Aligning Observed and Modelled Behaviour by Maximizing Synchronous Moves and Using Milestones

Vincent Bloemen[a,*], Sebastiaan van Zelst[b,c], Wil van der Aalst[c,b], Boudewijn van Dongen[d], Jaco van de Pol[e,a]

[a] *University of Twente, Enschede, The Netherlands*
[b] *Fraunhofer FIT, Sankt Augustin, Germany*
[c] *RWTH Aachen University, Aachen, Germany*
[d] *Eindhoven University of Technology, Eindhoven, The Netherlands*
[e] *University of Aarhus, Aarhus, Denmark*

Abstract

Given a process model and an event log, conformance checking aims to relate the two together, e.g. to detect discrepancies between them. For the synchronous product net of the process and a log trace, we can assign different costs to a synchronous move, and a move in the log or model. By computing a path through this (synchronous) product net, whilst minimizing the total cost, we create a so-called optimal alignment – which is considered to be the primary target result for conformance checking. Traditional alignment-based approaches (1) have performance problems for larger logs and models, and (2) do not provide reliable diagnostics for non-conforming behaviour (e.g. bottleneck analysis is based on events that did not happen). This is the reason to explore an alternative approach that maximizes the use of observed events. We also introduce the notion of milestone activities, i.e. unskippable activities, and show how the different approaches relate to each other. We propose a data structure, that can be computed from the process model, which can be used for (1) computing alignments of many log traces that maximize synchronous moves, and (2) as a means for analysing non-conforming behaviour. In our experiments we show the differences of various alignment cost functions. We also show how the performance of constructing alignments with our data structure relates to that of the state-of-the-art techniques.

Keywords: process mining, conformance checking, alignments, alignment cost function, transitive closure graph, milestone events

*Corresponding author
Email address: v.bloemen@utwente.nl (Vincent Bloemen)

1. Introduction

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 enterprise-level software systems [16, 15]. Such behavioural data is often referred to as an event log, which can be seen as a set 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 precisely to a valid process instance, i.e. it deviates from its intended behaviour as specified by its reference model.

In the field of process mining [24] there are three main branches: *process 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 model based on facts derived from event data.

Conformance checking [8] assesses to what degree the event log and model conform to each other. Early conformance checking techniques [22] are based on simple heuristics and, therefore, may yield ambiguous/unpredictable results. *Alignments* [25, 2] 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 the 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 execution of an action in 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 τ). We are unable to observe such actions.

Consider the example model of a simple file reading system given in Figure 1 and the trace $\sigma = \langle A, D, B, D \rangle$. An alignment for the model and σ is given by γ^0 (top right in Figure 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, (A, t_0), represents a synchronous move, in which both the log and the path in the model describe the execution of an A activity (open file). The next pair, $(D, \gg)$, is a log move where the logged trace describes the execution of a D activity that is not mapped to an activity in the model. 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. $(\gg, \mathrm{t}_2)$. Finally, the alignment ends with two synchronous moves.

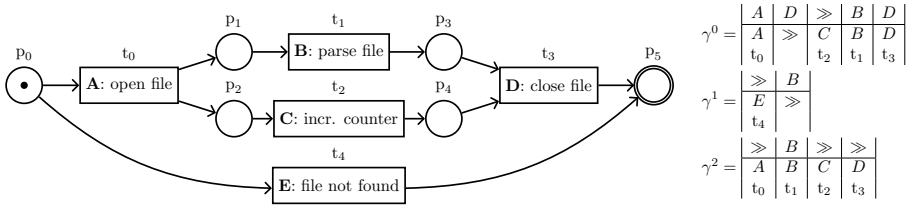


Figure 1: Example process model (in Petri net formalism) for a simple file reading system and an alignment for the trace $\sigma = \langle A, D, B, D \rangle$ (γ^0). For the trace $\sigma = \langle B \rangle$, two optimal alignments are given using the standard- (γ^1) and proposed (γ^2) cost functions.

An optimal alignment is an alignment that minimizes a given cost function. Typically, each type of move gets a non-negative value. 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 a cost 0 to synchronous and silent moves. In practice, the A* shortest path algorithm [1] is often used for computing optimal alignments using the aforementioned cost function as a distance function in its search graph.

In this paper, we argue that the standard cost function is not always the best-suited function for computing alignments. Consider the model in Figure 1 again, together with the trace $\sigma' = \langle B \rangle$. An optimal alignment using the standard cost function would result in γ^1. Considering that event B is observed behaviour, i.e. the system logged "parse file", it is 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 that the number of synchronous moves are maximized, an optimal alignment would result in γ^2. One may prefer this function, as it provides a more sensible explanation for the logged behaviour.

We formalise the relation between the event log and the process model, and show how different cost functions affect the resulting alignments. Regarding the cost-function where we maximize synchronous moves, we employ a data structure, called a *Milestone Transitive Closure Graph* (MTCG), that can be computed from the reference model. Note that the MTCG needs to be computed only once and can be used to align all traces in the log. We propose an algorithm for computing alignments, which exploits the structure and is comparable with the state-of-the-art when computing alignments for small models and many log traces.

1.1. Contributions

This paper is an extended version of our earlier work [6]. We extend the work with the following contributions.

Milestones. We introduce the notion of *milestone activities*, which are events on which the log and the model must agree, i.e. if the event occurs in the log trace, it must be synchronized with the model. There may be several reasons for introducing milestone activities. Milestone activities need to be observed and cannot

be skipped. They serve as minimal requirements for an alignment, avoiding that an arbitrary path is taken. To illustrate the concept, reconsider Figure 1 and assume activity E to be a milestone activity, i.e. disallow alignments to include the pair $(\gg, \mathrm{t}_4)$. Now, the standard cost function (with milestones) would also result in the desired alignment as the shortcut through the model is now unavailable. An interesting result is that in some scenarios, milestones can make it impossible to align a trace with a model (i.e. in case an alignment must include the milestone activity as a model move).

Milestones help to address a major limitation of the cost function that maximizes synchronous moves, which occurs when the process model contains a large cycle. We discuss this in Section 3 and show that milestones can be used to prevent such cycles from being traversed carelessly (by marking a particular event on the cycle as a milestone). We extend the transitive closure graph to also include milestones, and call it a *Milestone Transitive Closure Graph* (MTCG) and show how alignments can be computed.

To evaluate our approach, we analyse differences between alignments with and without milestones on a set of generated models and log traces, and show how their computation times are affected on a set of industrial models with many log traces.

Transitive closure graph for analysing non-conforming behaviour. We also consider the MTCG from a diagnostic point of view. Visualizing alignments by projecting them on the reference model or its corresponding marking graph may not always be desired. An issue is that model moves are not observed in the log trace, but are 'generated' in the diagnostics. When fabricating events that did not really happen (i.e. model moves), timestamps need to be created. This may lead to misleading results. By mapping alignments on a MTCG we only depict synchronous moves, i.e. only moves that really happened, which are easier to interpret and trust. We discuss the advantages and disadvantages of this method.

2. Preliminaries

We assume that the reader is familiar with the basics of automata theory and Petri nets (otherwise, we refer to [24]). A sequence or trace is an ordered list of events, which we denote by $\sigma = \langle \sigma_0, \sigma_1, \ldots, \sigma_{|\sigma|-1} \rangle$. Two sequences are concatenated using the $\cdot$ operation. Given a sequence σ and a set of elements S, we refer to $\sigma \backslash S$ as the sequence without any elements from S, e.g. $\langle a, b, b, c, a, f \rangle \backslash \{b, f\} = \langle a, c, a \rangle$. For two sequences σ_1 and σ_2, we call σ_1 a *subsequence* of σ_2 (denoted with $\sigma_1 \sqsubseteq \sigma_2$) if σ_1 is formed from σ_2 by deleting elements from σ_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 σ_1 is a strict subsequence of σ_2, thus σ_1 contains fewer elements than σ_2.

A *multiset* (or bag) is an unordered set that may contain multiple instances of the same element. For example, $[a^2, b, c^3]$ is a multiset in which a occurs 2 times (we represent multiple occurrences of an element in superscript). Two multisets

are combined with a $\uplus$ operation, e.g. $[a^2, b, c^3] \uplus [b, c^2, d^2] = [a^2, b^2, c^5, d^2]$. We will use multisets to represent markings and event logs. In figures, we use a more concise notation to denote multisets, e.g. we may write $[a^2, b, c^3]$ as a^2bc^3. Given a set of elements S, we denote the set of all possible multisets as $\mathcal{B}(S)$, and its power-set by 2^S.

Traces are sequences $\sigma \in \Sigma^*$, for which each element is called an *event* and is contained in the alphabet Σ, also called the set of events. We globally define the alphabet Σ, which does not contain the skip event ($\gg$) nor the invisible action or silent event (τ). An *event log* $\mathrm{E} \in \mathcal{B}(\Sigma^*)$ is a multiset of traces.

2.1. Petri nets

Petri nets are a mathematical formalism that allow us to describe processes, they allow us to represent parallel behaviour in a relatively compact manner. Consider Figure 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. It is not allowed 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* $\mathrm{N} = (P, \mathrm{T}, F, \Sigma_\tau, \lambda, m_0, m_\mathrm{F})$ *such that:*

- P *is a finite set of* places*,*
- T *is a finite set of* transitions *such that* $P \cap \mathrm{T} = \emptyset$*,*
- $F \subseteq (P \times \mathrm{T}) \cup (\mathrm{T} \times P)$ *is a set of directed arcs, called the* flow relation*,*
- Σ_τ *is a set of activities, with* $\Sigma_\tau = \Sigma \cup \{\tau\}$*,*
- $\lambda : \mathrm{T} \to \Sigma_\tau$ *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. For instance, in Figure 1, p_0 contains a token (represented by a black dot), and the marking $[\mathrm{p}_0]$ represents the initial marking. 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, one token is removed from these places and all places with 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)**.** *The corresponding* marking graph *or* state-space *for a Petri net* $\mathrm{N} = (P, \mathrm{T}, F, \Sigma_\tau, \lambda, m_0, m_\mathrm{F})$ *is given by the deterministic automaton* $MG = (Q, \Sigma_\tau, \delta, q_0, q_\mathrm{F})$*, 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),*
- $\delta \subseteq (Q \times \mathrm{T} \times Q)$ *is the set of edges in MG, i.e.* $(m, t, m') \in \delta$ *iff there is a* $t \in \mathrm{T}$ *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.*

Given an edge $e = (m, t, m') \in \delta$ *with* $\lambda(t) = a$*, we write* $\lambda(e)$ *to denote* $\lambda(t)$*. We use the notation* $m \xrightarrow{a} m'$ *to represent an edge* e *for which* $\lambda(e) = a$[1]*. The source and target markings of edge* e *are respectively denoted by* $\texttt{src}(e)$ *and* $\texttt{tgt}(e)$*, thus we have* $\texttt{src}(e) = m$*, and* $\texttt{tgt}(e) = m'$*. We refer to the transition of* e *by* $\texttt{trans}(e) = t$*.*

Definition 3 (Path, language)**.** *Given a Petri net* N *and its corresponding marking graph* $MG = (Q, \Sigma_\tau, \delta, q_0, q_\mathrm{F})$*, a sequence of edges* $\rho = \langle e_0, e_1, \ldots, e_n \rangle \in \delta^*$ *is called a* path *in* N *if it forms a trace on the marking graph of* N*:* $\texttt{src}(e_0) = m_0 \wedge \texttt{tgt}(e_n) = m_\mathrm{F} \wedge \forall_{0 \leq i < n} : \texttt{tgt}(e_i) = \texttt{src}(e_{i+1})$*. The set of all paths in* N *is denoted by* $Paths(\mathrm{N})$*. We overload notation and write* $\lambda(\rho)$ *for referring to the sequence of labels visited in* ρ*, i.e.* $\lambda(\rho) = \langle \lambda(e_0), \lambda(e_1), \ldots, \lambda(e_n) \rangle$ *(there may be different paths* ρ *and* ρ' *such that* $\lambda(\rho) = \lambda(\rho')$*, due to e.g. multiple transitions with the same label). We define the* language $\mathcal{L}$ *of a Petri net* N *by* $\mathcal{L}(\mathrm{N}) = \{\lambda(\rho) \mid \rho \in Paths(\mathrm{N})\}$*. By extension, we also call a sequence of transitions* $\omega \in \mathrm{T}^*$ *a path if there is a path* $\rho = \langle e_0, e_1, \ldots, e_n \rangle$ *such that* $\omega = \langle \texttt{trans}(e_0), \texttt{trans}(e_1), \ldots, \texttt{trans}(e_n) \rangle$*, and we define* $\lambda(\omega)$ *to equal* $\lambda(\rho)$*.*

Definition 4 (Trace to Petri net)**.** *Given a trace* $\sigma = \langle \sigma_0, \sigma_1, \ldots, \sigma_{n-1} \rangle \in \Sigma^*$*, its corresponding Petri net is defined as* $\mathrm{N}_\sigma = (P, \mathrm{T}, F, \Sigma_\tau, \lambda, m_0, m_\mathrm{F})$ *with*

- $P = \{p_0, p_1, \ldots, p_n, p_{n+1}\}$,
- $\mathrm{T} = \{t_0, t_1, \ldots, t_n\}$,
- $F = \{(p_0, t_0), (p_1, t_1), \ldots, (p_n, t_n)\} \cup \{(t_0, p_1), (t_1, p_2), \ldots, (t_n, p_{n+1})\}$,
- $\Sigma_\tau = \bigcup_{0 \leq i < n} \{\sigma_i\}$, $\forall_{0 \leq i < n} : \lambda(t_i) = \sigma_i$,
- $m_0 = p_0$*, and* $m_\mathrm{F} = p_{n+1}$*.*

2.2. Alignments

With alignments we map observed behaviour from an event log onto the behaviour that is described in the reference model. An alignment is represented by a sequence of log-model pairs.

[1] There may be multiple distinct edges that are represented by $m \xrightarrow{a} m'$.

Definition 5 (Alignment)**.** *Let $\sigma \in \Sigma^*$ be a log trace and let* N *be a Petri net model such that* $\mathrm{N} = (P, \mathrm{T}, F, \Sigma_\tau, \lambda, m_0, m_\mathrm{F})$*, 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 $\tau \notin \Sigma_\gg$). We define $\mathrm{T}_\gg$ as the set of transitions and the skip event: $\mathrm{T}_\gg = \mathrm{T} \cup \{\gg\}$. Now, let $\gamma \in (\Sigma_\gg \times \mathrm{T}_\gg)^*$ be a sequence of log-model pairs. For $\gamma = \langle(\gamma_0^0, \gamma_0^1), (\gamma_1^0, \gamma_1^1), \ldots, (\gamma_{|\gamma|-1}^0, \gamma_{|\gamma|-1}^1)\rangle$, we define γ^ℓ as the log trace events, i.e. $\gamma^\ell = \langle\gamma_0^0, \gamma_1^0, \ldots, \gamma_{|\gamma|-1}^0\rangle \setminus \{\gg\}$ and γ^m as the events for the path through the model, by $\gamma^m = \langle\gamma_0^1, \gamma_1^1, \ldots, \gamma_{|\gamma|-1}^1\rangle \setminus \{\gg\}$. We refer to the corresponding sequence of actions from γ^m by $\lambda(\gamma^m)$, i.e. $\lambda(\gamma^m) = \langle\lambda(\gamma_0^m), \lambda(\gamma_1^m), \ldots, \lambda(\gamma_{|\gamma|-1}^m)\rangle$. We call γ an* alignment *if the following conditions hold:*

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

Definition 6 (Alignment cost)**.** *Let $\gamma \in (\Sigma_\gg \times \mathrm{T}_\gg)^*$ be an alignment for $\sigma \in \Sigma^*$ and the Petri net* N*. The cost function maps elements of γ onto alignment costs of individual moves, $c : (\Sigma_\gg \times \mathrm{T}_\gg) \to (\mathbb{R}_{\geq 0} \cup \{\infty\})$, and we overload c for assigning costs to alignments; $c : (\Sigma_\gg \times \mathrm{T}_\gg)^* \to (\mathbb{R}_{\geq 0} \cup \{\infty\})$, such that $c(\gamma) = \sum_{i=0}^{|\gamma|-1} c(\gamma_i)$.*

We call an alignment γ under cost function c a *successful alignment* if $c(\gamma) \neq \infty$, otherwise, if $c(\gamma) = \infty$, we call γ a *failed alignment*. We call a successful alignment γ *optimal* iff $\nexists \gamma' : c(\gamma') < c(\gamma)$, i.e. there does not exist an alignment γ' with a smaller cost.

Definition 7 (Standard cost function)**.** *The* standard cost function c_st *is defined for an alignment pair $(\ell, m) \in (\Sigma_\gg \times \mathrm{T}_\gg)$ as follows (assuming small $\varepsilon > 0$[2]):*

$$c_\mathrm{st}(\ell, m) = \begin{cases} \varepsilon & \ell = \gg \text{ and } m \in \mathrm{T} \text{ and } \lambda(m) = \tau \ \big(\textit{silent move}\big) \\ 0 & \ell \in \Sigma \text{ and } m \in \mathrm{T} \text{ and } \lambda(m) = \ell \ \big(\textit{synchronous move}\big) \\ 1 & \ell \in \Sigma \text{ and } m = \gg \ \big(\textit{log move}\big) \\ 1 & \ell = \gg \text{ and } m \in \mathrm{T} \text{ and } \lambda(m) \neq \tau \ \big(\textit{model move}\big). \end{cases}$$

In the standard cost function, we penalize log- and model moves equally. Synchronous moves are the preferred choice in alignments, since they are free. We note that silent moves also have a small cost, which is there in case there

[2] The value for ε is chosen such that cycles are prevented and an acyclic path with costs of 0 and ε (upper-bounded by the longest acyclic path in the marking graph) is preferred over a single 1-cost move. If a larger value is chosen for ε, more 1-cost moves may be taken in optimal alignments as a result. However, we assume that this is undesired and therefore set a sufficiently low value for ε in our experiments.

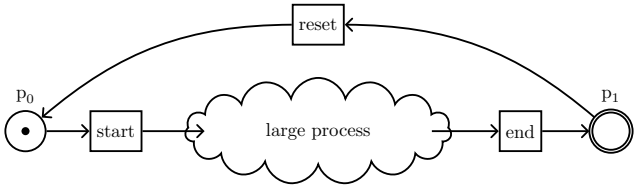


Figure 2: Example Petri net for which the c_{sync} cost function yields undesired results.

are so-called τ-cycles in the model (τ transitions in the model that compose a cycle in the marking graph). An alignment could otherwise contain an infinite sequence of silent moves, by continuously traversing these τ-cycles.

3. Maximizing Synchronous Moves and Milestones

The standard cost function from Definition 7 is the most commonly used cost function in literature [24, 28, 3, 1], though note that any cost function could be used (e.g. see [13] for an alternative). The standard cost function may lead to undesired results, as illustrated by the example from Figure 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_{sync} *for an alignment pair as follows (assuming small* $\varepsilon > 0$[2]*):*

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

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 ε 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 cost function is that observed behaviour is not sacrificed for shorter paths through the model (as Figure 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, especially in case the model contains cycles. Consider for example the Petri net model depicted in Figure 2. We are always able to synchronize all observed behaviour by executing the loop, i.e. the *reset* activity. By simply taking the *reset* transition one can traverse the model again to find a particular event, and report the steps to reach it as model moves. Therefore, we introduction the notion of milestone activities.

3.1. Milestones

To mitigate the problem with undesired alignments, we define a milestone as a label that may not be chosen in model moves. Note that we do not assign an infinite cost to log moves with milestone actions.

Definition 9 (Milestones)**.** *Given a Petri net* $\mathrm{N} = (P, \mathrm{T}, F, \Sigma_\tau, \lambda, m_0, m_\mathrm{F})$ *and a cost function* c*, milestones are labels* $Y \subseteq \Sigma$*, such that a new cost function* c' *is formed, which sets the cost for all model moves with a milestone label to infinity, i.e.*

$$\forall (\ell, m) \in (\Sigma_{\gg} \times \mathrm{T}_{\gg}) : c'(\ell, m) = \begin{cases} \infty & \text{if } \ell = \gg,\ m \in \mathrm{T}, \text{ and } \lambda(m) \in Y \\ c(\ell, m) & \text{otherwise.} \end{cases}$$

Consider the example model in Figure 2 and assume we compute an alignment with the c_{sync} cost function. If we choose the *reset* action to be a milestone, then the *reset* is only scheduled if it synchronizes with a log move. Therefore, an optimal alignment includes exactly as many *reset* actions in the model as there are in the log. Note that this is not necessarily the case if we employ a different cost function.

An interesting consequence of having milestones in the model, is that not every alignment is a successful one. If we assign the *start* action in Figure 2 as a milestone, and align it with a log trace that does not contain a *start* event, then the alignment cost would be infinite and therefore we call the alignment *failed*.

3.2. Relating the Model and Event Log

Given a Petri net model N and an event log $\mathrm{E} \in \mathcal{B}(\Sigma^*)$, we 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:** $\sigma_1 \in E : (\exists \sigma_2 \in \mathcal{L}(\mathrm{N}) : \sigma_1 = \sigma_2)$; the majority of log traces correspond to paths in the model. Then, these traces can be mapped onto the model by only using synchronous and silent moves, which is optimal for c_{st} and c_{sync}.
- **C2:** $\sigma_1 \in E : (\exists \sigma_2 \in \mathcal{L}(\mathrm{N}) : \sigma_1 \sqsubseteq \sigma_2)$; the majority of log traces correspond to subsequences of paths in the model. Then, these traces can be mapped onto the model without using any log moves. The example model from Figure 1 for $\sigma = \langle B \rangle$ is such an instance. We argue that c_{sync} provides better alignments in such instances as c_{st} may avoid synchronization in favour of shorter paths through the model.
- **C3:** $\sigma_1 \in E : (\exists \sigma_2 \in \mathcal{L}(\mathrm{N}) : \sigma_2 \sqsubseteq \sigma_1)$; for the majority of log traces there is a path through the model that forms a subsequence of the log trace. Then, these traces can be mapped onto the model without using any model moves. Here, c_{sync} and to some extent c_{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_{st} or c_{sync} could be preferred (which we discuss in Section 5.1),

We note that each of these cases could also be considered on a trace level, where e.g. we could have an event log that strictly contains log traces of the form C1 and C3. 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. That is, assuming that the model is correct, i.e. the model itself may also describe incorrect behaviour. 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 any more.

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

4. Preprocessing Reference Models

When constructing an alignment under the c_{sync} cost function, we 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 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. We extend the structure from [6] to include milestones, which we call a *Milestone Transitive Closure Graph*, or MTCG. Given a log trace, we can use dynamic programming to search for the maximum-length subsequence that is replayable in the MTCG. From this subsequence, we construct a path through the marking graph and obtain an optimal alignment.

4.1. Milestone Transitive Closure Graph (MTCG)

We construct an MTCG as described in Definition 10. Here, τ-edges are added for every edge in the marking graph, except for milestone edges (which cannot be skipped), such that, in case there are no milestones, every marking is reachable via τ-steps from the initial marking set. After determinization, for every path P in the original marking graph the MTCG contains a path P' such that $\lambda(P') = \lambda(P)$. In case there are no milestones, the MTCG also contains

every path P' for which $\lambda(P') \sqsubseteq \lambda(P)$ holds, that is, any subsequence of a path's labels can be formed in the MTCG.

Definition 10 (Milestone Transitive closure graph)**.** *Given a marking graph $MG = (Q, \Sigma_\tau, \delta, q_0, q_F)$ and a set of milestones Y, we first construct an extended marking graph $MG' = (Q, \Sigma_\tau, \delta', q_0, q_F)$, with*

$$\delta' = \delta \cup \{\big(\mathit{src}(e), \tau, \mathit{tgt}(e)\big) \mid e \in \delta \wedge \lambda(e) \notin Y\}.$$

A milestone transitive closure graph (MTCG) $\mathbb{M} = (\mathbb{Q}, \Sigma, \Delta, \mathbb{Q}_0, \mathbb{Q}_F)$ *is defined as the result of determinizing MG'. This can be achieved via a standard determinization algorithm [23]. The idea of a determinization algorithm is to perform τ-closures such that all unobserved behaviour is contracted to a single state. The initial state is formed by combining all reachable markings using only τ transitions (which consists of all markings in our case). Then, the successors of markings from this state are computed and combined for each label, followed by τ-closures on the successor states. This process is performed iteratively until no new successor states are found, such that*

- $\mathbb{Q} \subseteq 2^Q$,
- $\Sigma = \Sigma_\tau \setminus \{\tau\}$,
- $\Delta \subseteq (\mathbb{Q} \times \Sigma \times \mathbb{Q})$,
- $\mathbb{Q}_0 \subseteq Q$, *and*
- $\mathbb{Q}_F = \{F \mid F \in \mathbb{Q} \wedge q_F \in F\}$.

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

In Figure 3, we give an example of an MTCG with milestones B and D. Let us first consider the example without milestones. Because the marking graph consists of a single strongly connected component, every marking becomes reachable from every other marking. Therefore, if we compute its MTCG without any milestones, we obtain a single state with a self-loop for each label. Trivially, all paths and subsequences of paths are preserved in this structure, however, this is rather useless in practice.

If we add the milestones B and D to the model, we obtain a more interesting MTCG. An observation is that the initial state is no longer an accepting one, as we can only reach the $[\mathrm{p}_5]$ marking by firing a B and a D action. Other events can still be skipped, however, hence we can directly take a B action from the initial state to $\mathbb{Q}_2$, but taking an E action is not allowed. As a result, we ensure that paths through the MTCG may not skip milestone actions, and are forced to take these actions if there is no alternative path to reach a final state.

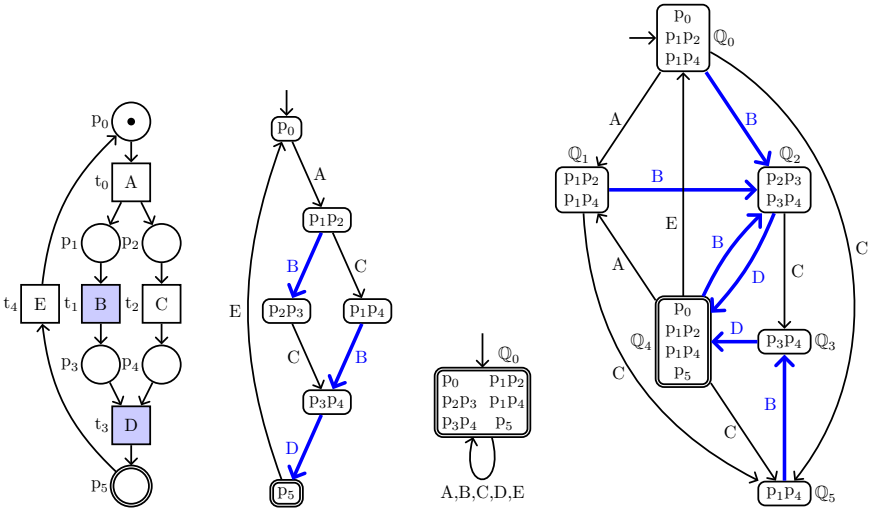


Figure 3: From left to right, a Petri net with milestone activities B and D, its corresponding marking graph, its corresponding MTCG without milestones, and the MTCG with milestones. All highlighted edges and transitions represent milestone activities.

4.2. Searching for optimal subsequences

Given a log trace σ, we use the MTCG of a model to obtain a subsequence of the log trace $\hat{\sigma} \sqsubseteq \sigma$, such that $\hat{\sigma}$ fully synchronizes with the model. From the construction of the MTCG we have the following property.

Lemma 1 (Paths in an MTCG synchronize with the model)**.** *Given a marking graph MG and its corresponding MTCG $\mathbb{M}$. If ρ is a path in $\mathbb{M}$, then there exists a path ρ' in MG such that $\lambda(\rho) \sqsubseteq \lambda(\rho')$.*

Proof. This follows from the construction of an MTCG. An MTCG $\mathbb{M}$ is formed from a marking graph MG by adding τ actions to existing edges. Therefore, the only additional behaviour that $\mathbb{M}$ has compared to MG is to replace visible actions by τ actions in its paths. Hence, any path in $\mathbb{M}$ forms a subsequence of a path in MG, i.e. $\forall \rho \in Paths(\mathbb{M}), \exists \rho' \in Paths(MG) : \lambda(\rho) \sqsubseteq \lambda(\rho')$. □

We use the result of Lemma 1 to search for a subsequence of a log trace that can fully synchronize with the model. For instance in the example of Figure 4 (without milestones), consider a log trace $\sigma = \langle F, D, E, B \rangle$. Note that this does not form a path through the model. The F event can be fired from $\mathbb{Q}_0$, after which the MTCG is in state $\mathbb{Q}_{10}$. From this state, it is not possible to perform any other event from log trace. A better choice is to skip the F event (which results in a log move) and form the subsequence $\langle D, E \rangle$, as highlighted[3]. We call the maximum-length subsequence $\hat{\sigma}$ from a log trace

[3] Note that after performing the D action in the MTCG, in the Petri net we have not yet

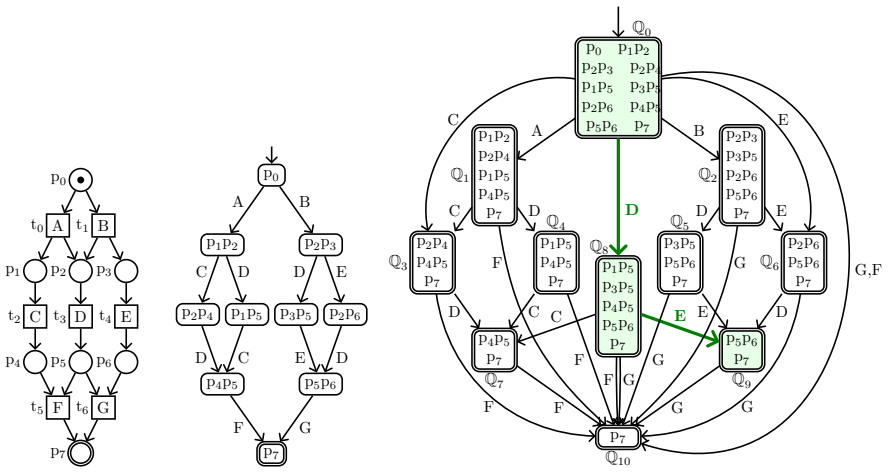


Figure 4: Example Petri net model (left), its corresponding marking graph (middle) and milestone transitive closure graph (right, without milestones) with the sequence $\langle D, E \rangle$ highlighted.

a *maximum fitting subsequence* if $\hat{\sigma}$ also forms a path through the MTCG, as defined in Definition 11.

Definition 11 (Maximum fitting subsequence)**.** *Given a sequence (log trace)* $\sigma \in \Sigma^*$ *and an MTCG* $\mathbb{M} = (\mathbb{Q}, \Sigma, \Delta, \mathbb{Q}_0, \mathbb{Q}_F)$*, then* $\hat{\sigma} \sqsubseteq \sigma$ *is a* maximum fitting subsequence *if and only if* $\hat{\sigma} \in \mathcal{L}(\mathbb{M}) \wedge \forall \hat{\sigma}' \sqsubseteq \sigma : \hat{\sigma}' \in \mathcal{L}(\mathbb{M}) \Rightarrow |\hat{\sigma}| \geq |\hat{\sigma}'|$.

Given a log trace σ and an MTCG $\mathbb{M} = (\mathbb{Q}, \Sigma, \Delta, \mathbb{Q}_0, \mathbb{Q}_F)$, we construct a maximum fitting subsequence $\hat{\sigma}$ by using dynamic programming to search for a subsequence of σ that forms a maximum-length path in $\mathbb{M}$. A straightforward implementation of this is by a 2D array ($|\sigma| \times |\mathbb{Q}|$) of sequences. We then iterate over each element σ_i from the log trace and track the paths in $\mathbb{M}$ that are formed by choosing to include σ_i or deciding to skip it (in case it is not a milestone). Subsequently we return the largest sequence that ends in a final state.

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 is achieved by using the MTCG and MG, by traversing backwards through $\hat{\sigma}$ as we show in Algorithm 1.

We assume a maximum fitting subsequence $\hat{\sigma} \neq \langle\rangle$ (we have a failed alignment if $\hat{\sigma} = \langle\rangle$). We first construct a path `MFP` from the subsequence $\hat{\sigma}$ (line 3), in the example from Figure 4 with $\hat{\sigma} = \langle D, E \rangle$ (see also Figure 5 for an illustration of the path construction process) this is $\texttt{MFP} = \langle(\mathbb{Q}_0, D, \mathbb{Q}_8), (\mathbb{Q}_8, E, \mathbb{Q}_9)\rangle$.

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.

Then in line 4, a backward search procedure (`BackwardsPath`) is called to search for a path P in the marking graph from an E-edge to the final marking ($[p_7]$).

The `BackwardsPath` 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 $F : Q \to (\delta \cup \{\bot\})$ 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 17-19 such that $\texttt{src}(e)$ is in S and not already visited. For every found 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 12-19, consider a predecessor W_i of m and search for edges directing to W_i. In this way, the search space is traversed backwards in a breadth-first manner, resulting in shortest paths to m.

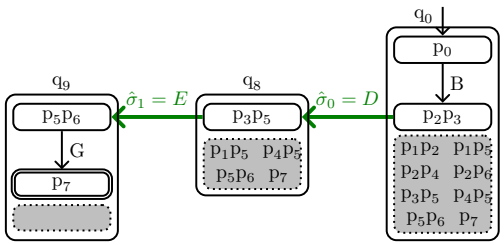
In line 13-16, the `BackwardsPath` procedure checks whether there is an edge $m' \xrightarrow{a} W_i$ for some m' (or an edge $q_0 \xrightarrow{a'} W_i$ for arbitrary a' in case $a = \bot$) and if so, constructs a path towards m in line 15 which is then returned. In the example, the path $\langle([p_3p_5], E, [p_5p_6]), ([p_5p_6], G, [p_7])\rangle$ will be returned for the first `BackwardsPath` call.

After the first `BackwardsPath` 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 in front of 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 $\bot$ to search for q_0 in the `BackwardsPath` call).

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

```
1  func PathConstr (𝕄 = (ℚ, Σ, Δ, ℚ_0, ℚ_F), MG = (Q, Σ_τ, δ, q_0, q_F), σ̂ = ⟨σ̂_0, σ̂_1, ..., σ̂_n⟩)
2      // Construct path MFP on MTCG such that λ(MFP) = σ̂
3      MFP := ⟨(ℚ_0, σ̂_0, S), (S, σ̂_1, S'), ..., (S'', σ̂_n, S''')⟩ s.t. ∀_{0≤i≤n} : MFP_i ∈ Δ
4      P := BackwardsPath(MG, q_F, σ̂_n, tgt(MFP_n)) // Path σ̂_n to q_F on MG
5      for i := n − 1; i ≥ 0; i := i − 1 do // Add paths from σ̂_i to σ̂_{i+1}
6          P := BackwardsPath(MG, src(P_0), σ̂_i, tgt(MFP_i)) · P
7      return BackwardsPath(MG, src(P_0), ⊥, ℚ_0) · P // Add path from q_0 to σ̂_0

8  func BackwardsPath (MG = (Q, Σ_τ, δ, q_0, q_F), m ∈ Q, a ∈ (Σ ∪ {⊥}), S ⊆ Q)
9      W := ⟨m⟩ // Sequence of unvisited markings in the backward search
10     ∀m ∈ S : F[m] := ⊥ // Mapping from markings to edges; F : Q → (δ ∪ {⊥})
11     i := 0
12     while i < |W| do // Continue for all markings in W
13         if ∃m' ∈ Q, a' ∈ Σ : (m', a', W_i) ∈ δ ∧ (a' = a ∨ (a = ⊥ ∧ m' = q_0)) then
14             P := ⟨(m', a', W_i)⟩ // Found path from a (or initial marking)
15             while tgt(P_{|P|−1}) ≠ m do P := P · F[tgt(P_{|P|−1})]
16             return P // Shortest path from a (or q_0) to m
17         forall e ∈ δ : src(e) ∈ (S \ W) ∧ tgt(e) = W_i do
18             W := W · ⟨src(e)⟩ // Add predecessor markings of m to W
19             F[src(e)] := e // Direct the source markings towards m
20         i := i + 1
21     return ⟨⟩ // No path from a (or q_0) is found (should never occur)
```



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

Figure 5: Path construction using Algorithm 1 on the example from Figure 4 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 γ is shown on the right.

In the example, we 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 $\hat{\sigma} \sqsubseteq \lambda(P)$, i.e.

$$P = \langle([\mathrm{p}_0], B, [\mathrm{p}_2\mathrm{p}_3]), ([\mathrm{p}_2\mathrm{p}_3], D, [\mathrm{p}_3\mathrm{p}_5]), ([\mathrm{p}_3\mathrm{p}_5], E, [\mathrm{p}_5\mathrm{p}_6]), ([\mathrm{p}_5\mathrm{p}_6], G, [\mathrm{p}_7])\rangle.$$

Alignment. 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. Regarding milestones, these cannot be taken as model moves, e.g. when constructing a path, because all milestone edges are explicitly present in the MTCG. Therefore, a milestone action is only taken if it is part of the maximum fitting trace. If the maximum fitting trace is empty, we have a failed alignment.

4.3. Limitations

We note that the MTCG algorithm does not exactly compute an alignment for the cost function c_{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 an alternative path through the model with a lower total cost (i.e. fewer model moves). This can be repaired by computing the alignments for all maximum fitting subsequences. If we consider a variant cost function c'_{sync}, where silent- and model moves have cost 0, then the MTCG algorithm does compute an optimal alignment.

If the marking graph contains cycles, the corresponding markings get contracted to a single state in the MTCG with a self-loop for each activity in the cycle (as shown in Figure 3), which may be prevented by adding milestones. The algorithm does prevent alignments to unnecessarily traverse cycles in the model (and thus self-loops in the MTCG).

Arguably, a limitation of our approach is that the user has to decide which activities should be marked as milestones. We remark that this process could

easily be automated, by e.g. selecting activities on cycles as milestones to prevent cycles from being collapsed (as in e.g. Figure 3).

Because each state in the MTCG consists of a subset of markings, there may be exponentially more states in the MTCG than there are markings in the marking graph. The construction of the MTCG relies on full knowledge of the marking graph, thus the MTCG cannot be constructed when the marking graph is of infinite size. As a result, this would suggest that the MTCG algorithm is mainly applicable for small-sized models (in combination with many log traces), and it may be ineffective for large models. However, since the size of the Petri net model is not a clear indicator for the size of its marking graph, we may have small models with a large marking graph and vice versa.

5. Experiments

The implementation for the MTCG algorithm is available in the `MaxSyncAlignments` package from the ProM 6.8 toolset and the results for the experiments are available at https://github.com/utwente-fmt/MaxSync-BPM2018. The corresponding models and event logs used in the experiments are available at https://data.4tu.nl/repository/uuid:5f168a76-cc26-42d6-a67d-48be9c978309.

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 containing many traces. All experiments were performed on an Intel® Core™ i7-4710MQ processor with 2.50GHz and 7.4GiB memory. For all experiments, we used a timeout of 60 seconds.

In Section 5.1 we investigate differences between the alignments resulting from using the standard- and max-sync cost functions, and the number of milestones. We also investigate the specific alignment problems for which events are only added or removed from log traces, i.e. cases C2 and C3 as discussed in Section 3.2. For all the above experiments we used generated Petri net models, each accompanied by a single log trace, for which we added various amounts of noise.

In Section 5.2 we also look at (small-scale) models that are accompanied by many log traces to compare the performance of the MTCG algorithm with related work. We compared our algorithm with the A* algorithm from [1] and the recent incremental alignment algorithm from Van Dongen [26]. However, we note that for the computation time comparison, the incremental alignment algorithm performs practically the same as the A* algorithm, as the experiments consist of many relatively small log traces. We therefore only present the results from the incremental algorithm. All algorithms have been implemented in ProM [30]. For each alignment computation we used a single thread[4].

[4] We consider multi-threaded experiments not as useful in this scenario, as the log traces can e.g. be divided over the different threads such that the alignments are computed independently.

Table 1: Average number of move types per alignment. Comparison between alignments generated using the c_{st} and c_{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_{st} cost function.

Moves	**Noise added (add, remove, swap)**								**Number of activities**						**Average**	
	10%		**30%**		**50%**		**70%**		**25**		**50**		**75**			
	c_{st}	c_{sync}	c_{st}	c_{sync}	c_{st}	c_{sync}	c_{st}	c_{sync}	c_{st}	c_{sync}	c_{st}	c_{sync}	c_{st}	c_{sync}	c_{st}	c_{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

5.1. Experiments using generated models and singleton event logs

Model generation. Using the PTandLogGenerator [12] 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 how alignments scale (the distribution of move types) 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 for c_{st} and c_{sync}. In Table 1 we compare the resulting alignments for the different cost functions. When comparing the overall results of c_{st} and c_{sync} (rightmost column), we observe that c_{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_{sync} from 3.2 times to 9.3 times as many compared to c_{st}. This only results in a few more synchronized moves than an alignment for a c_{st} cost function returns. The difference between log moves from c_{sync} and c_{st} stays relatively the same for increasing activities.

We conclude that, while c_{sync} does result in more synchronized moves than c_{st}, the number of additional model moves increases when the amount of noise, or the number of activities grows. As a result, the alignments resulting from c_{sync} and c_{st} become more diverse for larger models.

Alignment problems that only add or remove events. In Table 2, we compare the resulting alignments for adding or removing events. When inspecting the

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

Moves	**Log events added (`Add`)**									**Log events removed (`Rem`)**								
	10%			**30%**			**50%**			**10%**			**30%**			**50%**		
	c_{st}	c_{sync}	c_{add}	c_{st}	c_{sync}	c_{add}	c_{st}	c_{sync}	c_{add}	c_{st}	c_{sync}	c_{rem}	c_{st}	c_{sync}	c_{rem}	c_{st}	c_{sync}	c_{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

Table 3: Average number of move types per alignment. Comparison between alignments for the c_{sync} cost function on models that have either 2 or 5 randomly chosen milestones. `Failed` denotes the percentage of failed alignments.

	Noise added (add, remove, swap)								**Number of activities**						**Average**	
	10%		**30%**		**50%**		**70%**		**25**		**50**		**75**			
Milestones	2	5	2	5	2	5	2	5	2	5	2	5	2	5	2	5
Failed	2%	6%	4%	14%	8%	22%	12%	27%	4%	20%	10%	16%	4%	14%	6%	17%
Log	1.5	1.6	2.7	3.1	3.6	4.6	4.2	5.2	2.6	2.9	2.9	3.6	2.7	2.7	2.7	3.1
Model	2.2	2.0	4.2	3.2	5.4	3.9	5.8	4.3	2.8	2.0	4.3	3.6	6.2	4.7	4.0	3.1
Sync	19.6	19.6	15.2	14.0	11.9	10.8	9.4	9.1	10.7	10.8	15.5	14.7	18.7	16.5	13.9	13.4
Silent	12.4	12.0	11.5	10.6	10.8	9.6	10.1	9.4	8.5	8.2	11.1	10.0	18.5	16.3	11.2	10.5

`Add` case, we find that, in contrast to c_{sync}, c_{st} already avoids model moves for the most part, which is as we expected. Moreover, there are only small differences between alignments from c_{st} and c_{add}. For c_{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_{st} cost function is only partly able to describe the removal of events as it still chooses log moves. The c_{sync} cost function does not take any log moves as this maximizes the number of synchronous moves, making it equal to c_{rem}. When comparing c_{st} and c_{sync}, we argue that for the `Add` case, the c_{st} cost function better represents a 'correct' alignment and for the `Rem` case c_{sync} is better suited. For all cases, the effects become larger when the amount of noise is increased.

Influence of milestones on alignments. The results from Table 3 show how milestones affect alignments, and can be compared with the results from Table 1 for 0 milestones. The milestones were selected randomly (uniformly distributed over the activities). An interesting result is that even though we use the c_{sync} cost function, the number of model moves remains relatively low for both 2 and 5 milestones. For the alignments with milestones we observe that the number of activities does not significantly affect the ratio of failed alignments, even though a larger model should make it easier to 'avoid' a milestone. We do observe a rather clear correlation between the increasing amount of noise and the ratio of failed alignments. A possible explanation for this is that more noise in the log increases the probability that a milestone activity is removed. As expected,

having more milestones results in a larger alignment failure ratio.

When comparing the alignments resulting from the 2 and 5 milestone versions, we discover that these are remarkably similar. The largest difference is in the number of model moves, which is slightly lower for 5 milestones, which makes sense as we put a more severe restriction on which model moves can be taken. In general, only a few milestones are sufficient for mitigating the downsides of c_{sync}, as large detours through the model are now likely avoided.

5.2. Experiments using event logs with more traces

We now consider smaller models on which we 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. [11].

For our selection, we picked the 9 most interesting cases: the top models with the most number of places and transitions in the Petri net models, the largest number of markings and edges in the marking graph, the largest number of states and edges in the MTCG (for which we were able to compute the MTCG within 60 seconds), and models for which the computation of the marking graph and MTCG took longest (within 60 seconds).

For each model, we generated a set of 10, 100, and 1,000 log traces and added 30% noise by adding, removing, and swapping events. We also investigate the effects of having 0, 2, or 5 milestones. We compare the performance of the MTCG algorithm with the incremental algorithm from [26] using a single thread. Note that in our experiments, we only consider the c_{sync} cost function, as the MTCG algorithm is not applicable to the c_{st} cost function.

Results. The results are presented in Table 4. For the incremental algorithm we clearly see a computation time that is proportional to the log size. However, for the MTCG algorithm this is not the case. The computation time for aligning 10 log traces is practically the same when 1,000 log traces are aligned[5]. This shows us that after computing the MTCG as a precomputation step (which is included in the time and takes up most of it), actually aligning log traces takes very little time.

However, for the models b, c, and e, the MTCG is not able to compute alignments within 60 seconds, even for 10 log traces. Here, the size of the MTCG becomes too large. For each of the models for which MTCG timed out, the algorithm reported having explored more than 200,0000 different sets of markings (a set of marking forms a single state in the MTCG). This results from models that express concurrency or branching behaviour. Every allowed order for executing events in the model is tracked in the MTCG by storing all corresponding intermediate states (marking sets) that may be visited.

[5] In some cases we can even observe that MTCG performs faster for 100 traces than for 10 traces. However, this is caused by the randomness that is inherent to program execution.

Table 4: Comparison of computation times for aligning small models with a large number of log traces using our MTCG algorithm and the incremental algorithm from [26]. All times are in milliseconds. For every log trace we applied 30% noise by adding, removing and swapping events. Here, **M** is the model name and $|P|$ and $|T|$ denote the number of places and transitions in the model, respectively. The number of log traces is given by |**Log**| and |**Fail**| is the number of failed alignments. A **TO** indicates that the algorithm timed out (i.e. requires more than 60 seconds to compute).

				0 milestones		2 milestones			5 milestones		
M	$\|P\|$	$\|T\|$	**|Log|**	**MTCG**	**Incr**	**|Fail|**	**MTCG**	**Incr**	**|Fail|**	**MTCG**	**Incr**
a	35	26	10	723	288	2	465	346	7	327	138
			100	658	1,615	30	316	1,291	65	159	630
			1000	772	14,217	248	383	13,893	689	200	4,392
b	13	9	10	**TO**	23	2	764	17	6	68	6
			100	**TO**	224	37	728	142	79	80	35
			1000	**TO**	2,161	325	1,196	1,487	763	161	438
c	23	17	10	**TO**	102	1	**TO**	113	10	54	2
			100	**TO**	1,016	37	**TO**	778	77	59	182
			1000	**TO**	11,018	343	**TO**	8,966	696	130	3,289
d	40	26	10	310	82	0	127	60	0	29	55
			100	344	612	9	128	928	9	52	563
			1000	475	6,834	105	180	8,084	105	123	5,282
e	25	17	10	**TO**	281	1	**TO**	232	7	119	47
			100	**TO**	1,576	39	**TO**	1,343	78	163	266
			1000	**TO**	15,175	366	**TO**	11,926	790	285	2,745
f	59	43	10	602	260	0	174	274	3	131	796
			100	575	1,891	0	171	1,731	34	142	2,331
			1000	596	19,490	0	213	18,752	288	178	18,198
g	123	81	10	576	2,980	0	611	3,387	3	184	1,510
			100	575	29,136	0	638	22,235	29	201	18,992
			1000	690	**TO**	0	721	**TO**	285	253	**TO**
h	38	18	10	260	74	4	122	37	7	38	28
			100	271	503	28	130	310	58	40	171
			1000	293	4,540	256	156	2,911	583	93	1,677
i	24	18	10	103	62	3	15	55	9	6	6
			100	109	488	31	19	327	80	9	103
			1000	136	4,598	269	43	3,513	751	29	1,023

In particular for model g, we observe that the incremental algorithm already takes quite some time to compute alignments for 10 log traces, and times out for 1,000 traces. The MTCG algorithm is for this model significantly faster in computing alignments. An interesting aspect of this model is that, while the number of places and transitions is high, there is almost no branching in the process. The model has a few branching points, followed by long sequential processes. Arguably, this is a difficult situation for an on-the-fly algorithm such as the incremental algorithm or A*, as it may have to explore a large subprocess before realizing that a different choice at the start of the process would lead to a better alignment. Here, the MTCG remains reasonably small and for alignment computations we do not have to decide which path to take. We note that other approaches exist that address the limitation of exploring a large state-space, e.g. decomposition techniques [14, 19, 18], which may also be interesting to combine

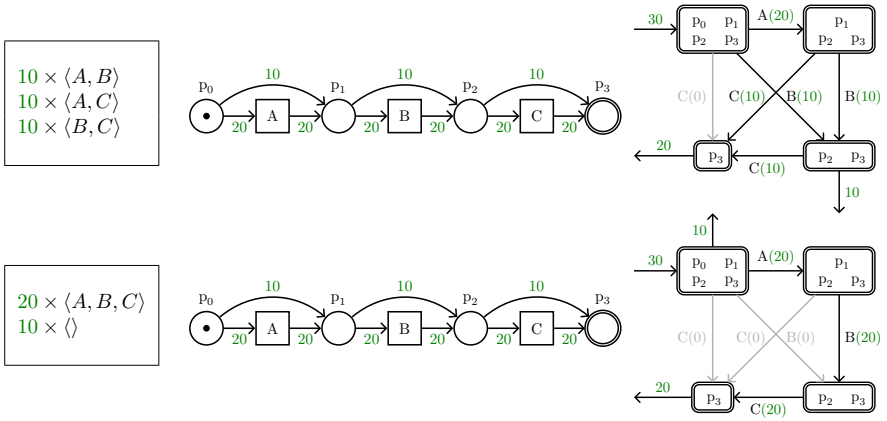


Figure 6: Two event logs (left) with their conforming and non-conforming behaviour displayed on a Petri net (middle) and its corresponding MTCG (right). The Petri net based visualization fails to show differences between the two markedly different event logs. The MTCG shows the differences much better.

with our technique.

Another interesting result is that the computation time decreases for both algorithms when we increase the number of milestones. The model becomes more constrained (we may not allow for particular model moves), hence there are fewer paths to explore in the model's state-space.

Overall, we observe that for 10 log traces, the incremental algorithm is favourable in terms of performance, but MTCG outperforms it from 100 log traces and beyond. Of course, the MTCG algorithm may only compute alignments for the c_{sync} cost function (with the inclusion of milestones), while the incremental algorithm (and A*) can compute alignments for arbitrary cost functions.

6. MTCG for Analysing Non-conforming Behaviour

Once alignments have been computed for a model and a set of log traces, we aim to obtain actionable insights from these alignments. For a single log trace, we can simply map the alignment on the Petri net or marking graph. However, this approach is not applicable for multiple alignments on a single model. Instead, two common ways of visualizing alignments are to (1) only depict the alignments, e.g. colour-coded for more convenient inspection, and (2) to map the alignments on the model such that the count or ratio of synchronized versus model moves is depicted on the model [8].

In Figure 6 (left and middle), we show how two sets of log traces are aligned on a Petri net model. We count the number of times that a particular transition is taken as a synchronous move (e.g. the arc denoting 20 from p_0 to A) and the number of times that a transition is skipped in the log, forming a model move (e.g. the arc from p_0 to p_1). This approach provides insights in the observed

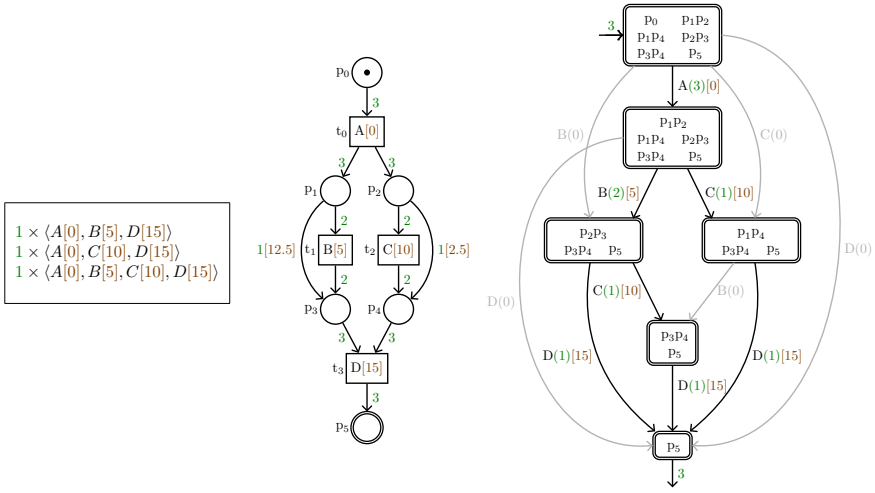


Figure 7: An event log with timestamps for each event (left) displayed on a Petri net (middle) and its corresponding MTCG (right). While the MTCG visualization only shows the observed events, the Petri net version needs to create time stamps for the generated model moves.

behaviour with respect to the model, e.g. each transition is fired an equal number of times and skipped in one third of the cases.

In Figure 7 we show how an event log with timing information is aligned to a Petri net model. Here, the timestamps are given for each event (e.g. B[5]), which are integrated into the model. In the MTCG we can clearly see how much time each event takes. This is also mostly true for the Petri net visualization. However, whenever an event is skipped in the model, we need to *generate* a model move along with a timestamp. For the first log trace, we could have constructed the following alignment:

$$\gamma = \begin{array}{|c|c|c|c|} \hline \text{A}[0] & \gg & \text{B}[5] & \text{D}[15] \\ \hline \text{A}[0] & \text{C}[2.5] & \text{B}[5] & \text{D}[15] \\ \text{t}_0 & \text{t}_2 & \text{t}_1 & \text{t}_3 \\ \hline \end{array}$$

where C[2.5] is a generated model action, with a timestamp that is the average of the time for the two neighbouring events (i.e. $\frac{0+5}{2}$). This approach may seem logical, but produces useless information. We could have also constructed an alignment in which events B and C are swapped, which leads to a different timestamp for C.

Drawbacks of mapping alignments on the model. There are some drawbacks to this approach. As we show in the picture, the results for two distinct event logs result in the same diagnostic information. Observe that a model move is a *generated* event. Hence, when applicable, all accompanying data for this event (e.g. the timestamp denoting when the event occurred) is generated as well, while in practice the event did not take place at all. By displaying model moves

we are effectively trying to 'fill the gap' between two synchronous moves, while in fact we observed that there is no gap to fill.

Depicting only synchronous moves. A useful property of the MTCG is that it allows us to represent all synchronous moves in an alignment, without having to generate artificial model moves to 'fill in the gaps'. Consider the rightmost illustrations from Figure 6. Here we only depict which moves in the model can be synchronized with the log traces. In the top image, we clearly see how the trace $\langle A, C \rangle$ is depicted. And, since we do not create artificial moves, the bottom illustration only depicts the 20 $\langle A, B, C \rangle$ traces (the $\langle\rangle$ traces remain in the initial state). Of course, here it is still possible to contrive examples in which two distinct event logs result in the same diagnostics, but the MTCG structure does mitigate that effect to some extent. The drawback is that for larger models it may become infeasible to represent alignments on a MTCG, simply because the graph becomes too big.

7. Related Work

One of the earliest works in conformance checking was from Cook and Wolf [10]. They compared log traces with paths generated from the model.

One technique to check for conformance is *token-based replay* [22]. 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 any more. However, token-based replay is faster to compute when compared to alignments, and it has been implemented in commercial tools like Celonis.

Alignments were introduced [25, 1] 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 [1, 29]. In earlier work [5] we presented an algorithm for computing alignments that relies on the symbolic computation of the marking graph. We showed that it is well-suited for large models, but its set-up time makes it not useful in computing alignments for many log traces [6]. However, our algorithm is able to efficiently handle a large number of log traces (albeit for small-scaled models).

For larger models, techniques have also been developed to decompose the Petri net in smaller subprocesses [19]. 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 [18]. It would be interesting to combine the MTCG algorithm with such decomposed models. Another approach is to combine the different log traces in a trie or directed acyclic automaton [20]. While we have not studied this in-depth, we

argue that this approach may combine well with the MTCG alignment procedure, as it allows us to compute multiple alignments at once. When combining the model with (a sample of) the event log, one may be able to approximate the part of the marking graph that is actually visited (as in [17]). While sacrificing optimality, such an approach may improve the applicability of conformance checking techniques.

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 [1, 27]. The MTCG approach that we introduced in this paper could also be suitable for computing prefix-alignments. Burattin and Carmona [7] introduced a technique similar to the MTCG 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.

When alignments are approximated, often only part of the state-space is considered, which may lead to overly optimistic results. This led to the concept of *anti-alignments* [9]. This is a trace that deviates as much as possible from the observed behaviour, with the goal to compensate for the optimistic view. Recent work by Bauer et al. [4] introduces a statistical approach for conformance checking with large quantities of event data. The idea is to only check for a small fraction (obtained by trace sampling) of the data and approximate the remaining results. While being an approximation, impressive results have been obtained.

An important (and arguably often neglected) aspect of conformance checking is to determine whether the detected discrepancies originate from the model or the event log. Rogge-Solti et al. [21] developed a framework that incorporates a 'trust value' to describe the quality of the model or event log, which can be applied to more accurately reflect the results to reality.

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 α is minimal [31].

8. Conclusion

In this paper, we considered a max-sync cost function that instead of minimizing discrepancies between a log trace and the model, maximizes the number of synchronous events. We empirically evaluated the differences with the standard cost function and compared the alignment computation times. The use of 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 is more applicable. Moreover, we introduce the notion of milestones, i.e. unskippable actions in the model. Milestones may be used to refine the model or guide the alignment computation. For instance, we observed that combining the max-sync cost function with a few milestones prevents alignments to take long deviations through the model. We consider milestones to be valuable in general as they allow a user to filter traces and steer alignment computations however he or she pleases.

On industrial models with many log traces, we showed that our new algorithm, which uses a preprocessing step on the model to compute a so-called milestone transitive closure graph or MTCG, is quite different to the state-of-the-art techniques for computing alignments. While it does not scale well for large models, it is able to handle many log traces with ease.

We conclude that the max-sync cost function as well as our MTCG structure and algorithm, are complementary to the standard techniques, as it provides a alternative views that may be preferable in some contexts. We also showed the added value of milestones, for improving and adapting the max-sync costs. Finally, we also show that we may use the MTCG structure for analysing non-conforming behaviour, such that we do not rely on constructing artificial model moves.

An interesting direction for future work is to focus on extending the applicability of the MTCG algorithm. When combined with a decomposition strategy, we may be able to compute alignments for significantly larger models. We may also consider approaches to reduce the number of visible actions in the model, by removing ones that are not interesting, and thereby reducing the size of the model. This way, diagnostic information can be represented more clearly.

Acknowledgement

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

References

[1] Arya Adriansyah. *Aligning observed and modeled behavior*. PhD thesis, Eindhoven University of Technology, The Netherlands, 2014.

[2] Arya Adriansyah, Natalia Sidorova, and Boudewijn F. van Dongen. Cost-based fitness in conformance checking. In *11th International Conference on Application of Concurrency to System Design, ACSD 2011*, pages 57–66. IEEE Computer Society, 2011.

[3] Arya Adriansyah, Boudewijn F. van Dongen, and Wil M. P. van der Aalst. Memory-efficient alignment of observed and modeled behavior. Technical report, 2013.

[4] Martin Bauer, Han van der Aa, and Matthias Weidlich. Estimating Process Conformance by Trace Sampling and Result Approximation. In *Proceedings*

of the 17th International Conference on Business Process Management, BPM 2019, pages 1–16, 2019.

[5] Vincent Bloemen, Jaco van de Pol, and Wil M. P. van der Aalst. Symbolically Aligning Observed and Modelled Behaviour. In *Proceedings of the 18th International Conference on Application of Concurrency to System Design, ACSD 2018*, pages 50–59. IEEE Computer Society, 2018.

[6] Vincent Bloemen, Sebastiaan J. van Zelst, Wil M. P. van der Aalst, Boudewijn F. van Dongen, and Jaco van de Pol. Maximizing Synchronization for Aligning Observed and Modelled Behaviour. In *Proceedings of the 16th International Conference on Business Process Management, BPM 2018*, volume 11080 of *Lecture Notes in Computer Science*, pages 233–249. Springer, 2018.

[7] Andrea Burattin and Josep Carmona. A framework for online conformance checking. In *Revised Papers of the Business Process Management Workshops, BPM 2017 International Workshops*, volume 308 of *Lecture Notes in Business Information Processing*, pages 165–177. Springer, 2017.

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

[9] Thomas Chatain and Josep Carmona. Anti-alignments in Conformance Checking - The Dark Side of Process Models. In Fabrice Kordon and Daniel Moldt, editors, *Proceedings of the 37th International Conference on the Application and Theory of Petri Nets and Concurrency, PETRI NETS 2016*, volume 9698 of *Lecture Notes in Computer Science*, pages 240–258. Springer, 2016.

[10] Jonathan E. Cook and Alexander L. Wolf. Software process validation: Quantitatively measuring the correspondence of a process to a model. *ACM Transactions on Software Engineering and Methodology*, 8(2):147–176, 1999.

[11] Dirk Fahland, Cédric Favre, Jana Koehler, Niels Lohmann, Hagen Völzer, and Karsten Wolf. Analysis on demand: Instantaneous soundness checking of industrial business process models. *Data & Knowledge Engineering*, 70(5):448–466, 2011.

[12] Toon Jouck and Benoît Depaire. PTandLogGenerator: A Generator for Artificial Event Data. In *Proceedings of the BPM Demo Track 2016*, volume 1789 of *CEUR Workshop Proceedings*, pages 23–27, 2016.

[13] Marie Koorneef, Andreas Solti, Henrik Leopold, and Hajo A. Reijers. Automatic Root Cause Identification Using Most Probable Alignments. In *Revised Papers of the 2017 Business Process Management Workshops*, volume 308 of *Lecture Notes in Business Information Processing*, pages 204–215. Springer, 2017.

[14] Wai Lam Jonathan Lee, H. M. W. Verbeek, Jorge Munoz-Gama, Wil M. P. van der Aalst, and Marcos Sepúlveda. Recomposing conformance: Closing the circle on decomposed alignment-based conformance checking in process mining. *Information Science*, 466:55–91, 2018.

[15] Maikel Leemans and Wil M. P. van der Aalst. Process mining in software systems: Discovering real-life business transactions and process models from distributed systems. In *Proceedings of the 18th International ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MoDELS 2015*, pages 44–53. IEEE Computer Society, 2015.

[16] Cong Liu, Boudewijn F. van Dongen, Nour Assy, and Wil M. P. van der Aalst. Component behavior discovery from software execution data. In *Proceedings of the 2016 IEEE Symposium Series on Computational Intelligence, SSCI 2016*, pages 1–8. IEEE, 2016.

[17] Jorge Munoz-Gama and Josep Carmona. A Fresh Look at Precision in Process Conformance. In Richard Hull, Jan Mendling, and Stefan Tai, editors, *Proceedings of the 8th International Conference on Business Process Management, BPM 2010*, volume 6336 of *Lecture Notes in Computer Science*, pages 211–226. Springer, 2010.

[18] Jorge Munoz-Gama, Josep Carmona, and Wil M. P. van der Aalst. Single-Entry Single-Exit decomposed conformance checking. *Information Systems*, 46:102–122, 2014.

[19] Artem Polyvyanyy, Jussi Vanhatalo, and Hagen Völzer. Simplified Computation and Generalization of the Refined Process Structure Tree. In *Revised Selected Papers of the 7th International Workshop on Web Services and Formal Methods, WS-FM 2010*, volume 6551 of *Lecture Notes in Computer Science*, pages 25–41. Springer, 2010.

[20] Daniel Reißner, Raffaele Conforti, Marlon Dumas, Marcello La Rosa, and Abel Armas-Cervantes. Scalable conformance checking of business processes. In *Proceedings of the International Conference on On the Move to Meaningful Internet Systems, OTM 2017*, volume 10573 of *Lecture Notes in Computer Science*, pages 607–627. Springer, 2017.

[21] Andreas Rogge-Solti, Arik Senderovich, Matthias Weidlich, Jan Mendling, and Avigdor Gal. In Log and Model We Trust? A Generalized Conformance Checking Framework. In Marcello La Rosa, Peter Loos, and Oscar Pastor, editors, *Proceedings of the 14th International Conference on Business Process Management, BPM 2016*, volume 9850 of *Lecture Notes in Computer Science*, pages 179–196. Springer, 2016.

[22] Anne Rozinat and Wil M. P. van der Aalst. Conformance checking of processes based on monitoring real behavior. *Information Systems*, 33(1):64–95, 2008.

[23] Thomas Sudkamp. *Languages and Machines : An Introduction to the Theory of Computer Science*. Addison-Wesley Longman Publishing Co., Inc., 1988.

[24] Wil M. P. van der Aalst. *Process Mining: Data Science in Action*. Springer, 2016.

[25] Wil M. P. van der Aalst, Arya Adriansyah, and Boudewijn F. van Dongen. Replaying history on process models for conformance checking and performance analysis. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 2(2):182–192, 2012.

[26] Boudewijn F. van Dongen. Efficiently Computing Alignments - Using the Extended Marking Equation. In *Proceedings of the 16th International Conference on Business Process Management, BPM 2018*, volume 11080 of *Lecture Notes in Computer Science*, pages 197–214. Springer, 2018.

[27] Sebastiaan J. van Zelst, Alfredo Bolt, Marwan Hassani, Boudewijn F. van Dongen, and Wil M. P. van der Aalst. Online conformance checking: relating event streams to process models using prefix-alignments. *International Journal of Data Science and Analytics*, 2017.

[28] Sebastiaan J. van Zelst, Alfredo Bolt, and Boudewijn F. van Dongen. Tuning Alignment Computation: An Experimental Evaluation. In *Proceedings of the International Workshop on Algorithms & Theories for the Analysis of Event Data, ATAED 2017*, volume 1847 of *CEUR Workshop Proceedings*, pages 6–20. CEUR-WS.org, 2017.

[29] Sebastiaan J. van Zelst, Alfredo Bolt, and Boudewijn F. van Dongen. Computing Alignments of Event Data and Process Models. *Transactions on Petri Nets and Other Models of Concurrency*, 13:1–26, 2018.

[30] Erik Verbeek, Joos C. A. M. Buijs, Boudewijn F. van Dongen, and Wil M. P. van der Aalst. Xes, xesame, and prom 6. In *Selected Extended Papers of Information Systems Evolution - CAiSE Forum 2010*, volume 72 of *Lecture Notes in Business Information Processing*, pages 60–75. Springer, 2010.

[31] Robert A. Wagner. Order-n Correction for Regular Languages. *Communications of the ACM*, 17(5):265–268, 1974.