# Maximizing Synchronization for Aligning Observed and Modelled Behaviour

Vincent Bloemen[1]*, Sebastiaan van Zelst[2], Wil van der Aalst[3], Boudewijn van Dongen[2], and Jaco van de Pol[1]

[1] University of Twente, Enschede, The Netherlands
[2] Eindhoven University of Technology, Eindhoven, The Netherlands
[3] RWTH Aachen University, Aachen, Germany

**Abstract.** Conformance checking is a branch of process mining that aims to assess to what degree event data originating from the execution of a (business) process and a corresponding reference model conform to each other. Alignments have been recently introduced as a solution for conformance checking and have since rapidly developed into becoming the de facto standard.
The state-of-the-art method to compute alignments is based on solving a shortest path problem derived from the reference model and the event data. Within such a shortest path problem, a cost function is used to guide the search to an optimal solution. The standard cost-function treats mismatches in the model and log as equal. In this paper, we consider a variant of this standard cost function which maximizes the number of correct matches instead. We study the effects of using this cost-function compared to the standard cost function on both small and large models using over a thousand generated and industrial case studies.
We further show that the alignment computation process can be sped up significantly in specific instances. Finally, we present a new algorithm for the computation of alignments on models with many log traces that is an order of magnitude faster (in maximizing synchronous moves) compared to the state-of-the-art A* based solution method, as a result of a preprocessing step on the model.

## 1 Introduction

Process mining [1] is a field of study involved with the *discovery*, *conformance checking*, and *enhancement* of processes, using event data recorded during process execution. In process discovery, we aim to discover process models based on traces of executed event data. In conformance checking, we assess to what degree a process model (potentially discovered) is in line with recorded event data. Finally, in process enhancement, we aim at improving or extending the process based on facts derived from event data.

Modern information systems allow us to track, often in great detail, the behaviour of the process it supports. Moreover, instrumentation and/or program tracing tools allow us to track the behavioural profile of the execution of

* This work is supported by the 3TU.BSR project.

enterprise-level software systems [2,3]. Such behavioural data is often referred to as an event log, which can be seen as a multiset of log traces, i.e. sequences of observed events in the system. However, it is often the case, due to noise or under/over-specification, that the observed behaviour does not conform to a valid process instance, i.e., it deviates from its intended behaviour as specified by its reference model.

Conformance checking assesses to what degree the event log and model conform to each other. Early conformance checking techniques [4] are based on simple heuristics and therefore, may yield ambiguous/unpredictable results.

*Alignments* [5,6] were introduced to overcome the limitations of early conformance checking techniques. Alignments map observed behaviour onto behaviour described by the process model. As such, we identify four types of relations between the model and event log in an alignment:

1. A *log move*, in which we are unable to map an observed event, recorded in the event log, onto the reference model.
2. A *model move*, in which an action is described by the reference model, yet this is not reflected in the event log.
3. A *synchronous move*, in which we are able to map an event, observed in the event log, to a corresponding action described by the reference model.
4. A *silent move*, in which the model performs a silent or invisible action (denoted with $\tau$).

Consider the example model of a simple file reading system given in Fig. 1 and the trace $\sigma = \langle A, D, B, D \rangle$. An alignment for the model and $\sigma$ is given by $\gamma^0$ (top right in Fig. 1). Here, the upper-part depicts the trace and the bottom-part depicts an execution path described by the model, starting at state $\mathrm{p}_0$ and ending at state $\mathrm{p}_5$. The first pair, $|\frac{A}{A}|$, represents a synchronous move, in which both the log and the path in the model describe the execution of an $A$ activity. The next pair, $|\frac{D}{\gg}|$, is a log move where the log trace describes the execution of a $D$ activity that is not mapped to a model move. The *skip* ($\gg$) symbol is used to represent such a mismatch. Observe that the model remains in the same state. This is continued by a model move in which the model executes a $C$ activity, which is not recorded in the trace, i.e., $|\frac{\gg}{C}|$. Finally, the alignment ends with two synchronous moves.

An optimal alignment is an alignment that minimizes a given cost function. Typically, each type of move gets a value assigned $\mathbb{R}_{\geq 0}$. The cost of an alignment is simply the sum of the costs of its individual moves. The most common way to do this is to assign a cost of 1 to both model and log moves and 0 to synchronous and silent moves. In practice, the A* shortest path algorithm [7] is often used for computing optimal alignments.

We argue that the standard cost function is not always the best-suited function for optimal alignments. Consider the model from Fig. 1 again, with the trace $\sigma' = \langle B \rangle$. An optimal alignment using the standard cost function would result in $\gamma^1$. Considering that event $B$ is observed behaviour, i.e., the system logged "parse file", it seems illogical to map this behaviour with a path in the model indicating that the file was not found. In case we set up the cost function such

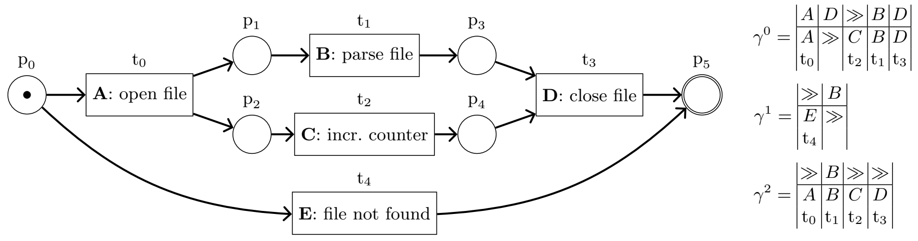

p0
t0
A: open file
p1
p2
t1
B: parse file
t2
C: incr. counter
p3
p4
t3
D: close file
p5
t4
E: file not found
$\gamma^0 =$ A D ≫ B D / A ≫ C B D / t0 t2 t1 t3
$\gamma^1 =$ ≫ B / E ≫ / t4
$\gamma^2 =$ ≫ B ≫ ≫ / A B C D / t0 t1 t2 t3

Fig. 1: Example process model (in Petri net formalism) for a simple file reading system and an alignment for the trace $\sigma = \langle A, D, B, D\rangle$ ($\gamma^0$). For the trace $\sigma = \langle B\rangle$, two optimal alignments are given using the standard- ($\gamma^1$) and variant ($\gamma^2$) cost functions.

that the number of synchronous moves are maximized, an optimal alignment would result in $\gamma^2$. Arguably, a more likely scenario is that not all parts of the program produced log output and $\gamma^2$ would be preferred.

Motivated by the example shown in Fig. 1, we consider the applicability of a cost function that maximizes the number of synchronous moves in a more general setting and study its effects. Our contributions are as follows.

- We formalise the relation between the event log and the reference model to distinguish different cases of alignment problems. We show how the cost functions affect the resulting alignments for these cases. We further show that when the reference model is an abstraction of the event log, the alignment computation process can be significantly improved.
- We study the differences in alignments and their computation times on over a thousand large instances that exhibit various characteristics. We also compare the results from the A* algorithm with a recent symbolic algorithm [8].
- We present a new algorithm for computing alignments that exploits our new cost function in a preprocessing step. Using a set of industrial models, we show that it performs an order of magnitude faster than the A* algorithm.

The remainder of this paper is structured as follows. Section 2 introduces preliminaries. Then, in Section 3 and Section 4 we introduce the synchronous cost function and formalise the relation between the event log and the reference model. We discuss existing algorithms for computing alignments in Section 5. In Section 6 we present the new algorithm that preprocesses the model to improve the alignment computation process. Experiments are presented in Section 7. Section 8 discusses related work. Section 9 concludes the paper.

## 2 Preliminaries

We assume that the reader is familiar with the basics of automata theory and Petri nets. We denote a trace or sequence by $\sigma = \langle \sigma_0, \sigma_1, \ldots, \sigma_{|\sigma|-1}\rangle$, two sequences are concatenated using the $\cdot$ operation. Given a sequence $\sigma$ and a set of elements $S$, we refer to $\sigma \setminus S$ as the sequence without any elements from $S$, e.g., $\langle a, b, b, c, a, f\rangle \setminus \{b, f\} = \langle a, c, a\rangle$. For two sequences $\sigma_1$ and $\sigma_2$, we call $\sigma_1$

a *subsequence* of $\sigma_2$ (denoted with $\sigma_1 \sqsubseteq \sigma_2$) if $\sigma_1$ is formed from $\sigma_2$ by deleting elements from $\sigma_2$ without changing its order, e.g., $\langle c, a, t \rangle \sqsubseteq \langle a, c, r, a, t, e \rangle$. Similarly, $\sigma_1 \sqsubset \sigma_2$ implies that $\sigma_1$ is a strict subsequence of $\sigma_2$, thus $\sigma_1 \neq \sigma_2$.

*Traces* are sequences $\sigma \in \Sigma^*$, for which each element is called an *event* and is contained in the alphabet $\Sigma$, also called the set of events. We globally define the alphabet $\Sigma$, which does not contain the skip event ($\gg$) nor the invisible action or silent event ($\tau$). Given a set $S$, we denote the set of all possible multisets as $\mathcal{B}(S)$, and its power-set by $2^S$. An *event log* $\mathrm{E}$ is a multiset of traces, i.e., $\mathrm{E} \subseteq \mathcal{B}(\Sigma^*)$.

## 2.1 Preliminaries on Petri Nets

*Petri nets* are a mathematical formalism that allow us to describe processes, typically containing parallel behaviour, in a compact manner. Consider Fig. 1 which is a simple example of a Petri net. The Petri net consists of *places*, visualized as circles, that allow us to express the state (or *marking*) of the Petri net. Furthermore, it consists of *transitions*, visualized as boxes, that allow us to manipulate the state of the Petri net. We are never able to connect a place with another place nor a transition with another transition. Thus, from a graph-theoretical perspective, a Petri net is a bipartite graph.

**Definition 1 (Petri net, marking).** *A* Petri net *is defined as a tuple* $N = (P, T, F, \Sigma_\tau, \lambda, m_0, m_\mathrm{F})$ *such that:*
- $P$ *is a finite set of* places*,*
- $T$ *is a finite set of* transitions *such that* $P \cap T = \emptyset$*,*
- $F \subseteq (P \times T) \cup (T \times P)$ *is a set of directed arcs, called the* flow relation*,*
- $\Sigma_\tau$ *is a set of activity events, with* $\Sigma_\tau = \Sigma \cup \{\tau\}$*,*
- $\lambda : T \to \Sigma_\tau$ *is a* labelling function *for each transition,*
- $m_0 \in \mathcal{B}(P)$ *is the initial* marking *of the Petri net,*
- $m_\mathrm{F} \in \mathcal{B}(P)$ *is the final marking of the Petri net.*

*A* marking *is defined as a multiset of places, denoting where tokens reside in the Petri net. A transition* $t \in T$ *can be* fired *if, according to the flow relation, all places directing to* $t$ *contain a token. After firing a transition, the tokens are removed from these places and all places having an incoming arc from* $t$ *receive a token. It may be possible for a place to contain more than one token.*

**Definition 2 (Marking graph).** *For a Petri net* $N = (P, T, F, \Sigma_\tau, \lambda, m_0, m_\mathrm{F})$*, the corresponding* marking graph *or* state-space $MG = (Q, \Sigma_\tau, \delta, q_0, q_\mathrm{F})$ *is a non-deterministic automaton such that:*
- $Q \subseteq \mathcal{B}(P)$ *is the (possibly infinite) set of vertices in* $MG$*, which corresponds to the set of* reachable *markings from* $m_0$ *(obtained from firing transitions),*
- $\delta \subseteq (Q \times T \times Q)$ *is the set of edges in* $MG$*, i.e.,* $(m, t, m') \in \delta$ *iff there is a* $t \in T$ *such that* $m'$ *is obtained from firing transition* $t$ *from marking* $m$*.*
- $q_0 = m_0$ *is the initial state of the graph,*
- $q_\mathrm{F} = m_\mathrm{F}$ *is the final state of the graph.*

*For an edge* $e = (m, t, m') \in \delta$*, we write* $\lambda(e)$ *to denote* $\lambda(t)$ *and use the notation* $m \xrightarrow{a} m'$ *to represent the edge* $e$ *for which* $\lambda(e) = a$ *(we assume that for two edges*

*$(m, t_1, m') \in \delta$ and $(m, t_2, m') \in \delta$, if $\lambda(t_1) = \lambda(t_2)$ then $t_1 = t_2$). The source and target markings of edge $e$ are respectively denoted by $\texttt{src}(e)$ and $\texttt{tar}(e)$.*

**Definition 3 (Path, language).** *Given a Petri net $N$ and corresponding marking graph $MG = (Q, \Sigma_\tau, \delta, q_0, q_\mathrm{F})$, a sequence of edges $P = \langle P_0, P_1, \ldots, P_n \rangle \in \delta^*$ is called a* path *in $N$ if it forms a path on the marking graph of $N$: $\texttt{src}(P_0) = m_0 \wedge \texttt{tar}(P_n) = m_\mathrm{F} \wedge \forall_{0 \leq i < n} : \texttt{tar}(P_i) = \texttt{src}(P_{i+1})$. The set of all paths in $N$ is denoted by $Paths(N)$. With $\lambda(P)$ we refer to the sequence of labels visited in $P$, i.e., $\lambda(P) = \langle \lambda(P_0), \lambda(P_1), \ldots, \lambda(P_n) \rangle$ (there may be different paths $P$ and $P'$ such that $\lambda(P) = \lambda(P')$). We define the* language *$\mathcal{L}$ of a Petri net $N$ by $\mathcal{L}(N) = \{\lambda(P) \mid P \in Paths(N)\}$.*

**Definition 4 (Trace to Petri net).** *Given a trace $\sigma = \langle \sigma_1, \sigma_2, \ldots, \sigma_n \rangle \in \Sigma^*$, its corresponding Petri net is defined as $N_\sigma = (P, T, F, \Sigma_\tau, \lambda, m_0, m_\mathrm{F})$ with $P = \{p_0, p_1, \ldots, p_n, p_{n+1}\}$, $T = \{t_0, t_1, \ldots, t_n\}$, $F = \{(p_0, t_0), (p_1, t_1), \ldots, (p_n, t_n)\} \cup \{(t_0, p_1), (t_1, p_2), \ldots, (t_n, p_{n+1})\}$, $\Sigma_\tau = \bigcup_{0 \leq i < n} \{\sigma_i\}$, $\forall_{0 \leq i < n} : \lambda(t_i) = \sigma_i$, $m_0 = p_0$, and $m_\mathrm{F} = p_{n+1}$.*

## 2.2 Preliminaries on Alignments

**Definition 5 (Alignment).** *Let $\sigma \in \Sigma^*$ be a log trace and let $N$ be a Petri net model, for which we obtain the marking graph $MG = (Q, \Sigma_\tau, \delta, q_0, q_\mathrm{F})$. We refer to $\Sigma_\gg$ as the alphabet containing skips: $\Sigma_\gg = \Sigma \cup \{\gg\}$ and $\Sigma_{\tau\gg}$ as the alphabet that also contains the silent event: $\Sigma_{\tau\gg} = \Sigma \cup \{\gg, \tau\}$. Let $\gamma \in (\Sigma_\gg \times \Sigma_{\tau\gg})^*$ be a sequence of log-model pairs (note that $\tau$ steps are only possible in the model). For $\gamma = \langle (\gamma^0_0, \gamma^1_0), (\gamma^0_1, \gamma^1_1), \ldots, (\gamma^0_{|\gamma|-1}, \gamma^1_{|\gamma|-1}) \rangle$, we define $\gamma^\ell$ as $\gamma^\ell = \langle \gamma^0_0, \gamma^0_1, \ldots, \gamma^0_{|\gamma|-1} \rangle \setminus \{\gg\}$ and $\gamma^m$ by $\gamma^m = \langle \gamma^1_0, \gamma^1_1, \ldots, \gamma^1_{|\gamma|-1} \rangle \setminus \{\gg\}$. We call $\gamma$ an* alignment *if the following conditions hold:*

1. *$\gamma^\ell = \sigma$ (the activities of the log-part, equals to $\sigma$),*
2. *$\gamma^m \in \mathcal{L}(N)$ ($\gamma^m$ forms a path in $N$),*
3. *$\forall a, b \in \Sigma : a \neq b \Rightarrow (a, b) \notin \gamma$ (illegal moves),*
4. *$(\gg, \gg) \notin \gamma$, (the 'empty' move may not exist in $\gamma$).*

**Definition 6 (Alignment cost).** *Let $\gamma \in (\Sigma_\gg \times \Sigma_{\tau\gg})^*$ be an alignment for $\sigma \in \Sigma^*$ and the Petri net $N$. The cost function $c$ for pairs of $\gamma$ is given as follows; $c : (\Sigma_\gg \times \Sigma_{\tau\gg}) \to \mathbb{R}_{\geq 0}$, and we overload $c$ for alignments; $c : (\Sigma_\gg \times \Sigma_{\tau\gg})^* \to \mathbb{R}_{\geq 0}$, for which we have $c(\gamma) = \sum_{i=0}^{|\gamma|-1} c(\gamma_i)$.*

*We call an alignment $\gamma$ under cost function $c$* optimal *iff $\nexists \gamma' : c(\gamma') < c(\gamma)$, i.e., there does not exist an alignment $\gamma'$ with a smaller cost.*

**Definition 7 (Standard cost function).** *The* standard cost function $c_\mathrm{st}$ *is defined for an alignment pair $(\ell, m) \in (\Sigma_\gg \times \Sigma_{\tau\gg})$ as follows:*

$$c_\mathrm{st}(\ell, m) = \begin{cases} 0 & \ell = \gg \text{ and } m = \tau \text{ (silent move, e.g., } (\gg, \tau)) \\ 0 & \ell \in \Sigma \text{ and } m \in \Sigma \text{ and } \ell = m \text{ (e.g., synchronous move } (a, a)) \\ 1 & \ell \in \Sigma \text{ and } m = \gg \text{ (e.g., log move } (a, \gg)) \\ 1 & \ell = \gg \text{ and } m \in \Sigma \text{ (e.g., model move } (\gg, a)) \end{cases}$$

## 3 Maximizing Synchronous Moves

We gather that the standard cost function from Definition 7 is the most commonly used cost function in literature [1,9,10,7], though note that any cost function could be used. The standard cost function may, however, lead to undesired results, as illustrated by the example from Fig. 1. We consider a new cost function that maximizes the number of synchronous moves, since it explains as many log moves as possible. We propose the alternative cost function as follows.

**Definition 8 (max-sync cost function).** *We define the* max-sync cost function $c_{\text{sync}}$ *for an alignment pair as follows (for small* $\varepsilon > 0$*):*

$$c_{\text{sync}}(\ell, m) = \begin{cases} 0 & \ell = \gg \textit{ and } m = \tau \ (\textit{silent move, e.g., } (\gg, \tau)) \\ 0 & \ell \in \Sigma \textit{ and } m \in \Sigma \textit{ and } \ell = m \ (\textit{e.g., synchronous move } (a, a)) \\ 1 & \ell \in \Sigma \textit{ and } m = \gg \ (\textit{e.g., log move } (a, \gg)) \\ \varepsilon & \ell = \gg \textit{ and } m \in \Sigma \ (\textit{e.g., model move } (\gg, a)) \end{cases}$$

This cost function only penalizes log moves, which as a consequence causes an optimal alignment to minimize the number of log moves and thus maximize the number of synchronous moves. The $\varepsilon$ cost for model moves further filters optimal alignments to only include shortest paths through the model that maximize synchronous moves.

An advantage of the max-sync cost function over the standard one is that synchronized behaviour is not sacrificed for shorter paths through the model (as Fig. 1 illustrates). A disadvantage is that in order to maximize the number of synchronous moves, it may be possible that many model moves are required.

## 4 Relating the Model and Event Log

Given a Petri net model N and an event log E $\subseteq \mathcal{B}(\Sigma^*)$, we can distinguish four cases based on the languages that they describe. By distinguishing the relative granularities of N and E we define cases of alignment problems as follows.

**C1:** $\forall \sigma_1 \in E : (\exists \sigma_2 \in \mathcal{L}(\mathrm{N}) : \sigma_1 = \sigma_2)$; all log traces correspond to paths in the model. Then, every log trace can be mapped onto the model by only using synchronous and silent moves, which is optimal for $c_{\text{st}}$ and $c_{\text{sync}}$.

**C2:** $\forall \sigma_1 \in E : (\exists \sigma_2 \in \mathcal{L}(\mathrm{N}) : \sigma_1 \sqsubseteq \sigma_2)$; all log traces correspond to subsequences of paths in the model. Then, every log trace can be mapped onto the model without using any log moves. The example from Fig. 1 for $\sigma = \langle B \rangle$ is such an instance. We hypothesize that $c_{\text{sync}}$ provides better alignments in such instances as $c_{\text{st}}$ may avoid synchronization in favour of shorter paths through the model.

**C3:** $\forall \sigma_1 \in E : (\exists \sigma_2 \in \mathcal{L}(\mathrm{N}) : \sigma_2 \sqsubseteq \sigma_1)$; for every log trace there is a path that forms a subsequence of the log trace. Then, every log trace can be mapped onto the model without using any model moves. Here, $c_{\text{sync}}$ and to some extent $c_{\text{st}}$ can arguably lead to bad results as model moves may be taken to synchronize with 'undesired' behaviour.

**C4:** None of the properties hold. All move types may be necessary for alignments. We regard this as a standard scenario. Depending on the use case, either $c_{\mathrm{st}}$ or $c_{\mathrm{sync}}$ could be preferred.

Aside from C4, we consider cases C2 and C3 as common instances in practice, as logging software often causes either too many or too little events to be logged or in case the model is over/underspecified. Discrepancies then show whether the model is of the right granularity. We note that it is also possible to hide certain activities in the model or log before alignment. This is however not trivial, especially if there are (slight) deviations in the log such that the alignment problem does not fit C2 or C3 exactly anymore.

When considering instances that exactly fit case C2 or C3, we can construct alignments by respectively removing all log or model moves from the product of the model and log. We define the cost functions $c_{\mathrm{add}}$ and $c_{\mathrm{rem}}$ to be variants of $c_{\mathrm{st}}$ such that model and log moves respectively have a cost of $\infty$. We argue that this results in a better 'alignment quality' and reduces the time for its construction.

## 5 Algorithms for Computing Alignments

We consider two algorithms for computing alignments, which we discuss as follows. Both algorithms take the product Petri net as input.

**A*.** The A* algorithm [7] computes the shortest path from the initial marking to the final marking on the marking graph for a given cost function. The heuristic function for A* exploits the Petri net marking equation, which can be achieved using Integer Linear Programming (ILP), to prune the search space.

**Symbolic algorithm.** The symbolic algorithm [8] was recently developed as an improvement over A* for large state spaces. It exploits symbolic reachability to search for an alignment, i.e., considering sets of markings instead of single ones. By restricting the cost function to only allow 0 or 1-cost moves, optimal alignments can be computed by only taking a 1-cost move after exploring all markings reachable via 0-cost steps. We refer to this algorithm by Sym.

## 6 Preprocessing Reference Models for Large Event Logs

When constructing an alignment under the $c_{\mathrm{sync}}$ cost function, we can disregard the cost for model moves to a certain extent. The goal is to find a path through the model that maximizes the number of synchronous moves. We can achieve this by searching for a subsequence in the log trace that is also included in the language of the reference model. By computing the transitive closure of the model's marking graph, we find all paths and subsequences of paths through the model. For every log trace we can use dynamic programming to search for the maximum-length subsequence in the log trace that can be replayed in the transitive closure graph (TCG), from which we can construct a path through the marking graph and obtain an optimal alignment.

We construct a TCG as described in Definition 9. Here, $\tau$-edges are added to the marking graph such that every marking is reachable via $\tau$-steps. After determinization, for every path $P$ in the original marking graph the TCG contains all paths $P'$ such that $\lambda(P') \sqsubseteq \lambda(P)$.

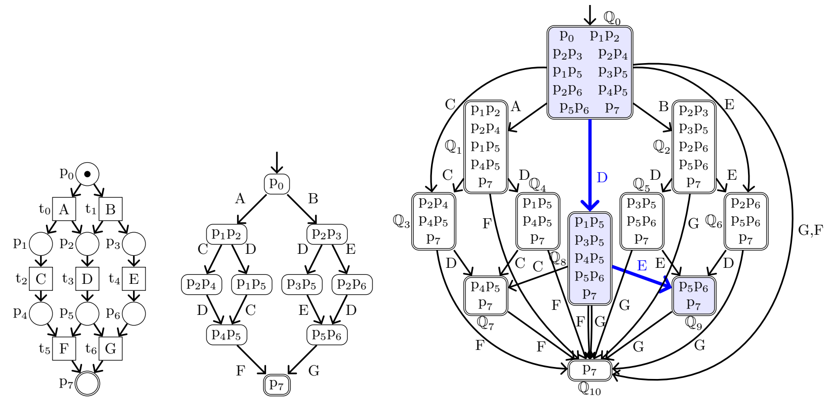


Fig. 2: Example Petri net model (left), its corresponding marking graph (middle) and transitive closure graph (right) with the sequence $\langle D, E \rangle$ highlighted.

We can use this property to search for a subsequence of the log trace that can fully synchronize with the model. For instance in the example of Fig. 2, consider a log trace $\sigma = \langle F, D, E, B \rangle$. The $F$ event can be fired from $\mathbb{Q}_0$, after which the TCG is in state $\mathbb{Q}_{10}$. From this state, it is not possible to perform any other event from log trace. A better choice would be to skip the $F$ event (which would then be a log move) and form the subsequence $\langle D, E \rangle$, as highlighted[4]. We call the maximum-length subsequence $\hat{\sigma}$ from the log trace a *maximum fitting subsequence* if $\hat{\sigma}$ also forms a path through the TCG, as defined in Definition 10.

**Definition 9 (Transitive closure graph).** *Given a marking graph* $MG = (Q, \Sigma_\tau, \delta, q_0, q_\mathrm{F})$, *we first construct an extended marking graph* $MG' = (Q, \Sigma_\tau, \delta', q_0, q_\mathrm{F})$ *with* $\delta' = \delta \cup \{(\mathtt{src}(e), \tau, \mathtt{tar}(e)) \mid e \in \delta\}$. *A* transitive closure graph (TCG), $TCG = (\mathbb{Q}, \Sigma, \Delta, \mathbb{Q}_0, \mathbb{Q}_\mathrm{F})$ *is defined as the result of determinizing* $MG'$ *(by using a standard determinization algorithm [11]) and by then removing all non-final states from the TCG such that* $\mathbb{Q} \subseteq 2^Q$, $\Sigma = \Sigma_\tau \setminus \{\tau\}$, $\Delta \subseteq (\mathbb{Q} \times \Sigma \times \mathbb{Q})$, $\mathbb{Q}_0 = Q$, *and* $\mathbb{Q}_\mathrm{F} = \mathbb{Q}$.

*For an edge* $e \in \Delta$ *we also use the notation* $\mathtt{src}(e)$ *and* $\mathtt{tar}(e)$ *to respectively refer to the source and target marking sets in the TCG. Paths over the TCG are defined analogously to paths over marking graphs (Definition 3) and we use* $Paths(TCG)$ *and* $\mathcal{L}(TCG)$ *to respectively denote the set of all paths in the TCG and the language of the TCG.*

**Definition 10 (Maximum fitting subsequence).** *Given a sequence (log trace)* $\sigma \in \Sigma^*$ *and* $TCG = (\mathbb{Q}, \Sigma, \Delta, \mathbb{Q}_0, \mathbb{Q}_\mathrm{F})$, *then* $\hat{\sigma} \sqsubseteq \sigma$ *is a* maximum fitting subsequence *if and only if* $\hat{\sigma} \in \mathcal{L}(TCG) \wedge \forall \hat{\sigma}' \sqsubseteq \sigma : \hat{\sigma}' \in \mathcal{L}(TCG) \Rightarrow |\hat{\sigma}| \geq |\hat{\sigma}'|$. *We construct* $\hat{\sigma}$ *by using dynamic programming to search for a subsequence of* $\sigma$ *that is a maximum-length path in the TCG.*

[4] It might be interesting to note that after performing the $D$ action in the TCG, in the Petri net we have not yet made the choice to fire either an $A$ or a $B$ transition; we implicitly make the decision to fire the $B$ transition after choosing the $E$ event.

**Algorithm 1:** Path construction from a maximum fitting subsequence $\hat{\sigma}$

```
1  func PC($TCG = (\mathbb{Q}, \Sigma, \Delta, \mathbb{Q}_0, \mathbb{Q}_F), MG = (Q, \Sigma_\tau, \delta, q_0, q_F), \hat{\sigma} = \langle \hat{\sigma}_0, \hat{\sigma}_1, \ldots, \hat{\sigma}_n \rangle$)
2      // Construct path MFP on $TCG$ such that $\lambda(\texttt{MFP}) = \hat{\sigma}$
3      $\texttt{MFP} := \langle (\mathbb{Q}_0, \hat{\sigma}_0, S), (S, \hat{\sigma}_1, S'), \ldots, (S'', \hat{\sigma}_n, S''') \rangle$ s.t. $\forall_{0 \leq i \leq n} : \texttt{MFP}_i \in \Delta$
4      $P :=$ BWD($MG, q_F, \hat{\sigma}_n, \texttt{tar}(\texttt{MFP}_n)$) // Path $\hat{\sigma}_n$ to $q_F$ on $MG$
5      for $i := n - 1;\ i \geq 0;\ i := i - 1$ do // Add paths from $\hat{\sigma}_i$ to $\hat{\sigma}_{i+1}$
6          $P :=$ BWD($MG, \texttt{src}(P_0), \hat{\sigma}_i, \texttt{tar}(\texttt{MFP}_i)$) $\cdot P$
7      return BWD($MG, \texttt{src}(P_0), \bot, \mathbb{Q}_0$) $\cdot P$ // Add path from $q_0$ to $\hat{\sigma}_0$
8  func BWD($MG = (Q, \Sigma_\tau, \delta, q_0, q_F), m \in Q, a \in (\Sigma \cup \bot), S \subseteq Q$)
9      $W := \langle m \rangle$ // Sequence of unvisited markings in the backward search
10     $\forall m \in S : F[m] := \texttt{Null}$ // Mapping from markings to edges ($F : Q \to \delta$)
11     for $i := 0;\ i < |W|;\ i := i + 1$ do // Continue for all markings in $W$
12         if $\exists m' \in Q, a' \in \Sigma : (m', a', W_i) \in \delta \wedge (a' = a \vee (a = \bot \wedge m' = q_0))$ then
13             $P := \langle (m', a', W_i) \rangle$ // Found path from $a$ (or initial marking)
14             while $tar(P_{|P|-1}) \neq m$ do  $P := P \cdot F[\texttt{tar}(P_{|P|-1})]$
15             return $P$ // Shortest path from $a$ (or $q_0$) to $m$
16         forall $e \in \delta : src(e) \in (S \setminus W) \wedge tar(e) = W_i$ do
17             $W := W \cdot \langle \texttt{src}(e) \rangle$ // Add predecessor markings of $m$ to $W$
18             $F[\texttt{src}(e)] := e$ // Direct the source markings towards $m$
19     return $\langle \rangle$ // No path from $a$ (or $q_0$) is found (should never occur)
```

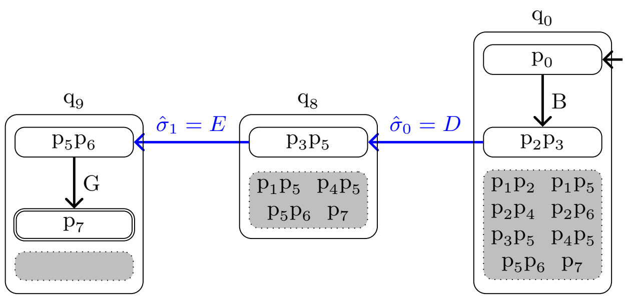
Once we have found the maximum fitting subsequence $\hat{\sigma}$ for a given model and log trace, we still have to determine which model moves should be applied to form a path through the original model. This can be achieved by using the TCG and traversing $\hat{\sigma}$ in a backwards fashion as we show in Algorithm 1.

We first construct a path `MFP` from the subsequence $\hat{\sigma}$ (line 3), in the example from Fig. 2 with $\hat{\sigma} = \langle D, E \rangle$ (see also Fig. 3 for an illustration of the path construction process) this would be $\texttt{MFP} = \langle (\mathbb{Q}_0, D, \mathbb{Q}_8), (\mathbb{Q}_8, E, \mathbb{Q}_9) \rangle$. Then in line 4, a backward search procedure (`BWD`) is called to search for a path $P$ in the marking graph from an $E$-edge to the final marking ($\mathrm{p}_7$).

The `BWD` procedure takes a target marking $m$, label $a$ and search space $S$ as arguments. A sequence $W$ is maintained to process unvisited markings from $S$ and a mapping $F : Q \to \delta$ is used for reconstructing the path. Starting from the target marking $m$ (which is $W_0$), the procedure searches for edges $e$ directing towards $m$ in line 16-18 such that $\texttt{src}(e)$ is in $S$ and not already visited. For every such edge $e$, its source is appended to $W$ (to be considered in a future iteration) and $\texttt{src}(e)$ is mapped to $e$ for later path reconstruction.

Following iterations of the for loop in line 11-18 consider a predecessor $W_i$ of $m$ and search for edges directing to $W_i$. This way, the search space is traversed backwards in a breadth-first manner, resulting in shortest paths to $m$.

In line 12-15 the `BWD` procedure checks whether there is an edge $m' \xrightarrow{a} W_i$ for some $m'$ (or an edge $q_0 \xrightarrow{a'} W_i$ for arbitrary $a'$ in case $a = \bot$) and if so, constructs a path towards $m$ in line 14 which is then returned. In the example, the path $\langle (\mathrm{p}_3\mathrm{p}_5, E, \mathrm{p}_5\mathrm{p}_6), (\mathrm{p}_5\mathrm{p}_6, G, \mathrm{p}_7) \rangle$ will be returned for the first `BWD` call.



$$\gamma = \begin{array}{|c|c|c|c|c|c|} F & \gg & D & E & B & \gg \\ \hline \gg & B & D & E & \gg & G \\ & \mathrm{t}_1 & \mathrm{t}_3 & \mathrm{t}_4 & & \mathrm{t}_6 \end{array}$$

Fig. 3: Path construction using Algorithm 1 on the example from Fig. 2 for a maximum fitting subsequence $\hat\sigma = \langle D, E\rangle \sqsubseteq \langle F, D, E, B\rangle$. Markings in the grey region are not part of the path. The resulting alignment $\gamma$ is shown on the right.

After the first `BWD` call, the main function iterates backwards over all remaining edges from `MFP` (line 5-6) to create paths between $\hat\sigma_i$ and $\hat\sigma_{i+1}$, which are inserted in the path before $P$. Finally, in line 7 a path from the initial marking $q_0$ towards the first label $\hat\sigma_0$ is inserted before $P$ to complete the path (here the label is set to $\perp$ to search for $q_0$ in the `BWD` procedure).

In the example we first compute the path $\langle(\mathrm{p}_3\mathrm{p}_5, E, \mathrm{p}_5\mathrm{p}_6), (\mathrm{p}_5\mathrm{p}_6, G, \mathrm{p}_7)\rangle$ in line 4, then after line 5-6 we insert the path $\langle(\mathrm{p}_2\mathrm{p}_3, D, \mathrm{p}_3\mathrm{p}_5)\rangle$, and in line 7 we insert the path from the initial state $q_0 = \mathrm{p}_0$, $\langle(\mathrm{p}_0, B, \mathrm{p}_2\mathrm{p}_3)\rangle$ to create the complete minimal-length path $P$ in the marking graph such that $\hat\sigma \sqsubseteq \lambda(P)$.

The alignment can be reconstructed by marking all events in the maximum fitting subsequence as synchronous moves, by marking the remaining labels in the log trace as log moves, and inserting the model and silent moves (as computed by Algorithm 1) at the appropriate places.

Note that the TCG algorithm does not exactly compute an alignment for the cost function $c_{\text{sync}}$. The backwards BFS does ensure a shortest path through the model from the initial to the final marking while synchronizing with the maximum fitting subsequence. However, there might exist a different maximum fitting subsequence that leads to a different path through the model with a lower total cost (fewer model moves). This can be repaired by computing the alignments for all maximum fitting subsequences. If the marking graph contains cycles, the corresponding markings get contracted to a single state in the TCG with a self-loop for each activity in the cycle. Also, the TCG may in theory contain exponentially more states than there are markings in the marking graph. However, in industrial models (Section 7.3), we found that in many cases the number of states in the TCG is at most two times more than the number of markings in the marking graph.

## 7 Experiments

For the experiments, we considered two types of alignment problems. On the one hand, a large reference model accompanied by an event log consisting of a single log trace, and on the other hand a smaller reference model accompanied by an event log of many traces. All experiments were performed on an Intel® Core™ i7-4710MQ processor with 2.50GHz and 7.4GiB memory. For all experiments, we have set a timeout of 60 seconds. When computing averages, a timeout also counts as 60 seconds.

We investigate differences between the alignments resulting from using the standard- and max-sync cost functions, and compare alignment computation times for A* (with ILP, using the implementation from RapidProM [12]) and the symbolic algorithm (implemented in the LTSmin model checker [13]). We further investigate specific alignment problems, cases C2 and C3 as discussed in Section 4. Finally, we also look at models accompanied by many log traces to compare the performance of the TCG algorithm (implemented in ProM [14]) with the other algorithms. For all large models with singleton log traces we used 8 threads for computing alignments, and for smaller models with many log traces we only used a single thread per alignment computation[5]. All results are available online at https://github.com/utwente-fmt/MaxSync-BPM2018.

## 7.1 Experiments Using Large Models and Singleton Event Logs

**Model generation.** Using the PTandLogGenerator [15] we generated Petri net models with process operators and additional features set to their defaults; where the respective probabilities for sequence, XOR, parallel, loop, OR are set to 45%, 20%, 20%, 10%, and 5%. The additional features for the occurrence of silent and duplicate activities, and long-term dependencies were all set to 20%.

To examine scalability we ranged the average number of activities from 25, 50, and 75, resulting in respectively 110, 271, and 370 transitions on average. For these settings, we generated 30 models (thus 90 in total) and generated a single log trace per model. For this log trace we added 10%, 30%, 50%, and 70% noise in three different ways (thus 12 noisy singleton logs are created); by (1) adding, removing and swapping events (resembling case C4), (2), by only adding events (resembling case C3), and (3) by only removing events (resembling case C2). In total there are 1,080 noisy singleton logs. We first consider noise of type 1.

**Alignment differences.** In Table 1 we compare the resulting alignments, produced by Sym, for the different cost functions. When comparing the overall results of $c_{\text{st}}$ and $c_{\text{sync}}$ (rightmost column), we observe that $c_{\text{sync}}$ uses about 43% fewer log moves, which are added as synchronous moves. However in doing so, more than six times as many model moves are required.

When looking at an increase in the amount of noise, the relative difference between the number of log moves remains the same, while this difference in model moves slightly drops. When increasing the number of activities from 25 to 75, We observe an increase in the number of model moves for $c_{\text{sync}}$ from 3.2 times to 9.3 times as many compared to $c_{\text{st}}$. As a corresponding result from this

[5] We consider multi-threaded experiments not as useful in this scenario, as the problem can be parallelized by dividing the log traces over the different threads and computing the alignments independently.

Table 1: Comparison between alignments generated using the $c_{\text{st}}$ and $c_{\text{sync}}$ cost functions. The numbers show averages, e.g., the value of 2.3 in the top-left corner denotes the average number of log moves for all computed alignments for which 10% noise is added, using the $c_{\text{st}}$ cost function.

| | **Noise added (add, remove, swap)** | | | | | | | | **Number of activities** | | | | | | **Average** | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | **10%** | | **30%** | | **50%** | | **70%** | | **25** | | **50** | | **75** | | | |
| | $c_{\text{st}}$ | $c_{\text{sync}}$ | $c_{\text{st}}$ | $c_{\text{sync}}$ | $c_{\text{st}}$ | $c_{\text{sync}}$ | $c_{\text{st}}$ | $c_{\text{sync}}$ | $c_{\text{st}}$ | $c_{\text{sync}}$ | $c_{\text{st}}$ | $c_{\text{sync}}$ | $c_{\text{st}}$ | $c_{\text{sync}}$ | $c_{\text{st}}$ | $c_{\text{sync}}$ |
| **Log** | 2.3 | 1.3 | 6.5 | 3.6 | 9.4 | 5.4 | 10.9 | 6.3 | 4.7 | 3.2 | 8.9 | 4.6 | 8.4 | 4.5 | 7.0 | 4.0 |
| **Model** | 2.0 | 15.7 | 4.6 | 30.9 | 5.8 | 35.3 | 6.2 | 38.1 | 3.3 | 10.7 | 5.6 | 39.1 | 5.4 | 50.2 | 4.5 | 29.4 |
| **Sync** | 28.5 | 29.6 | 20.9 | 23.7 | 16.8 | 20.8 | 14.5 | 19.1 | 13.8 | 15.4 | 23.2 | 27.5 | 29.4 | 33.3 | 20.6 | 23.6 |
| **Silent** | 17.3 | 24.4 | 14.7 | 30.4 | 13.6 | 35.3 | 12.8 | 35.1 | 10.0 | 13.3 | 16.2 | 39.6 | 21.6 | 51.6 | 14.7 | 31.0 |

effect, the difference between log moves from $c_{\text{sync}}$ and $c_{\text{st}}$ stays relatively the same for increasing activities.

We conclude that for $c_{\text{sync}}$ the relative reduction in log moves stays mostly the same, when fluctuating the amount of noise or size of the model. The size of the model seems to greatly affect the number of model moves for $c_{\text{sync}}$, making alignments from $c_{\text{st}}$ and $c_{\text{sync}}$ more diverse for larger models.

**Performance results.** We observed that while Sym is faster in computing alignments than the A* algorithm on $c_{\text{st}}$ (it takes on average 15.8s for computing an alignment using A* and 10.5s for Sym), for the $c_{\text{sync}}$ cost function A* is outperforming the symbolic algorithm (13.7s for A* and 16.5s for Sym). This has to do with the effect that the symbolic algorithm will explore the entire model before attempting a single log move whereas A* does not.

## 7.2 Alignment Problems that Only Add or Remove Events

**Alignment differences.** In Table 2 we compare the resulting alignments for adding or removing events. When inspecting the `Add` case, we find that the $c_{\text{st}}$ already avoids model moves for the most part as we would expect. Moreover, there are only small differences between alignments from $c_{\text{st}}$ and $c_{\text{add}}$. For $c_{\text{sync}}$, many model moves may be chosen to increase the number of synchronous moves. These additional synchronous moves are arguably not part of the 'desired' alignment since they require a large detour through the model.

When removing events from the log trace, the $c_{\text{st}}$ cost function is only partly able to describe the removal of events as it still chooses log moves. The $c_{\text{sync}}$ cost function does not take any log moves as this maximizes the number of synchronous moves, making it equal to $c_{\text{rem}}$. When comparing $c_{\text{st}}$ and $c_{\text{sync}}$, we could argue that for the `Add` case, the $c_{\text{st}}$ cost function better represents a 'correct' alignment and for the `Rem` case $c_{\text{sync}}$ is better suited.

**Performance results.** We observed that for $c_{\text{st}}$, A* performs relatively bad for the `Rem` case (14.1s on average), but significantly better for $c_{\text{sync}}$ (2.3s on average). We argue that A* for $c_{\text{st}}$ tries to perform many log moves, that results in a lot of backtracking, while for $c_{\text{sync}}$ the algorithm avoids log moves entirely. The symbolic algorithm uses 6.6s and 7.3s on average for $c_{\text{st}}$ and $c_{\text{sync}}$ respectively. For the `Rem` case, we do not observe a significant difference in the

Table 2: Comparison between alignments generated using the $c_{\mathrm{st}}$ and $c_{\mathrm{sync}}$ cost functions for alignment problems, where noise only consist of adding (`Add`) or removing (`Rem`) events. The cost functions $c_{\mathrm{add}}$ and $c_{\mathrm{rem}}$ are variations on $c_{\mathrm{st}}$ such that model and log moves respectively have a cost of $\infty$.

| | **Log events added** (`Add`) | | | | | | | | | **Log events removed** (`Rem`) | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | **10%** | | | **30%** | | | **50%** | | | **10%** | | | **30%** | | | **50%** | | |
| | $c_{\mathrm{st}}$ | $c_{\mathrm{sync}}$ | $c_{\mathrm{add}}$ | $c_{\mathrm{st}}$ | $c_{\mathrm{sync}}$ | $c_{\mathrm{add}}$ | $c_{\mathrm{st}}$ | $c_{\mathrm{sync}}$ | $c_{\mathrm{add}}$ | $c_{\mathrm{st}}$ | $c_{\mathrm{sync}}$ | $c_{\mathrm{rem}}$ | $c_{\mathrm{st}}$ | $c_{\mathrm{sync}}$ | $c_{\mathrm{rem}}$ | $c_{\mathrm{st}}$ | $c_{\mathrm{sync}}$ | $c_{\mathrm{rem}}$ |
| **Log** | 3.1 | 2.0 | 3.1 | 7.5 | 5.1 | 7.6 | 10.6 | 7.4 | 10.8 | 0.3 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 | 2.5 | 0.0 | 0.0 |
| **Model** | 0.0 | 13.1 | 0.0 | 0.1 | 21.6 | 0.0 | 0.2 | 23.1 | 0.0 | 3.0 | 3.3 | 3.3 | 6.3 | 7.7 | 7.7 | 8.0 | 11.9 | 11.9 |
| **Sync** | 29.4 | 30.5 | 29.4 | 26.5 | 28.9 | 26.4 | 24.1 | 27.4 | 23.9 | 30.3 | 30.7 | 30.7 | 21.0 | 22.0 | 22.0 | 13.9 | 16.4 | 16.4 |
| **Silent** | 16.3 | 23.6 | 16.2 | 15.5 | 31.0 | 15.4 | 14.0 | 30.0 | 13.8 | 18.4 | 18.5 | 18.5 | 16.0 | 16.7 | 16.7 | 13.2 | 16.0 | 16.0 |

performance times when considering $c_{\mathrm{rem}}$, i.e., removing the log moves. This is because both algorithms already avoid log moves for the $c_{\mathrm{sync}}$ cost function.

For the `Add` case, both A* and Sym require more time for computing alignments for $c_{\mathrm{sync}}$ than for $c_{\mathrm{st}}$. When removing model moves ($c_{\mathrm{add}}$), A* and Sym perform in respectively 36% and 77% of the time required for $c_{\mathrm{st}}$ (thus 3.4s and 9.3s). By removing the model moves, both algorithms no longer have to explore a large part of the state-space and only have to decide on which log moves, synchronous and silent actions to chose, which is especially beneficial for A*.

## 7.3 Experiments Using Event Logs with More Traces

We now consider smaller models that have to align many log traces. For our experiments, we selected 9 instances from the 735 industrial business process Petri net models from financial services, telecommunications and other domains, obtained from the data sets presented in Fahland et al. [16].

For our selection, we computed the transitive closure graph (TCG) and considered the instances for which we were able to compute TCG within 60 seconds. From this set, we selected the 9 most interesting cases, e.g., the models with the largest Petri net models, largest marking graphs, largest TCG graph, and largest TCG construction time. On average the marking graph contains 108 markings and the TCG 134 states. In the worst case, the number of states in the TCG was 200, which doubled the number of markings in the marking graph. We did not find a large difference between the performance results of the individual experiments.

For each model, we generated a set of 10, 100, 1,000, and 10,000 log traces for 10%, 30%, 50%, and 70% noise added by adding, removing, and swapping events. Thus in total, we have 16 event logs per model. We compared the performance of the TCG algorithm with that of A* using a single thread. We also experimented with the symbolic algorithm, but its setup time per alignment computation is too large to provide meaningful results. Note that in our experiments, we only consider the $c_{\mathrm{sync}}$ cost function. The TCG algorithm is not applicable to the $c_{\mathrm{st}}$ cost function.

**Results.** The results are summarized in Table 3. On average, the TCG algorithm used 270 milliseconds for computing the transitive closure graph. When

Table 3: Alignment computation time (in milliseconds) for models with many log traces. TCG-comp, TCG-align, and TCG respectively denote the time for computing the TCG, the time for aligning all log traces, and the sum of the two.

| **Log size** | **TCG-comp** | **TCG-align** | **TCG** | **A*** |
|---:|---:|---:|---:|---:|
| 10 | 272 | 9 | 281 | 426 |
| 100 | 269 | 20 | 289 | 3,539 |
| 1,000 | 265 | 161 | 426 | 13,247 |
| 10,000 | 274 | 1,542 | 1,936 | 33,906 |

| **Noise** | **TCG** | **A*** |
|---:|---:|---:|
| 10% | 727 | 9,320 |
| 30% | 729 | 13,919 |
| 50% | 750 | 14,199 |
| 70% | 727 | 13,679 |

increasing the number of log traces (left table), we see that the preprocessing step of the TCG algorithm remains a significant part of its total time for up to 1,000 log traces. The A* algorithm has to create a synchronous product of the model and log trace for each instance, and expectedly takes more time in total. For 10,000 log traces, A* is 17 times slower than the TCG algorithm. But even for 10 log traces, the TCG algorithm outperforms A* by almost a factor of two.

When comparing the results for different amounts of noise (right table), we see practically no difference in the computation times for the TCG algorithm. The A* algorithm does require significantly more time for 30%, 50%, and 70% noise compared to the 10% case. We argue that from 30% noise onwards, A* has to visit most of the state-space to construct an optimal alignment. In the TCG algorithm, noise does not seem to affect its performance.

## 8 Related Work

One of the earliest works in conformance checking was from Cook and Wolf [17]. They compared log traces with paths generated from the model.

One technique to check for conformance is *token-based replay* [4]. The idea is to 'replay' the event logs by trying to fire the corresponding transitions, while keeping track of possible missing and remaining tokens in the model. However, this technique does not provide a path through the model. When traces in the event log deviate a lot, the Petri net may get flooded with tokens and the tokens do not provide good insights anymore.

Alignments were introduced [5,7] to overcome the limitations of the token-based replay technique. Alignments formulate conformance checking as an optimization problem, i.e., minimizing the alignment cost-function. Since its introduction, alignments have quickly become the standard technique for conformance checking along with the A* algorithm for computing alignments [9]. In previous work [8] we presented the symbolic algorithm for alignments and we analysed how different model characteristics influence the computation times for $c_{\mathrm{st}}$.

For larger models, techniques have been developed to decompose the Petri net in smaller subprocesses [18]. For instance, fragments that have a single-entry and single-exit node (SESE) represent an isolated part of the model. This way, localizing conformance problems becomes easier in large models. It would be interesting to combine the TCG algorithm with such decomposed models.

A sub-field of alignments is to compute a *prefix-alignment* for an incomplete log trace. This is useful for analysing processes in real-time instead of a-posteriori. Several techniques exist for computing prefix-alignments [7,19]. The TCG approach that we introduced in this paper could also be suitable for computing prefix-alignments. Recently, Burattin and Carmona [20] introduced a technique similar to the TCG approach, in which the marking graph is extended with additional edges to allow for deviations. However, it cannot guarantee optimality as a *single* successor marking is chosen per event, while instead we consider all possible successors and can, therefore, better adapt for future events.

In a more general setting, conformance checking is related to finding a *longest common subsequence*, computing a *diff*, or computing minimal *edit distances*. Here, the problem is translated to searching for a string $B$ from a regular language $L$ such that the edit distance of $B$ and an input word $\alpha$ is minimal [21].

## 9 Conclusion

In this paper, we considered a max-sync cost function that instead of minimizing discrepancies between the log trace and the model, maximizes the number of synchronous moves. We empirically evaluated the differences with the standard cost function, compared the alignment computation times. The max-sync cost function also lead to a new algorithm for computing alignments.

We observed that in general, a considerable amount of model moves may be required to add a few additional synchronous moves, when comparing max-sync with the standard cost function. However, when alignment problems are structured such that log moves are on a lower granularity than the model, a max-sync cost function may be better suited. We also observed a significant performance improvement in alignment construction if alignments can be formed without taking any model moves or without any log moves.

On industrial models with many log traces, we showed that our new algorithm, which uses a preprocessing step on the model, is an order of magnitude faster in computing alignments on many log traces for the max-sync cost function.

We conclude that the max-sync cost function is complementary to the standard one as it provides an alternative view that may be preferable in some contexts, and it may also significantly reduce the alignment construction time.

## References

1. van der Aalst, W.M.P.: Process Mining: Data Science in Action. Springer (2016)
2. Liu, C., van Dongen, B.F., Assy, N., van der Aalst, W.M.P.: Component behavior discovery from software execution data. In: 2016 IEEE Symposium Series on Computational Intelligence, SSCI 2016, December 6-9, 2016. (2016) 1–8
3. Leemans, M., van der Aalst, W.M.P.: Process mining in software systems: Discovering real-life business transactions and process models from distributed systems. In: 18th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MoDELS 2015, September 30 - October 2, 2015. (2015) 44–53

4. Rozinat, A., van der Aalst, W.M.P.: Conformance checking of processes based on monitoring real behavior. Information Systems **33**(1) (2008) 64 – 95
5. van der Aalst, W.M.P., Adriansyah, A., van Dongen, B.F.: Replaying history on process models for conformance checking and performance analysis. Wiley Interdiscip. Reviews: Data Mining and Knowledge Discovery **2**(2) (2012) 182–192
6. Adriansyah, A., Sidorova, N., van Dongen, B.F.: Cost-Based Fitness in Conformance Checking. In: 11th International Conference on Application of Concurrency to System Design, ACSD 2011, 20-24 June, 2011. (2011) 57–66
7. Adriansyah, A.: Aligning observed and modeled behavior. PhD thesis, Eindhoven University of Technology, The Netherlands (2014)
8. Bloemen, V., van de Pol, J., van der Aalst, W.M.P.: Symbolically Aligning Observed and Modelled Behaviour. In: 18th International Conference on Application of Concurrency to System Design, ACSD 2018, 24-29 June, 2018. (2018)
9. van Zelst, S.J., Bolt, A., van Dongen, B.F.: Tuning Alignment Computation: An Experimental Evaluation. In: Proc. of the Int. Workshop on Algorithms & Theories for the Analysis of Event Data, ATAED 2017, June 25-30, 2017. (2017) 1–15
10. Adriansyah, A., van Dongen, B.F., van der Aalst, W.M.P.: Memory-efficient alignment of observed and modeled behavior. Technical report (2013)
11. Sudkamp, T.A.: Languages and Machines : An Introduction to the Theory of Computer Science. Addison-Wesley Longman Publishing Co., Inc. (1988)
12. van der Aalst, W.M.P., Bolt, A., van Zelst, S.J.: RapidProM: Mine Your Processes and Not Just Your Data. CoRR **abs/1703.03740** (2017)
13. Kant, G., Laarman, A., Meijer, J., van de Pol, J., Blom, S., van Dijk, T.: LTSmin: High-Performance Language-Independent Model Checking. In Baier, C., Tinelli, C., eds.: Tools and Algorithms for the Construction and Analysis of Systems. Volume 9035 of LNCS. Springer Berlin Heidelberg (2015) 692–707
14. Verbeek, H.M.W., Buijs, J.C.A.M., van Dongen, B.F., van der Aalst, W.M.P. In: XES, XESame, and ProM 6. Springer Berlin Heidelberg (2011) 60–75
15. Jouck, T., Depaire, B.: PTandLogGenerator: A Generator for Artificial Event Data. In: Proceedings of the BPM Demo Track 2016 Co-located with the 14th International Conference on Business Process Management (BPM 2016), September 21, 2016. (2016) 23–27
16. Fahland, D., Favre, C., Koehler, J., Lohmann, N., Völzer, H., Wolf, K.: Analysis on demand: Instantaneous soundness checking of industrial business process models. Data Knowl. Eng. **70**(5) (2011) 448–466
17. Cook, J.E., Wolf, A.L.: Software Process Validation: Quantitatively Measuring the Correspondence of a Process to a Model. ACM Trans. Softw. Eng. Methodol. **8**(2) (1999) 147–176
18. Polyvyanyy, A., Vanhatalo, J., Völzer, H. In: Simplified Computation and Generalization of the Refined Process Structure Tree. Springer Berlin Heidelberg (2011) 25–41
19. van Zelst, S.J., Bolt, A., Hassani, M., van Dongen, B.F., van der Aalst, W.M.P.: Online conformance checking: relating event streams to process models using prefix-alignments. International Journal of Data Science and Analytics (2017)
20. Burattin, A., Carmona, J.: A Framework for Online Conformance Checking. In: Proc. of the 13th Int. Workshop on Business Process Intelligence (BPI 2017). (2017)
21. Wagner, R.A.: Order-n Correction for Regular Languages. Commun. ACM **17**(5) (1974) 265–268