children} \leftarrow s.n.children$
5: $N_{matched} \leftarrow \emptyset$
6: **while** $lim > s.n.level$ **do**
7: $N_{matched} \leftarrow matchSubstring(N_{children}, \hat{\sigma}_{check})$
8: **if** $|N_{matched}| > 0$ **then**
9: **break**
10: $lim \leftarrow lim - 1$
11: $N_{children} \leftarrow n.children \; \forall n \in N_{children}$
12: **if** $lim = 0$ *and* $|\hat{\sigma}_{check}| > 1$ **then**
13: $prune(\hat{\sigma}_{check})$
14: $N_{children} \leftarrow s.n.children$
15: **if** $|N_{matched}| > 0$ **then**
16: **for each** $n \in N_{matched}$ **do**
17: $S_{model} \leftarrow constructStates(n)$
18: Return S_{model}

states. This part of the algorithm is explored with the arrival of the second b in the running example. Handling log moves is simple, as the arrived event activity is simply appended as a log move, and no traversal in the trie is necessary (Line 13). The state id 3 is constructed in this phase. Handling model moves (Line 14) is the most complex part of the algorithm, as multiple model moves may be possible and have the same cost. The state id 4 is constructed when executing the handling of model moves. A more detailed description of handling model moves is described in Algorithm 2. Once the non-synchronous moves are generated, a cost filter is applied to keep only the states with the lowest added cost (Line 15). This is most relevant when the decay time is longer, there are many states in the state buffer, and some of the states find more optimal paths than other states.

In the final part, the old states receive housekeeping as the associated decay time is updated, and states that have exhausted the decay time are removed from memory (Line 16). For example, if the algorithm has processed the second b, then states with ids 0 and 1 (Table 4) are removed. An optional limit, defined during the algorithm's initialization, checks if the number of cases in the state buffer is more than allowed; if yes, the case that has not received an update for the longest time is removed from the state buffer. Thereafter, the state buffer is updated with the housekept old states and newly generated states (Line 17). Ultimately, the latest prefix-alignment is returned (Line 18).

The algorithm for model moves (Algorithm 2) expects a state and an activity as input. First, the event activity is appended to the state suffix to ensure that any unprocessed activities are played out, and the result is stored in a variable $\hat{\sigma}_{check}$ (Line 1). Referring to the example in Figure 6 and state id 2 from Table 4, the state suffix is empty when the second b arrives. Thus, $\hat{\sigma}_{check}$ will consist of only the activity b.

An empty set is initialized for holding potential model moves (Line 3), and the look-ahead limit is initialized (Line 2) with the parameters defined in Definition 3.5. For the running example, $lim = 1 + 2 + 1 = 4$, meaning that the algorithm traverses maximally to a distance of 4 from the root node.

Two sets of nodes are initialized – children nodes (Line 4) are used for traversing in the trie, and matched nodes (Line 5) are used for potential storing of nodes that have a substring match. In the running example (Figure 6), the children nodes would be c, d, and e, which are direct children of the node b shown in blue. The matching nodes are initialized as an empty set.

The most exhaustive part of the algorithm is within the while loop (Line 6). All the children nodes are checked for a potential substring match (Line 7) and if matching nodes are found then the while loop is exited (Lines 8-9). Based on the running example, activity b does not match the nodes c, d, or e. Thus, the look-ahead limit is decreased (Line 10) and new children nodes are assigned (Line 11). For the running example, the children nodes are now the nodes depicted in orange in Figure 6. A substring match is found between activity b and the orange node b, and thus the algorithm breaks out of the while loop.

If the look-ahead limit is exhausted, but there is more than one activity in the state suffix (Line 12), then the first element of the suffix is pruned, and the look-ahead limit is reinitialized with the new size. For an example of why these steps are needed, we introduce a different example in Figure 7. Here, a full substring match for activities $\langle c, x, y, z \rangle$ is not possible from node b. However, by removing activity c from $\hat{\sigma}_{check}$, we can get a substring match on $\langle x, y, z \rangle$ by doing a model move on the node q.

Finally, if matching nodes were found (Lines 15-17), then new states are constructed for each matching node by finding possible synchronous moves, model moves, and log moves. For the example in Figure 7, this would mean that our matching node is z, and we traverse by reversing the trace suffix $\langle c, x, y, z \rangle$. Here, z, y, and x are synchronous moves, and then model moves are needed until node b is reached — that is, making a model move on node q. Finally, since activity c was previously pruned in order to get the substring match, the activity is reinstated as a log move.

In the last step, the matching nodes are returned to the main algorithm (Line 18). If no matching nodes were found, then an empty set is returned.

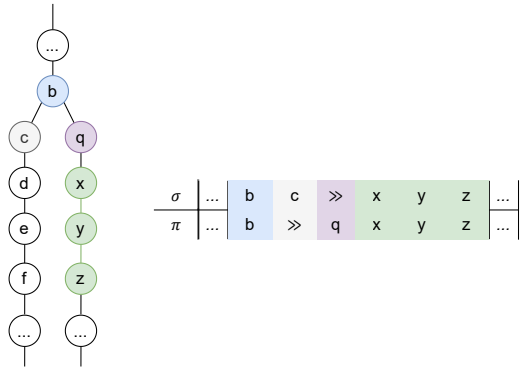

Figure 7: Example use case and resulting alignment for look-ahead limit's suffix pruning. The last synchronous node b is shown in blue; node c is synchronous but leads to a suboptimal path, while nodes x, y, z are more optimal but do not get a substring match if starting from node c.

3.2.1 Space complexity. The trie and the state buffer are the two objects that need to be stored in memory. The trie is static, i.e. it does not change during the execution of the algorithm, while the state buffer is continuously evolving based on the data stream. The trie is in the worst case linear to the size of the proxy log, $O(|L'|)$, indicating that each trace in the proxy log has a unique first activity. Usually, the trie is logarithmic compared to the proxy log size $O(log|L'|)$, because business processes have common prefixes (c.f. Table 3 and Figure 4). As discussed previously, the trie is computed beforehand and it is immutable.

The size of the state buffer depends on two factors, the number of cases $|\mathcal{U}_{case}|$ in the event stream, and the number of states stored for each case. The number of cases stored can be controlled by a simple limiting function that removes the cases that have not received an update for the longest time. The number of states per caseID is dependent on the branching factor (bf) of the trie and the deviation in the behavior between a trace and the trie. In the best case, when an alignment consists of synchronous moves, the state buffer grows as $O(|\mathcal{U}_{case}|.dt)$ that is because, for each newly arriving event, one new state is generated with a synchronous move. In the worst-case scenario, the states' growth can be equal to the bf, of the trie node, i.e. $O(|\mathcal{U}_{case}|.(bf+1).dt)$, we have only one possible log move but bf model moves. Storage of previous states can be controlled by the decay time setting. Using a fixed decay time, there is a fixed upper bound on the number of states stored per case, that can be computed based on the precomputed trie. Using a discounted decay time, the upper bound is still dependent on the trie, while for each individual case, the upper bound diminishes as the case evolves.

3.2.2 Time complexity. For each newly arriving event, we fetch the relevant states in $O(1)$. We retrieve in the worst case $O(|\mathcal{U}_{case}|.(bf+1).dt)$ states, as discussed under space complexity. Synchronous and log moves can be done in $O(1)$. Handling model moves depends on the trie branching factor and the look-ahead limit. The look-ahead limit lim can be bounded by the decay time setting, e.g. the size of the trace suffix can never be longer than the decay time. That is, in the worst case, we need $O(bf.min(dt, lim))$ steps to define the new states to be added to the state buffer.

4 EXPERIMENTS

In this section, we present the experiments that we conducted to validate the proposed approach.

All the executions were done on a single thread using Windows 10 running a CPU @ 1.60GHz, Java 8, and heap size set to 8GB.

The implementation in Java and the execution results are available in a git repository[1].

The preliminaries and experimental results for answering the research questions are discussed in the next subsections.

4.1 Comparative Analysis

In the following, the naming convention from [18] will be used. The current state of the art will be referred to as OCC with two variations: OCC-W1, referring to a window size of 1, and OCC-Winf, with infinite window size. OCC-W1 is the current state of the art in terms of computation time of prefix-alignments. However, due to the window size limitation, the output approximates the optimal alignment. The OCC-Winf provides the baseline for the alignment cost, as the algorithm has an infinite window size and is thus guaranteed to calculate an optimal alignment [26]. The algorithm introduced in this paper will be referred to as IWS (I Will Survive).

Some of the more recent alternative algorithms were excluded from the comparison, as they have not substantially improved the execution time, but have rather focused on memory-handling aspects. Furthermore, a comparison to, for example, the IASR and IAS algorithms introduced in [18] would have been unfair, as the algorithms are only implemented in Python, whereas the OCC algorithms and IWS are implemented in Java. Based on [18], the OCC-W1 implementation in Python outperformed both IASR and IAS time-wise. The algorithm from [28] improves the memory performance of OCC, but it is built on top of the existing OCC by abstracting away a part of the previously calculated prefix-alignment. While the algorithm still outputs the cost of the prefix-alignment, it does not output the complete prefix-alignment itself, rendering a potential analysis of the deviation more obscure. Therefore, it was not considered for comparison with OCC and IWS that both are able to output the entire prefix-alignment.

The comparative analysis aims to examine how IWS fares in terms of alignment cost and computation time. For calculating the computation time, only the time taken to process each event is taken into account. This is done to mimic a streaming scenario, where the loading of a model is done beforehand. The algorithms were executed in an *offline* mode for the experiments. The event log was loaded from a file and fed to the algorithm event by event. This was done to have a fair comparison because the OCC implementation would have needed extensive refactoring, and also, offline mode

[1] https://github.com/DataSystemsGroupUT/ConformanceCheckingUsingTries/tree/streaming

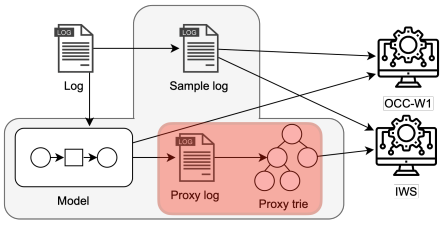

Figure 8: Setup: Gray (red) area shows the artifacts produced by the preprocessing for real-life (synthetic) logs.

allows for avoiding discrepancies from networking or other external factors.

4.1.1 Datasets. Some well-known synthetic and real-life process logs were used for running the experiments. The synthetic process logs[2] also contained a reference WF-net process model. The real-life process logs were BPI 2012[3], BPI 2017[4], and BPI 2020 Travel Permits[5], which do not have an associated reference model.

The OCC takes as input an event and a WF-net model, while IWS requires an event and a trie. Thus, some preprocessing was applied to both the synthetic and real-life datasets. The preprocessing for synthetic data is shown in Figure 8 with the red area indicating the steps done. For OCC, the existing log and model were used. For IWS, a proxy log was simulated from the reference model. The simulation method from [27] was used with default settings of random path simulation, 2000 generated traces, and a maximum looping factor of 3. From the proxy log, the trie was constructed and fed into the IWS algorithm, together with the original log.

For the real-life data, the first step was to construct a process model from the log (Figure 8, shown in gray). For this, the Inductive Miner (IM) [14] plugin in ProM [25] was used with noise thresholds set to 0.2, 0.5, 0.8, and 0.95. For the generation of the trie, the same steps with the same settings were done as for the synthetic data. Finally, while running the experiments, it appeared that the OCC algorithms were unable to output a result due to the size of some of the original logs. Thus, sample logs of the 1000 most frequent trace variants were generated, and the algorithms used the sample logs instead of the original logs.

Information about the logs and models used in the experiments is shown in Table 5. Transitions and τ transitions refer to the WF-net model characteristics. Trie construction time indicates the time taken to construct the underlying trie based on the proxy log – the trie construction is done offline. The number of events indicates how many events are in each sample log.

4.1.2 Results. The results of the experiments are shown in Table 6. The average alignment cost per trace is reported for each dataset. For example, the M1 dataset has 500 traces, and the total alignment costs across the whole dataset were 2918, 2702, and 2439 leading to the average cost per trace of 5.8, 5.4, and 4.9 as described in Table 6. A time per event in milliseconds is reported in terms of computation time.

[2]https://github.com/PADS-UPC/RL-align/tree/master/data/originals/M-models
[3]https://doi.org/10.4121/uuid:3926db30-f712-4394-aebc-75976070e91f
[4]https://doi.org/10.4121/uuid:5f3067df-f10b-45da-b98b-86ae4c7a310b
[5]https://doi.org/10.4121/uuid:52fb97d4-4588-43c9-9d04-3604d4613b51

In terms of the synthetic datasets, in five instances, the OCC algorithms were left running for 1 hour, but no output was produced. These are marked with a - in the table. For other synthetic logs, the cost deviations of IWS were modest. The highest cost deviation was reported for the M9 log, where IWS reported a cost of 1.43x higher than the optimal alignment (29.2 vs 20.5). For M4 and M8,

Dataset	Transitions	τ transitions	Trie nodes	Trie constr. time (ms)	# events
M1	39	3	8281	353	6555
M2	34	2	20742	407	8809
M3	123	14	26434	524	17980
M4	52	8	13261	512	13421
M5	33	1	49571	652	17028
M6	72	2	98781	1239	26719
M7	62	4	46481	812	18803
M8	15	0	1682	326	8246
M9	55	0	7223	576	22163
M10	146	3	87289	1197	29118
BPI 2012-0.2	46	22	3962	1659	33509
BPI 2012-0.5	46	23	1721	387	33509
BPI 2012-0.8	25	8	726	262	33509
BPI 2012-0.95	24	4	106	219	33509
BPI 2017-0.2	46	21	3338	276	32646
BPI 2017-0.5	29	8	192	150	32646
BPI 2017-0.8	32	7	69	163	32646
BPI 2017-0.95	31	6	69	152	32646
BPI 2020-0.2	85	37	23603	397	15810
BPI 2020-0.5	64	15	945	71	15810
BPI 2020-0.8	65	16	887	64	15810
BPI 2020-0.95	59	14	1591	235	15810

Table 5: Dataset metadata

Dataset	Cost per trace			Time per event (ms)		
	IWS	OCC-W1	OCC-Winf	IWS	OCC-W1	OCC-Winf
M1	5.8	5.4	4.9	0.3	1.3	1.9
M2	10.6	9.4	8.1	0.2	6.3	12.2
M3	23.9	-	-	0.6	-	-
M4	22.1	23.0	20.5	0.8	10.9	25.4
M5	26.0	-	-	0.8	-	-
M6	45.9	-	-	12.4	-	-
M7	29.2	-	-	0.6	-	-
M8	7.6	7.8	6.7	0.2	0.7	1.1
M9	29.2	26.1	20.5	0.7	19.6	32.8
M10	50.0	-	-	9.0	-	-
BPI 2012-0.2	27.1	0.8	0.3	0.3	2.0	3.8
BPI 2012-0.5	26.6	6.3	3.0	0.3	2.4	12.4
BPI 2012-0.8	28.3	26.1	16.8	0.3	1.2	5.3
BPI 2012-0.95	30.1	30.1	26.6	0.2	5.2	9.0
BPI 2017-0.2	26.1	4.4	1.7	0.3	3.5	9.9
BPI 2017-0.5	25.3	26.7	25.1	0.2	3.3	14.0
BPI 2017-0.8	28.6	29.0	25.4	0.2	3.2	10.7
BPI 2017-0.95	28.6	29.0	25.4	0.2	2.3	11.4
BPI 2020-0.2	12.1	6.5	5.2	0.3	4.6	7.3
BPI 2020-0.5	10.7	10.8	6.8	0.1	1.6	5.1
BPI 2020-0.8	10.3	11.2	8.7	0.1	1.8	7.6
BPI 2020-0.95	12.1	7.6	6.8	0.2	1.3	3.6

Table 6: Comparative analysis results.

IWS alignment	A	B	P	P	D	E	P	E	L	K	O	G	H	I	E	I	M	F	I
	A	B	P	P	≫	E	P	≫	≫	≫	≫	≫	≫	≫	≫	≫	≫	≫	≫

OCC-W1 alignment	A	B	P	≫	≫	P	≫	≫	D	≫	E	≫	P	≫	E	≫	L	≫	≫	K	O	≫	≫	G	H	I	≫	≫	≫	E	≫	≫	≫	≫	≫	≫	≫	I	≫	≫	≫	M	≫	≫	F	I
	A	B	P	τ	τ	P	τ	τ	D	τ	E	τ	P	τ	E	τ	L	τ	τ	K	O	τ	τ	G	H	I	τ	τ	τ	E	τ	τ	τ	τ	τ	τ	τ	I	τ	τ	τ	M	τ	τ	≫	I

Table 7: Example of a trace's prefix-alignment from BPI 2012 log with IM 0.2 threshold.

IWS outperformed OCC-W1 in terms of cost, and only had an error rate of 1.08x and 1.14x compared to optimal alignments.

In terms of execution time, the results are strongly in favor of IWS. In almost all cases, IWS was able to process an event in less than one millisecond, while OCC-W1 was able to process an event in less than a millisecond only for the dataset M8. The M8 dataset also exhibited the smallest difference in terms of execution time, as IWS finished execution 3.5x faster than OCC-W1. For M2, M4, and M9 datasets, IWS computed the result in only a fraction of the time compared to the OCC-W1 execution time. This is especially interesting in terms of dataset M4, where IWS outperformed OCC-W1 by both – producing alignments that were more optimal and finding these alignments more than 14.4x faster. For the datasets M3, M5, and M7, IWS computed alignments in 10-14 seconds, while the OCC algorithms were unable to output a result within 1 hour. This indicates an execution time difference of potentially more than three orders of magnitude.

Looking at the real-life datasets, the results were a bit more varied in terms of cost. IWS displayed an inferior cost performance for the BPI 2012 dataset, where the model was discovered with an IM threshold setting of 0.2. Here, IWS reported a cost 35.9x higher than OCC-W1 and 81.3x higher than optimal alignments, which would indicate an unreasonably high error. The reason for such poor performance is discussed in the next section. Some other poor results were for BPI 2012 with IM 0.5, and BPI 2017 with IM 0.2. However, for more than half of the datasets, the cost error was comparable between IWS and OCC-W1. In fact, for five datasets, IWS outperformed OCC-W1 by producing more optimal alignments; for BPI 2017 with IM 0.5, the cost difference between IWS and optimal alignments was only 1.01x.

In terms of execution time, IWS is the fastest across all datasets. The smallest difference is for BPI 2012 with IM setting 0.8, where IWS finished 4.6x faster than OCC-W1 and 19.5x faster than optimal alignments. The biggest execution time difference was for BPI 2012, with IM setting 0.95. Interestingly, for this dataset, both IWS and OCC-W1 had the same alignment cost, but IWS could compute the alignments 24.5x faster than OCC-W1.

4.1.3 Discussion. The results indicate that IWS is, in most cases, better suited for streaming conformance checking than the current state of the art (OCC-W1). IWS beats OCC-W1 in computation time and, in some cases, also in terms of finding more optimal prefix-alignments. However, there were a few cases in the real-life datasets where the cost error of IWS was very high compared to the state of the art. Let us investigate the reason for this with some examples from the BPI 2012 log.

First of all, the Inductive Miner, used for discovering the WF-net models, has a tendency to discover *flower models*[14], i.e. models allowing any kind of behavior, if the noise threshold is set to a low level. In such cases, the set of allowed behavior in the model is very large. IWS is dependent on the existence of a trie, which in turn is dependent on the existence of a proxy log – a set of behavior *extracted* from the model. The more behavior the model allows for, the more difficult it is to extract a representative proxy log.

One way to have a more representative proxy log would be to increase the size of the proxy log. However, for *flower models*, this may be infeasible. The WF-net model produced by IM for the BPI 2012 log with a 0.2 setting has 46 transitions, out of which 24 transitions are labeled and 22 are silent. The first two labeled transitions are fixed, but after that, due to the τ transitions, almost any of the 22 labeled transitions can occur. Due to loops, the following transition can be any of the 22 labeled transitions, including the label itself. Thus, with each new event, the possible behavior increases exponentially. For a trace with ten events, assuming the first two events are always sequential, there can be $2 + 22^8 = 54875873538$ possible variants. The BPI 2012 log has, on average, 33 events per trace. A simulation method to extract a proxy log that would not find deviations becomes impractical from a computational point of view. Furthermore, it can be argued that exercising conformance checking on a process model that allows any behavior has no intrinsic value since any kind of behavior is conforming.

An example prefix-alignment of the BPI 2012 log is shown in Table 7. The prefix alignment from the IWS algorithm is much shorter because the trie does not contain τ transitions. The OCC-W1 prefix alignment has many model moves on τ transitions, allowing it to find synchronous moves for almost every event in the trace. By convention, the τ transitions are not penalized and have an alignment cost of 0, as they are valid passages through the model. However, it can be argued that while the OCC-W1 alignment has a much lower cost – it has a cost of 1, compared to the IWS cost of 13 – the alignment itself becomes hard to decipher due to the many τ transitions and is hardly usable for an analyst trying to pinpoint deviations. Thus, such an alignment provides little value in a real-life setting. For reference, the worst cost error in the BPI 2012 log is for a trace with 170 events. IWS reports a cost of 164, while OCC-W1 reports a cost of 2. However, in the prefix alignment, OCC-W1 has 537 moves on τ transitions. In total, the output from

Model type	Small	Medium	Large
Activities	16	153	471
Traces in proxy log	256	23409	25000
Trie nodes	841	713640	760343
Trie building time	1267	19934	101960
Computation time	833438	1244394	1541570
Idle time	2766562	2355606	2058430
Events	278063	365425	354602
Event Computation time	3.0	3.4	4.3

Table 8: Stress test: description of models and results. All times are expressed in milliseconds (ms).

OCC-W1 across the whole BPI 2012 log for the IM 0.2 model has 76172 τ transitions across the 1000 sample traces.

In conclusion, IWS outperforms OCC-W1 in terms of computation time and is comparable or better in terms of cost error in most cases. If the model allows for a large variety of behavior, e.g., when dealing with *flower models*, then the cost difference between IWS and OCC-W1 is prominent, and for such models, OCC-W1 is better suited, especially if calculation time is not a constraint. Importantly, however, it can be argued that the alignments produced by the OCC-W1 for *flower models* are complex to grasp. Furthermore, computing conformance for models which allow any kind of behavior does not seem practical, as it is counterintuitive to the purpose of conformance checking.

4.2 Stress test

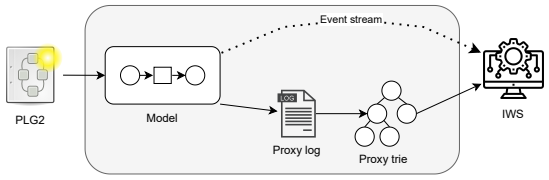

Figure 9: Setup for stress testing. The gray rectangle shows the artifacts produced by the PLG2 software.

In order to test the algorithm for speed and memory consumption, the PLG2 software [6] was used to simulate three process models of various sizes. PLG2 was further used to simulate a proxy log from these models and to stream events to the socket while adding some noise to the event stream so as to mimic discrepant behavior. An overview of the approach can be seen in Figure 9. The memory consumption was measured using the tool VisualVM [19].

A description of the process models generated by PLG2 and the resulting tries is in Table 8. The number of unique activities illustrates the complexity level of each of the models. The proxy log sizes were determined by squaring the activity count, except for the *Large* model, where the simulation was stopped at 25000 traces due to the long execution time of the simulation process.

The event streams were generated with the preset configuration of having noise only on the control-flow, indicating that traces had a 5% chance of missing activities. The streams were left running for an hour, with IWS set to forget cases when the number of cases in memory is over 10000.

The algorithm was successfully able to keep up with the stream. For the event stream on the small model, the algorithm was idle for $\frac{3}{4}$ of the time, signaling that higher throughput could have been achieved. For medium and large models, the algorithm was idle for $\frac{2}{3}$ and $\frac{4}{7}$ of the time, respectively.

As expected, the experiments using the small model had the least strain on the CPU and memory, with less than 5% of CPU utilized by the Java application running the algorithm and memory usage not exceeding 200 MB during the execution. The results for the small model are depicted in Figure 10. The figure includes a run on the same model with different algorithm and stream settings – a higher noise level, indicating more discrepant behavior, and higher decay time settings, forcing the algorithm to store more states in memory. As can be seen empirically, the number of states and the memory usage stabilizes and remains bounded due to the case limit and the usage of decay time in the state buffer, while for the higher noise and decay time, the sheer amount of states kept in memory is higher.

Memory usage for medium and large models was much higher – likely due to a higher amount of nodes in the medium and large tries and longer traces, which lead to more states being kept in memory. For the medium model, the memory usage was between 600-1600 MB, stabilizing at 900 MB with a slightly increasing upward trend, while for the large model, the memory usage was between 1500-2250 MB. The accompanying figures can be found in the repository due to the space limitation of this article.

The CPU and memory usage results show that the IWS is usable for online conformance checking for an extended period of time. Notably, the current experiments incorporated only a simple case bound to remove processed traces from memory. For more complex options for handling the memory and limiting the cases, we refer to [28]. Such case and state management techniques can be used to extend the IWS algorithm. It would also be possible to set rules that define when memory should be flushed to disk.

Ultimately, the algorithm displays very fast processing of event streams with a low strain on memory. Thus, the algorithm would be applicable for real-life streaming conformance checking.

5 CONCLUSION

This paper presents a new approximate approach (IWS) for online conformance checking based on prefix alignments. IWS uses a trie as the underlying structure for holding the model behavior as a compacted proxy log. A *State buffer* is used as a way to keep track of seen traces, *decay time* is used for releasing states from the buffer, and *look-ahead limit* is used for optimizing possible model moves.

We compared IWS against the state-of-the-art solutions (OCC-W*) using synthetic and real-life datasets. IWS outperformed OCC in all instances in terms of computation time. In some cases, IWS produced an output in 8-13 seconds, while OCC-W1 failed to finish within an hour. At the same time, the IWS achieved comparable cost error for over a quarter of the datasets and even achieved a lower error cost than OCC-W1. IWS showed high error for process models with low precision – *flower models*. However, the alignments produced by OCC-W1 for such models are hard to decipher. Furthermore, conformance checking on models allowing any behavior is arguably not sensible.

The algorithm seems suitable for streaming conformance checking, but it has some limitations. For example, the trie can be exponentially large depending on the specific process and the size of the proxy log. Moreover, there is a high dependence on the quality of the proxy log. While discovery algorithms can generalize on the constructs such as parallelism and loops, a trie is merely a one-to-one representation of the proxy log, potentially overfitting the model behavior. Therefore, we plan to investigate generating the trie directly from an existing process model for future work. Furthermore, current research in streaming conformance checking has not touched upon the possibility of stream imperfections. In real-life settings, imperfections such as out-of-order events are likely to occur. IWS, utilizing a discounted Decay Time, could be a solution

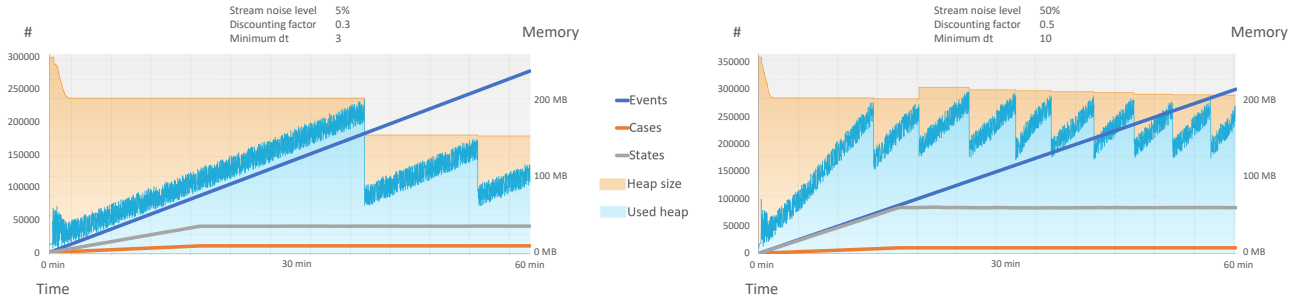

Figure 10: Memory consumption, number of events, states, and cases within one hour of stress test.

to handle stream imperfections. Finally, we plan to leverage Stream Processing engines, e.g., Beamline [6] built on Apache Flink, for improving IWS even further and allowing the conformance checking to utilize parallelism and to occur on distributed systems.

REFERENCES

[1] Arya Adriansyah, Boudewijn F Van Dongen, and Nicola Zannone. 2013. Controlling break-the-glass through alignment. In *2013 International Conference on Social Computing*. IEEE, 606–611.

[2] Ahmed Awad, Kristo Raun, and Matthias Weidlich. 2021. Efficient Approximate Conformance Checking Using Trie Data Structures. In *2021 3rd International Conference on Process Mining (ICPM)*. IEEE, 1–8.

[3] Vincent Bloemen, Sebastiaan J van Zelst, Wil MP van der Aalst, Boudewijn F van Dongen, and Jaco van de Pol. 2018. Maximizing synchronization for aligning observed and modelled behaviour. In *BPM*. Springer, 233–249.

[4] Mathilde Boltenhagen, Thomas Chatain, and Josep Carmona. 2021. A discounted cost function for fast alignments of business processes. In *International Conference on Business Process Management*. Springer, 252–269.

[5] Mathilde Boltenhagen, Thomas Chatain, and Josep Carmona. 2021. Optimized SAT encoding of conformance checking artefacts. *Computing* 103, 1 (2021), 29–50.

[6] Andrea Burattin. 2016. PLG2: Multiperspective Process Randomization with Online and Offline Simulations.. In *BPM (Demos) (CEUR Workshop Proceedings, Vol. 1789)*. CEUR-WS.org, 1–6.

[7] Andrea Burattin and Josep Carmona. 2017. A framework for online conformance checking. In *International Conference on Business Process Management*. Springer, 165–177.

[8] Andrea Burattin, Sebastiaan J van Zelst, Abel Armas-Cervantes, Boudewijn F van Dongen, and Josep Carmona. 2018. Online conformance checking using behavioural patterns. In *International Conference on Business Process Management*. Springer, 250–267.

[9] Josep Carmona, Boudewijn F. van Dongen, Andreas Solti, and Matthias Weidlich. 2018. *Conformance Checking - Relating Processes and Models*. Springer.

[10] Thomas Chatain and Josep Carmona. 2016. Anti-alignments in conformance checking–the dark side of process models. In *Application and Theory of Petri Nets and Concurrency*. Springer, 240–258.

[11] Michael Daum, Manuel Götz, and Jörg Domaschka. 2012. Integrating CEP and BPM: how CEP realizes functional requirements of BPM applications (industry article). In *DEBS*. ACM, 157–166. https://doi.org/10.1145/2335484.2335503

[12] Mohammadreza Fani Sani, Sebastiaan J van Zelst, and Wil MP van der Aalst. 2020. Conformance checking approximation using subset selection and edit distance. In *International Conference on Advanced Information Systems Engineering*. Springer, 234–251.

[13] Wai Lam Jonathan Lee, Andrea Burattin, Jorge Munoz-Gama, and Marcos Sepúlveda. 2021. Orientation and conformance: A HMM-based approach to online conformance checking. *Information Systems* 102 (2021), 101674.

[14] Sander JJ Leemans, Dirk Fahland, and Wil MP Van Der Aalst. 2013. Discovering block-structured process models from event logs containing infrequent behaviour. In *BPM*. Springer, 66–78.

[15] Tian Li and Sebastiaan J van Zelst. 2022. Cache Enhanced Split-Point-Based Alignment Calculation. In *ICPM*. IEEE, 120–127.

[16] Jesus Ojeda. 2023. Conformance checking artefacts through weighted partial MaxSAT. *Information Systems* (2023), 102168.

[17] Bijan Rad, Fei Song, Vincent Jacob, and Yanlei Diao. 2021. Explainable anomaly detection on high-dimensional time series data. In *Proceedings of the 15th ACM International Conference on Distributed and Event-based Systems*. ACM, 2–14.

[18] Daniel Schuster and Sebastiaan J van Zelst. 2020. Online process monitoring using incremental state-space expansion: an exact algorithm. In *BPM*. Springer, 147–164.

[19] J Sedlacek and T Hurka. 2017. VisualVM All-in-One Java Troubleshooting Tool.

[20] Wil van der Aalst, Arya Adriansyah, and Boudewijn van Dongen. 2012. Replaying history on process models for conformance checking and performance analysis. *WIREs Data Mining and Knowledge Discovery* 2, 2 (March 2012), 182–192. https://doi.org/10.1002/widm.1045

[21] Wil MP van der Aalst. 2022. Process mining: a 360 degree overview. In *Process Mining Handbook*. Springer, 3–34.

[22] Wil M. P. van der Aalst. 1997. Verification of Workflow Nets. In *ICATPN (LNCS, Vol. 1248)*. Springer, 407–426. https://doi.org/10.1007/3-540-63139-9_48

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

[24] Wil M. P. van der Aalst. 2019. Everything You Always Wanted to Know About Petri Nets, but Were Afraid to Ask. In *BPM* (Vienna, Austria). Springer-Verlag, Berlin, Heidelberg, 3–9. https://doi.org/10.1007/978-3-030-26619-6_1

[25] Boudewijn F Van Dongen, Ana Karla A de Medeiros, HMW Verbeek, AJMM Weijters, and Wil MP van Der Aalst. 2005. The ProM framework: A new era in process mining tool support. In *ICATPN*. Springer, 444–454.

[26] Sebastiaan J van Zelst, Alfredo Bolt, Marwan Hassani, Boudewijn F van Dongen, and Wil MP van der Aalst. 2019. Online conformance checking: relating event streams to process models using prefix-alignments. *International Journal of Data Science and Analytics* 8, 3 (2019), 269–284.

[27] Seppe Vanden Broucke, Jochen De Weerdt, Jan Vanthienen, and Bart Baesens. 2012. *An improved process event log artificial negative event generator*. Technical Report. Department of Decision Sciences and Information Management, KU Leuven, Belgium.

[28] Rashid Zaman, Marwan Hassani, and Boudewijn F. van Dongen. 2021. A Framework for Efficient Memory Utilization in Online Conformance Checking. https://doi.org/10.48550/ARXIV.2112.13640

[6] https://www.beamline.cloud