Symbolically Aligning Observed and Modelled Behaviour

Vincent Bloemen
University of Twente
Enschede, The Netherlands
v.bloemen@utwente.nl

Jaco van de Pol
University of Twente
Enschede, The Netherlands
j.c.vandepol@utwente.nl

Wil M.P. van der Aalst
RWTH Aachen University
Aachen, Germany
wvdaalst@pads.rwth-aachen.de

***Abstract*—Conformance checking is a branch of process mining that aims to assess to what degree a given set of log traces and a corresponding reference model conform to each other. The state-of-the-art approach in conformance checking is based on the concept of alignments. Alignments express the observed behaviour in terms of the reference model while minimizing the number of mismatches between the event data and the model. The currently known best algorithm for constructing alignments applies the A* shortest path algorithm for each trace of event data.**

In this work, we apply insights from the field of model checking to aid conformance checking. We investigate whether alignments can be computed efficiently via symbolic reachability with decision diagrams. We designed a symbolic algorithm for computing shortest-paths on graphs restricted to 0- and 1-cost edges (which is typical for alignments).

We have implemented our approach in the LTSMIN model checking toolset and compare its performance with the A* implementation supported by ProM. We generated more than 4000 experiments (Petri net model and log trace combinations) by setting various parameters, and analysed performance and related these to structural properties.

Our empirical study shows that the symbolic technique is in general better suited for computing alignments on large models than the A* approach. Our approach is better performing in cases where the size of the state-space tends to blow up. Based on our experiments we conclude that the techniques are complementary, since there is a significant number of cases where A* outperforms the symbolic technique and vice versa.

***Index Terms*—conformance checking, process mining, model checking, symbolic reachability, alignment, algorithm, graph search**

I. 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 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 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 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 form a correct path through the model.

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

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 log in an alignment:

1) We are unable to map observed behaviour in the event log onto the model (a *log-move*).
2) An action in the underlying model is needed, yet this is not reflected in the log (a *model-move*).
3) A *synchronous move* in which both the model and the log perform the same event.
4) A *silent-move* in which the model performs a silent or invisible action (denoted with τ).

Consider the example given in Figure 1. The Petri net represents the product of an event log $\langle A, C, E, B \rangle$ (depicted in orange) and the model, which is portrayed in blue (model-moves) and grey (silent-moves). The green transitions illustrate synchronous moves. An alignment is a path from the initial marking on the product (`p0q0`) to the final marking (`p7q4`), which is given by γ. Here, the synchronous move $|\frac{A}{A}|$ is fired to obtain the marking `p1q1`. This is followed by a model-move, $|\frac{\gg}{B}|$, to obtain the marking `p2p3q1`. Here, $\gg$ denotes the *skip* action to indicate that nothing happens in the log. A silent-move is given by $|\frac{\gg}{\tau}|$ and the last pair is the log-move $|\frac{B}{\gg}|$, in which the model takes no action.

An alignment is *optimal* if it minimizes a given cost function. In the example, γ is optimal for the so-called *standard-cost* function, which assigns a cost of 0 to synchronous and silent-moves, and assigns a cost of 1 to model- and log-moves. The goal is to search for optimal alignments, which is an NP-hard problem. In practice, an optimal alignment is found by

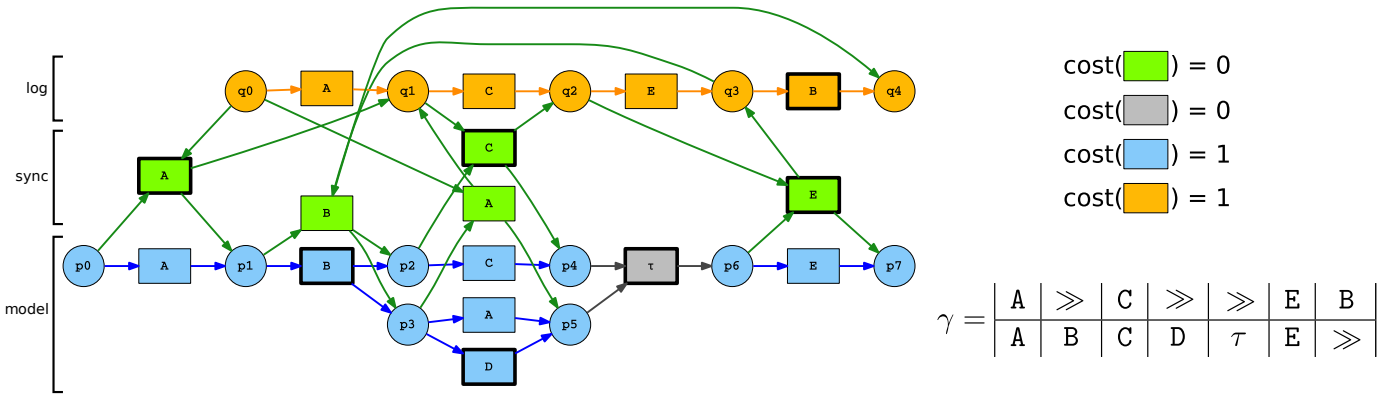

Fig. 1: Example Petri net model (blue) combined with a log trace (orange), with synchronous actions in green. The right part shows the standard cost function and an optimal alignment γ using that cost function for the model and trace.

computing the shortest path on the state-space of the product Petri net.

The state-of-the-art technique uses an A* [7] approach to find such optimal alignments. Several optimizations are used to improve the performance of the algorithm, such as computing the state-space on-the-fly. The marking equation of Petri nets is exploited to prune the search space (remove states that cannot reach the final state any more) and the marking equation can also be used as a good heuristic function for A* [8].

However, searching for alignments in an explicit-state fashion can still take a long time for large state-spaces. In the field of model checking, such a 'state-space explosion' is a common and well-studied problem [9]. One of the techniques to deal with large state spaces is to employ *symbolic model checking* [10]. A symbolic representation in the form of decision diagrams is used to efficiently store a set of states. This is combined with operations such as intersect, unite, and a symbolic next-state function to explore the state-space. Especially in structured models[1], symbolic reachability is generally able to explore state-spaces that are orders of magnitudes larger than explicit techniques are capable of [10], [11].

To the best of our knowledge, while there exist symbolic algorithms for computing shortest paths [12], [13], there is as of yet no symbolic algorithm to compute alignments. Since Petri nets tend to have structured state-spaces, a symbolic solution may be well-suited for computing alignments. In this paper we present a symbolic algorithm for computing alignments and compare its performance on over 4,000 experiments that exhibit various characteristics. We show that our symbolic algorithm is favourable for larger models when compared to the A* approach, especially in cases where the state-space of the Petri net tends to blow up.

[1]With a structured model, we mean that a model state can be encoded in compact manner, by e.g., using few variables and exploiting locality and symmetries. This keeps the decision diagrams small, which correspondingly reduces the memory and time usage per operation compared to an unstructured model.

A. Contributions

Our contributions are as follows:

- We present a novel algorithm for computing alignments and shortest-paths symbolically, which is applicable for all uniform-cost functions (that only assign a 0 or 1 as edge-costs).
- We generated a set of 4,320 benchmark experiments (Petri net models and log trace combinations) with varying parameters such that structural properties can be analysed.
- We implemented our approach in the LTSMIN model checking toolset and compare its performance with the A* implementation from the process mining tool ProM on the benchmark instances.
  - We observe that when comparing average compute times, the symbolic algorithm significantly outperforms A*.
  - For smaller models, A* computes alignments faster than the symbolic algorithm.
  - We observe that our algorithm is especially favourable for larger instances, where the state-space tends to blow up (e.g., due to parallel transitions, OR-branches, and increases in activities in the Petri net model).
  - We observe that our technique is complementary to A*, since there are cases where A* significantly outperforms the symbolic technique and vice versa.

B. Outline

The remainder of this paper is structured as follows. In the next section, we introduce relevant preliminaries. In Section III we present and discuss our algorithm. Details on the implementation are provided in Section IV. In Section V we discuss the experimental results. The related work is discussed in Section VI and we conclude in Section VII.

II. Preliminaries

We assume that the reader is familiar with basic results from 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 σ and a set of elements S, we refer to $\sigma \setminus S$ as the sequence without any elements from S, e.g., $\langle a, b, b, c, a, f \rangle \setminus \{b, f\} = \langle a, c, a \rangle$. Log traces are sequences, 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). Given a set S, we denote the set of all possible multisets as $\mathcal{B}(S)$, and its power-set by 2^S.

A. Preliminaries on Petri nets

Definition 1 (Petri net, marking)**.** *A* Petri net *is a tuple* $\mathcal{N} = (P, T, F, \Sigma_\tau, \lambda, m_0, m_\mathrm{F})$ *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*,*
- Σ_τ *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.*

Definition 2 (Marking graph)**.** *Given a Petri net* $\mathcal{N} = (P, T, F, \Sigma_\tau, \lambda, m_0, m_\mathrm{F})$*, the corresponding* marking graph *or* state-space $\mathcal{M} = (Q, \Sigma_\tau, \delta, q_0, q_\mathrm{F})$ *is a non-deterministic automaton such that:*

- $Q \subseteq \mathcal{B}(P)$ *is the (possibly infinite) set of vertices in* $\mathcal{M}$*, which corresponds to the set of* reachable *markings from* m_0 *(obtained by firing transitions),*
- $\delta \subseteq (Q \times \Sigma_\tau \times Q)$ *is the set of edges in* $\mathcal{M}$*, i.e.,* $(m, a, m') \in \delta$ *iff there is a* $t \in T$ *such that* m' *is obtained by firing transition* t *from marking* m *and* $\lambda(t) = a$*,*
- $q_0 = m_0$ *is the initial state of the graph,*
- $q_\mathrm{F} = m_\mathrm{F}$ *is the final state of the graph.*

We call sequence of edges $\mathcal{P}$ *an (accepting) path iff* $\mathcal{P}$ *starts from the initial state and ends in the final state, and for every two successive edges, the endpoint of the first edge is the starting point of the second edge.*

B. Preliminaries on alignments

Definition 3 (Alignment)**.** *Let* $\sigma \in \Sigma^*$ *be a log trace and let* $\mathcal{N}$ *be a Petri net model, for which we obtain the marking graph* $\mathcal{M} = (Q, \Sigma_\tau, \delta, q_0, q_\mathrm{F})$*. 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. 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* $\gamma^\ell = \langle \gamma_0^0, \gamma_1^0, \ldots, \gamma_{|\gamma|-1}^0 \rangle \setminus \{\gg\}$ *and* γ^m *by* $\gamma^m = \langle \gamma_0^1, \gamma_1^1, \ldots, \gamma_{|\gamma|-1}^1 \rangle \setminus \{\gg\}$*. We call* γ *an* alignment *if the following conditions hold:*

1) $\gamma^\ell = \sigma$ *(the activities of the log-part, equals to* σ*),*
2) $m_0 \xrightarrow{\gamma_0^m} m' \xrightarrow{\gamma_1^m} \ldots \xrightarrow{\gamma_{|\gamma^m|-1}^m} m_\mathrm{F}$ *(*γ^m *forms a path),*
3) $\forall a, b \in \Sigma \wedge a \neq b : (a, b) \notin \gamma$ *(illegal moves),*
4) $(\gg, \gg) \notin \gamma$*, (the 'empty' move may not exist in* γ*).*

Definition 4 (Alignment cost)**.** *Let* $\gamma \in (\Sigma_\gg \times \Sigma_{\tau\gg})^*$ *be an alignment for* $\sigma \in \Sigma^*$ *and the Petri net* $\mathcal{N}$*. We define the cost function* c *for pairs of* γ*;* $c : (\Sigma_\gg \times \Sigma_{\tau\gg}) \to \mathbb{R}_{\geq 0}$ *and overload* c *for alignments;* $c : \gamma \to \mathbb{R}_{\geq 0}$*, which we define as follows:* $c(\gamma) = \sum_{i=0}^{|\gamma|-1} c(\gamma_i)$*.*

We call an alignment γ *under cost function* c optimal *iff* $\nexists \gamma' : c(\gamma') < c(\gamma)$*, i.e., there does not exist an alignment with a smaller cost. We call a cost function* uniform-cost *iff* $c : (\Sigma_\gg \times \Sigma_{\tau\gg}) \to \{0, 1\}$*.*

Definition 5 (standard cost function)**.** *The* standard cost function c_st *is a uniform-cost function and is defined for an alignment pair as follows:*

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

For the remainder of the paper, we use the standard-cost function, unless stated otherwise.

C. Preliminaries on symbolic reachability

Definition 6 (Binary decision diagram)**.** *An (ordered)* Binary decision diagram (BDD) *[14] is a rooted directed acyclic graph that represents a boolean function. Each decision node is labelled by a boolean variable* v_i *and has two child nodes: a low child and a high child, representing the assignment of respectively False and True to* v_i*. A BDD has two terminal nodes, 0 and 1, and a path from the root to a 0 (or 1) terminal*

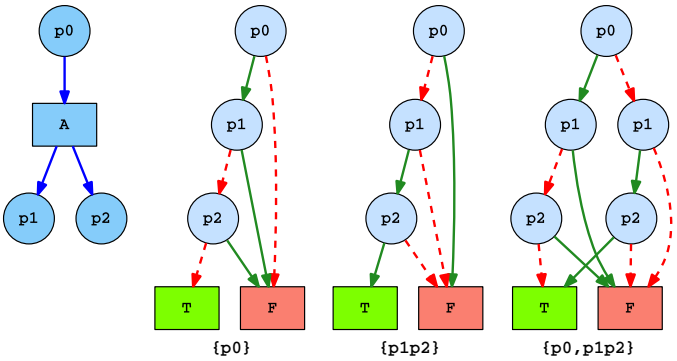

Fig. 2: From left to right: A simple Petri net model, a BDD representing the state `p0`, a BDD representing the state `p1p2`, and a BDD representing the union of states `p0` and `p1p2`. Here, green solid arrows represent assignments to True and red dotted arrows are assignments to False.

represents a variable assignment for which the represented boolean function evaluates to False (or True, respectively).

A set of Petri net markings can be represented as a BDD by encoding the token count of each place as (possibly multiple) boolean variables. Figure 2 gives an example of how a simple 1-safe Petri net (i.e., every place can have at most one token) can be encoded as a BDD. Here, a single boolean variable is assigned for every place in the Petri net. In case the Petri net is not 1-safe, multiple boolean variables are required for representing a set of Petri net markings.

The size (number of nodes) of a BDD is in the worst-case exponential in the number of boolean variables. In practice, the size greatly depends on the variable ordering, i.e., choosing to decide on variable v_i before deciding v_j may greatly reduce the BDD size as a result of locality or symmetries.

Set operations such as $A \cup B$ are possible with BDDs, an example of this is given in Figure 2. It is also possible to compute successors, by encoding a transition relation in the form of a BDD. This relation then checks (part of) a variable assignment and, by using auxiliary variable nodes, sets the corresponding assignment for the successor. This makes it possible to perform reachability in a symbolic fashion. However, note that most operations on BDDs are expensive in the sense that they have linear or quadratic complexity in the number of nodes, and exponential in the number of variables.

III. Symbolic Alignment

We first discuss the main idea of our symbolic alignment algorithm and an improvement to the base version. Then, the detailed algorithm is explained, an example of its usage is given and we show its correctness.

A. General idea

The idea of the algorithm is to split up the Petri net transitions into two groups, a group that only consists of 0-cost moves (in the standard cost function: τ-moves and synchronous moves) and 1-cost moves (in the standard cost function: model-moves and log-moves). We call these groups respectively $\mathtt{T}_0$ and $\mathtt{T}_1$. Note that this is only applicable for uniform-cost functions, thus our algorithm is only applicable in that setting.

A decision diagram is used to symbolically store a set of markings. Moreover, it is possible to apply transitions on such sets to compute a new set that contains all successors for every marking in the original set.

With the $\mathtt{T}_0$ and $\mathtt{T}_1$ groups, we can compute a shortest path from the initial marking to the final marking by first considering all reachable markings when we repeatedly apply $\mathtt{T}_0$ transitions. We thus create a transitive closure with $\mathtt{T}_0$ operations, which we denote as $\mathtt{T}_0{}^*$. After applying $\mathtt{T}_0{}^*$ on the initial marking we obtain the set S_1 of all markings that are reachable by only applying 0-cost steps. Then, on the set S_1 we apply a *single* $\mathtt{T}_1$ step, for which we obtain the set S_2. On S_2 we again apply $\mathtt{T}_0{}^*$ to obtain S_3, which is the set of markings that can be reached from the initial states with at most 1 cost.

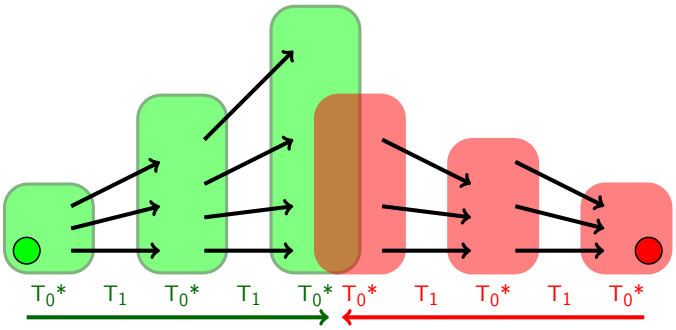

Fig. 3: Abstract representation for the procedure of Algorithm 1, where the green and red nodes respectively represent the initial and final markings. Coloured regions denote sets of markings and the brown region is the intersection of the forward and backward search.

The previous process is repeated until the smallest n such that S_n contains the final marking. Once this is reached, a path can be constructed from the final marking to the initial marking by reverting the symbolic operations. Note that this path is also a shortest path (with minimal cost) from the initial marking to the final marking, where the cost equals the number of $\mathtt{T}_1$ transitions taken. Also note that we assume that the final marking is reachable via a finitely many steps from the initial marking, with the consequence that each set S_i contains a finite number of markings.

B. Bidirectional search

Instead of just searching from the initial marking to the final marking, we can also search from the final marking to the initial marking by taking the transitions in the opposite direction. When the forward and backward search have a non-empty intersection, a shortest path can be constructed. A shortest path is formed by selecting a marking from the non-empty intersection and reverting the symbolic operations from this marking to get back to the initial and final marking.

The illustration from Figure 3 depicts the procedure. The intuition for searching in both directions is that the total number of visited states (which may relate to the number of decision nodes) is reduced this way. In our preliminary experiments we found that it is more time- and memory efficient to search bidirectionally compared to only searching in the forward direction.

Instead of switching search directions after each iteration, preliminary experiments showed that it is more beneficial to continue searching on the set that has the smallest number of nodes in the decision diagrams. The algorithm first performs a $\mathtt{T}_0{}^*$ step in both directions, after which it applies a $\mathtt{T}_1$ transition followed by a $\mathtt{T}_0{}^*$ application on the set with the smallest number of nodes. It then again performs a $\mathtt{T}_1$-$\mathtt{T}_0{}^*$ step on the set with the smallest number of nodes until it terminates. In general, the number of nodes in the decision diagram correlates with the time required for performing a symbolic operation.

Algorithm 1: Symbolic alignment algorithm

```
1  function DoTrans (Cur, T_X, dir)
2      N := ∅
3      forall t ∈ T_X do
4          if dir = FWD then N := N ∪ Next(Cur, t)
5          else N := N ∪ Prev(Cur, t)
6      return N
7  function T0Closure (Cur, T_0, V_FWD, V_BWD, dir)
8      N := Cur
9      while N ≠ ∅ ∧ V_FWD ∩ V_BWD = ∅ do
10         N := DoTrans(Cur, T_0, dir) \ V_dir
11         V_dir := V_dir ∪ N
12         Cur := N
13     return V_dir
14 function SymAlign (init, final, T_0, T_1)
15     V_FWD := {init}; V_BWD := {final} // Visited
16     V_FWD := T0Closure(V_FWD, T_0, V_FWD, V_BWD, FWD)
17     V_BWD := T0Closure(V_BWD, T_0, V_FWD, V_BWD, BWD)
18     if V_FWD ∩ V_BWD ≠ ∅ then return trace
19     do
20         if NC(V_FWD) < NC(V_BWD) then dir := FWD
21         else dir := BWD // dir ∈ {FWD, BWD}
22         N := DoTrans(V_dir, T_1, dir) \ V_dir
23         V_dir := V_dir ∪ N
24         V_dir := T0Closure(N, T_0, V_FWD, V_BWD, dir)
25         if V_FWD ∩ V_BWD ≠ ∅ then return trace
26     while N ≠ ∅
27     return no-trace
```

C. Detailed algorithm

Here we discuss Algorithm 1. We first discuss the auxiliary functions DoTrans, which computes the successors after applying either a T_0 or T_1 transition, and T0Closure, which computes the transitive closure for T_0 transitions.

Given a set of current states Cur, the DoTrans function attempts to fire all transitions in T_X (which is either T_0 or T_1) to form a set of successor or predecessor states N, which is returned. The for loop in line 3-5 iterates over all transitions in T_X and depending on the current direction (dir) computes the successors or predecessors from Cur by attempting to fire the transition t on all markings in Cur. The set of all successors is then stored in N and the union of all successors is returned.

The T0Closure function repeatedly calls the DoTrans function to compute T_0^*, i.e., the transitive closure of applying T_0 transitions. The set of successors N is computed in line 10 and used as the current set in the following iteration. Note that already visited states are removed (with the $\setminus V_{dir}$ operation) and newly found states are added to V_{dir}. The function returns when either there are no new successors ($N = \emptyset$) or when a shortest path is found ($V_{FWD} \cap V_{BWD} \neq \emptyset$) which is discussed later.

The main function, SymAlign works as follows. The forward and backward visited sets are initialized in line 15. From the initial state, the T0Closure is called to perform a forward T_0^* application. The set V_{FWD} now contains all states reachable from the initial state via T_0 transitions.

The T0Closure is also called from the final state in the backwards direction (line 17). It may be possible that an optimal alignment exists with a total cost of 0, then the final state should be included in V_{FWD} and hence $V_{FWD} \cap V_{BWD} \neq \emptyset$. If this is the case, then a trace can be reconstructed and returned (line 18). See Section IV-C for more information on the trace reconstruction.

In line 19-26 the algorithm continuously performs a T_1 transition followed by a T_0^* application until it either found and returned a shortest path or found no new states with a T_1 transition.

In line 20-21 the algorithm decides on which direction to search. The number of decision diagram nodes for V_{FWD} and V_{BWD} are compared (with the NC function) and the one with the fewest number of nodes is chosen for the current iteration.

The T_1 successors or predecessors are computed for all states in V_{dir} in line 22 (note that N only contains *new* states). The visited set is updated and a T_0^* application is performed. If there is a non-empty intersection between the forward and backward search, a shortest path is found and returned (line 25). Otherwise, the algorithm continues with a next iteration.

If there is no path from the initial to the final state, the entire state space gets explored (assuming this is finite) and no new states can be found, after which the loop ends and the function returns no-trace. Note that the two while-loops always terminate on finite state-spaces, either when a path to the final marking is found, or when no new successors are found after trying to fire transitions from T_0 or T_1.

D. Correctness

To show that Algorithm 1 is correct, we prove that the algorithm always returns a trace if there is a path from the initial to the final marking (Theorem 1), and that a returned trace forms an optimal alignment (Theorem 2).

Theorem 1 (Completeness)**.** *Given a marking graph $\mathcal{M} = (Q, \Sigma_\tau, \delta, q_0, q_F)$ and a uniform-cost function c, if there is a finite path from q_0 to q_F, then Algorithm 1 returns a trace for $\mathcal{M}$ and c.*

Proof. Assume that there is a path from q_0 to q_F and assume that the algorithm does not return a trace. We consider two cases: (1) The algorithm terminates and does not return a trace, and (2) the algorithm does not terminate.

(1) The algorithm terminates, thus it has finished exploring the state-space. Note that $T_0 \cup T_1$ forms the set of all transitions and that the algorithm performs a T_1 and T_0^* application in each step (aside from the first one). Therefore, both the forward and backward search have visited the respective forward and backward reachable state-spaces. Since there is a path from the initial to the final state, there must be a non-

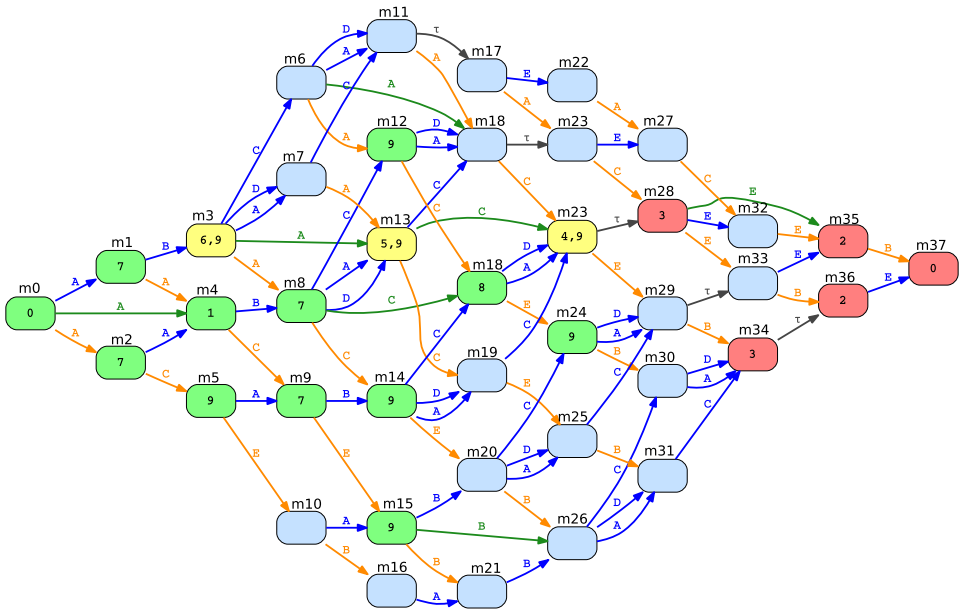

Fig. 4: Marking graph for the example of Figure 1. The numbers in the nodes denote the order of exploring the marking graph by Algorithm 1. Green nodes denote a forward search, red nodes denotes a backward search, and yellow nodes denote the intersection between the forward and backward search.

empty intersection between these searches and a trace must be reported.

(2) The algorithm does not terminate. Since there is a finite path of length L from q_0 to q_F, the forward or backward search must find this path after at most $2L+1$ steps. This consists of two times the initial step, $L-1$ times a forward (or backward) step and L times a backward (or forward) step. Since all paths with a length of L must be included in this search, the path from the initial to the final state must have been detected. □

Theorem 2 (Soundness)**.** *Given a marking graph $\mathcal{M}$ and a uniform-cost function c, if Algorithm 1 returns a trace, it is optimal for $\mathcal{M}$ and c.*

Proof. Assume that the total cost for an optimal trace is K. Note that an optimal trace in our setting minimizes the number of 1-cost moves. By considering all possible states reachable via $\mathtt{T}_0$ transitions both before and after a single $\mathtt{T}_1$ application we consider every state reachable with a maximum cost of 1. Thus after K iterations of applying a 1-cost move (note that the directional changes do not affect the result) we must also detect the states that form a shortest path, which is encountered before any longer path. □

From the above results, we can also conclude that Algorithm 1 needs at most $2+K$ (2 initial $\mathtt{T}_0{}^*$ applications and K times $\mathtt{T}_1$-$\mathtt{T}_0{}^*$) steps to return an optimal alignment of cost K.

E. Example

Figure 4 depicts the marking graph for the Petri net of Figure 1 and the numbers in the nodes denote the exploration order by Algorithm 1. The initial and final states are respectively $\mathtt{m}_0$ and $\mathtt{m}_{37}$. Algorithm 1 starts by first computing $\mathtt{T}_0{}^*$ from the initial state. It finds $\mathtt{m}_4$ via the synchronous $\mathtt{A}$ transition. Then, no more 0-cost moves are possible. The same procedure is then started from the final state. No 0-cost transitions can be taken to $\mathtt{m}_{37}$. In this example, for simplicity, we assume that the number of markings equals the number of decision nodes.

Then, since the backward set contains fewer markings (decision nodes), a $\mathtt{T}_1$ transition is taken towards $\mathtt{m}_{37}$ to obtain states $\mathtt{m}_{35}$ and $\mathtt{m}_{36}$. Then $\mathtt{T}_0{}^*$ is computed. States $\mathtt{m}_{28}$ and $\mathtt{m}_{34}$ are found in the first iteration, then $\mathtt{m}_{23}$ is encountered, then $\mathtt{m}_{13}$, and finally $\mathtt{m}_3$. After these $\mathtt{T}_0$ iterations, no other 0-cost transition can be taken from the backward set.

Now, the forward set contains fewer decision nodes. A $\mathtt{T}_1$ transition is taken to obtain the states $\mathtt{m}_1$, $\mathtt{m}_2$, $\mathtt{m}_8$, and $\mathtt{m}_9$. After applying $\mathtt{T}_0{}^*$, we obtain $\mathtt{m}_{18}$. The forward set again contains fewer decision nodes, thus we take a $\mathtt{T}_1$ transition from the forward set. We find states $\mathtt{m}_3$, $\mathtt{m}_5$, $\mathtt{m}_{12}$, $\mathtt{m}_{13}$, $\mathtt{m}_{14}$, $\mathtt{m}_{15}$, $\mathtt{m}_{23}$, and $\mathtt{m}_{24}$.

Since we now have a non-empty intersection between the forward and backward search, we found a shortest path and can compute the corresponding alignment. By taking the path $\mathtt{m}_0 \xrightarrow{\mathtt{A}} \mathtt{m}_4 \xrightarrow{\mathtt{B}} \mathtt{m}_8 \xrightarrow{\mathtt{C}} \mathtt{m}_{18} \xrightarrow{\mathtt{D}} \mathtt{m}_{23} \xrightarrow{\tau} \mathtt{m}_{28} \xrightarrow{\mathtt{E}} \mathtt{m}_{35} \xrightarrow{\mathtt{B}} \mathtt{m}_{37}$, we find the optimal alignment γ as given in Figure 1.

IV. Implementation

A. Decision diagrams

We implemented the decision diagrams using the multi-core BDD package Sylvan [11], [15]. Instead of using BDDs for the implementation, we used a variant, called list decision diagrams (LDDs). Here instead of a boolean node, a decision

node represents a linked list of possible values for a variable. Unless the Petri net is 1-safe, meaning that each place can contain at most one token (e.g., see Figure 2), it is difficult to decide how many boolean variables are required for representing a place. An LDD is extended on-the-fly and is better suited for model checking [11].

For variable reordering, we experimented with various algorithms that have been shown to perform well in practice for symbolic reachability [16]. However, none of the variable reordering algorithms we tested improved the overall performance. We suspect this is due to the fact that we split up the transitions into two groups, which is non-standard in symbolic reachability.

B. Evaluation of bidirectional search

We experimented with various alternatives for choosing to search forward or backward. We only observed a slight performance improvement from choosing the smallest decision diagram versus alternating in each step. However, when compared to a strictly forward search, we found that the (combined) LDD size of the bidirectional search is often significantly smaller than the one for the strictly forward search. The performance of the bidirectional search is also significantly better.

C. Trace construction

When a non-empty intersection is found, a trace is returned. This is realized by storing the LDD after each operation during the algorithm, which we call *levels*. We show how a trace is reconstructed from the forward search. From the non-empty intersection, a single state s is selected and the transition relation is applied in the backward direction to obtain a set S of predecessor states. We intersect S with the previous level from the forward search and pick a state s'. The same procedure is applied until the initial state is retrieved. The trace is combined with the backward reconstruction (also starting from state s).

The trace reconstruction is efficient in practice since the transitions are only applied on single states (and thus also small LDDs). Note that it is also possible to obtain *all* shortest paths by selecting the entire intersection as a starting point and by iteratively computing all possible predecessors[2].

V. Experiments

We implemented our algorithm in the LTSmin model checking toolset [17] and compared its performance with the A* version from the process mining toolkit ProM [18]. We used RapidProM [19], an extension for the RapidMiner platform, to generate experiments and perform the A* experiments. We performed all our experiments on an Intel® Core™ i7-4710MQ processor with 2.50GHz and 7.4GiB memory, using 8 threads. The A* implementation uses multiple cores for computing the heuristic function via integer linear programming. Our implementation makes use of the multi-core BDD package Sylvan [11], which parallelizes operations on decision diagrams. However, both for A* and the symbolic algorithm we observed practically no performance difference when compared to their single-threaded results. We argue that there is plenty of room for improvement in terms of multi-core scalability for the symbolic algorithm as future work.

A. Model generation

Using the PTandLogGenerator [20] we generated Petri net models with specific characteristics, which we explain as follows.

- We specify the number of different activities in the Petri net to be on average 25, 50, or 75. This resulted in a Petri net containing on average respectively 89, 256, or 442 transitions and 84, 242, or 410 places.
- We set the process operators to what is considered a standard (STD) setting [20]. The operators are as follows:
  - the probability for sequence operators: 45%,
  - the probability for XOR operators: 20%,
  - the probability for parallel operators: 20%,
  - the probability for loop operators: 10%,
  - the probability for OR operators: 5%.

  We also consider variants, where sequence operators occur 25% instead of 45% and with a 20% increase in probability for one of the other operators (the total must be 100%):
  - the probability for XOR: 40% (+XOR),
  - the probability for parallel: 40% (+PAR),
  - the probability for loop: 30% (+LOOP),
  - the probability for OR: 25% (+OR).

  We also consider another variant (ALT) with sequence, parallel, XOR, loop, and OR respectively set to 46%, 35%, 19%, 0%, 0%, to resemble standard models without loops [21], [22].
- We also set additional features, for which the standard (STD) setting is as follows:
  - the probability for silent activities: 20%,
  - the probability for duplicate activities: 20%,
  - the probability for long-term dependencies: 20%.

  For each of these parameters, we consider a variant with one feature set to 0% (–SIL, –DUP, –LONG) and a variant with one feature set to 50% (+SIL, +DUP, +LONG). We only use these variants for standard process operators and use the standard additional features setting when we change one of the process operator settings, such that only one aspect is changed.

With the above we obtain $3 \cdot (6+6) = 36$ different settings. For each setting, we generate four models and generate a single random log trace per model. In the four log traces we add 10%, 30%, 50% and 70% noise by adding, removing, and swapping events. We now have $36 \cdot 4 = 144$ different models and log traces and we repeat this procedure 30 times to obtain $144 \cdot 30 = 4,320$ different experiments[3].

[2] In the case for returning all shortest paths, Algorithm 1 should be updated to only return a trace *after* all $\texttt{T}_0^*$ procedures have finished.

[3] We could have used the same model in multiple experiments by for instance generating multiple log traces, but by using different experiments we eliminate the randomness of model and log generation as much as possible.

TABLE I: Experimental results for the various types of experiments. From top to bottom it shows the total number of experiments per category, the number of time-outs that occurred for respectively A* and the symbolic algorithm, the number of times that A* was faster than the symbolic algorithm and vice versa, and finally the relative time improvement of the symbolic algorithm.

	Activities			**Process operators**					
	25	50	75	+LOOP	+OR	+PAR	+XOR	ALT	STD
Experiments	1,440	1,440	1,440	360	360	360	360	360	2,520
A* time-out	30	366	653	58	192	137	64	93	505
Sym time-out	2	78	263	23	29	35	12	34	210
# A* < Sym	1,086	653	529	239	95	133	208	186	1,407
# Sym < A*	354	739	733	109	242	201	146	147	981
A* / Sym	2.77	2.40	1.69	1.45	3.13	2.65	2.59	1.85	1.69

	Additional features							**Noise**			
	–DUP	–LONG	–SIL	+DUP	+LONG	+SIL	STD	10%	30%	50%	70%
Experiments	360	360	360	360	360	360	2,160	1,080	1,080	1,080	1,080
A* time-out	81	25	60	70	66	120	627	258	270	273	248
Sym time-out	32	9	27	37	17	64	157	60	78	96	109
# A* < Sym	182	211	217	228	208	171	1,051	549	562	560	597
# Sym < A*	157	147	125	109	144	142	1,002	495	469	446	416
A* / Sym	1.85	2.95	1.58	1.20	1.99	1.48	2.26	2.38	2.05	1.93	1.59

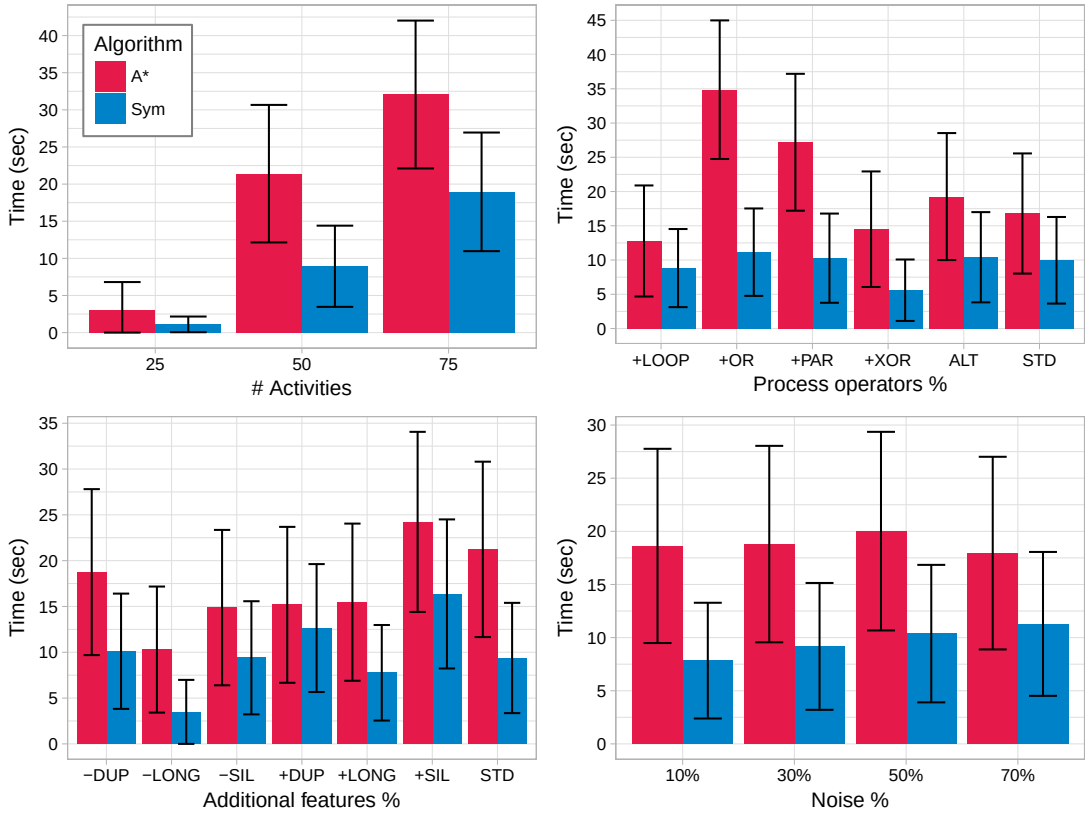

Fig. 5: Average times for the A* and symbolic algorithm on various types of experiments with 95% confidence intervals.

B. Experimental setup

For each of the 4,320 experiments, we execute the A* algorithm with 8 cores and set a time-out of 60 seconds, and do the same for our symbolic algorithm. The times include the trace generation process to form the alignment. When summarizing the results, we average the times and in case of a time-out use 60 seconds for the time.

C. Results

The results of the experiments are depicted in Figure 5 and Table I. Overall, the average time that A* requires for computing an alignment is 95% more than that of the symbolic algorithm. Interestingly, in 2,268 experiments A* outperformed the symbolic algorithm and the symbolic algorithm outperformed A* only 1,826 times (in 226 cases both algorithms did not compute an alignment within 60 seconds).

It should also be noted that A* had a time-out in 1,049 cases and the symbolic algorithm had 343 time-outs. From this, we can conclude the following.

1) The symbolic algorithm is significantly faster in computing alignments when it comes to the average time.
2) A* was able to compute more experiments faster than the symbolic algorithm than the other way around.
3) If the A* algorithm faced a time-out, in 78% of the cases the symbolic algorithm was able to compute an alignment, and in almost 34% of the cases A* was able to compute an alignment when the symbolic algorithm timed out. Thus, the techniques are complementary.

In the results for our symbolic algorithm, we observed that 63% of the time is spent on the search procedure, and the remaining 37% of the time is used for setting up the model checker (i.e., parsing the Petri net) and the trace construction. With more optimizations we think it is possible to reduce the trace construction time. All experiments and results can be found online at https://github.com/utwente-fmt/SymbolicAlign-ACSD18.

a) Results for increasing activities: The number of activities in the Petri net corresponds to the size of the model. We can see that when the number of activities increases, both algorithms take on average more time to compute alignments. Interestingly, the symbolic algorithm seems to perform relatively better for the smaller class of models than for the larger models. We argue that this is mainly due to the large influence of the 60 second time-out penalties in the 25 activity cases. When we compare the number of times that one algorithm is faster than the other, we see that for smaller models A* is better and for larger models the symbolic algorithm is preferred.

b) Results for various process operator settings: From the results, we observe large differences when changing the probabilities for various process operators. The symbolic algorithm (relatively) performs exceptionally well for the +OR, +PAR and +XOR cases. One similarity between these three instances is that they all significantly increase the size of the marking graph. Just as with the increase in activities, we consider the symbolic algorithm better suited for larger state-spaces that have a regular structure. This is supported by the observed performance increase. Another interesting result is that when we increase the probability of loops, the performance of the symbolic algorithm drops compared to A*. Loops may cause the symbolic algorithm to compute many already visited states in next-state calls, which might explain the result.

c) Results for various additional feature settings: For the additional features the most interesting result is when we increase the number of duplicate activities. In this case, the symbolic algorithm performs worst compared to A* (though it's average compute time is still lower). We argue that due to this effect, there are more synchronous moves in the model, which is bad for the symbolic algorithm as this could lead to many $\mathtt{T}_0$ operations. In the other instances, we do not observe too many differences with the standard.

d) Results for increasing noise: For the increase in noise we can see that the performance of A* stays relatively the same while the performance of the symbolic algorithm drops. We suppose this is a consequence of the fact that for a higher amount of noise, the total cost for an optimal alignment increases. When the alignment cost is high, the symbolic algorithm has to perform more $\mathtt{T}_1$ operations and thus more symbolic operations.

VI. Related Work

We refer to [1] for an overview of different process mining techniques. One of the earliest works in conformance checking was from Cook and Wolf [23]. They compared log traces with paths generated from the model.

Token-based replay [4] is one technique to check for conformance. 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 does not provide good insights any more.

Alignments were introduced [5], [24], [8] to overcome the limitations from 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. The A^* shortest path algorithm [7] in combination with Integer Linear Programming (ILP) to prune the search space (i.e., the synchronous product of the model and the log trace) is considered the state-of-the-art in computing alignments [8].

For larger models, merely constructing alignments on the synchronous product may quickly become too computationally intensive to handle. Several decomposition techniques have been developed [25], [26], [27] that partition the Petri net into smaller subprocesses. For instance, fragments that have a single-entry and single-exit node (SESE) represent an isolated part of the model [27]. This way, localizing conformance problems becomes easier in large models. Such decompositions may be combined with any conformance checking technique, including our approach.

A different solution to deal with large instances is to no longer guarantee optimality. Several techniques exists [28], [29] that can very efficiently compute 'good', but not necessarily optimal, alignments for large instances.

There are a number of existing algorithms for computing the shortest path symbolically. Symbolic versions of Dijkstra's algorithm, Bellman-Ford, and A* exist [12], [13]. However one of the main problems with the related approaches is the overhead for bookkeeping or updating the shortest path info. For instance, the symbolic A* implementation [13] tracks a $g \times h$ matrix, where each cell represents the set of states that have current cost g from the initial state and heuristic cost h to the final state. Our approach is designed in such a way that no additional cost information needs to be stored directly for the

decision diagrams, with the downside that it is only suitable for uniform-cost functions.

VII. Conclusion

We have designed and implemented a symbolic algorithm for computing shortest paths on uniform-cost functions. We have applied this algorithm in the context of computing alignments and compared its performance with the state-of-the-art approach for computing alignments. For the empirical study we generated a set of over 4,000 experiments to test the effect of various characteristics on performance.

From the experimental results we conclude that our symbolic algorithm is well-suited for computing alignments. We found that the symbolic technique is more robust in computing alignments when compared to A*. We further observe that the symbolic approach is especially favourable in computing alignments for models with large state-spaces. However, A* seems to perform better for smaller instances. The techniques are complementary; due to the different underlying techniques, instances for which A* takes a (too) long time to compute may be solved quickly with the symbolic approach and vice versa.

One direction for future work is to compare performance differences on industrial case studies. Another direction is to design an algorithm that analyses the Petri net model and selects the most appropriate algorithm for computing alignments. We also think that the performance of our algorithm may be further improved by e.g., improving the multi-core scalability, experimenting with different variable orderings and improving the trace construction performance.

References

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

[2] C. Liu, B. F. van Dongen, N. Assy, and W. M. P. van der Aalst, "Component behavior discovery from software execution data," in *2016 IEEE Symposium Series on Computational Intelligence, SSCI 2016, Athens, Greece, December 6-9, 2016*, pp. 1–8, 2016.

[3] M. Leemans and W. M. P. van der Aalst, "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, Ottawa, ON, Canada, September 30 - October 2, 2015*, pp. 44–53, 2015.

[4] A. Rozinat and W. M. P. van der Aalst, "Conformance checking of processes based on monitoring real behavior," *Information Systems*, vol. 33, no. 1, pp. 64 – 95, 2008.

[5] W. M. P. van der Aalst, A. Adriansyah, and B. F. van Dongen, "Replaying history on process models for conformance checking and performance analysis," *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 2, no. 2, pp. 182–192, 2012.

[6] A. Adriansyah, N. Sidorova, and B. F. van Dongen, "Cost-Based Fitness in Conformance Checking," in *11th International Conference on Application of Concurrency to System Design, ACSD 2011, Newcastle Upon Tyne, UK, 20-24 June, 2011*, pp. 57–66, 2011.

[7] P. E. Hart, N. J. Nilsson, and B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

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

[9] A. Valmari, "The state explosion problem," in *Lectures on Petri Nets I: Basic Models: Advances in Petri Nets* (W. Reisig and G. Rozenberg, eds.), (Berlin, Heidelberg), pp. 429–528, Springer Berlin Heidelberg, 1998.

[10] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang, "Symbolic model checking: 10^{20} States and beyond," *Information and Computation*, vol. 98, no. 2, pp. 142 – 170, 1992.

[11] T. van Dijk and J. van de Pol, *Sylvan: Multi-Core Decision Diagrams*, pp. 677–691. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015.

[12] D. Sawitzki, "Experimental Studies of Symbolic Shortest-Path Algorithms," in *Experimental and Efficient Algorithms* (C. C. Ribeiro and S. L. Martins, eds.), (Berlin, Heidelberg), pp. 482–497, Springer Berlin Heidelberg, 2004.

[13] S. Edelkamp and P. Kissmann, "Optimal Symbolic Planning with Action Costs and Preferences," in *Proceedings of the 21st International Joint Conference on Artifical Intelligence*, IJCAI'09, (San Francisco, CA, USA), pp. 1690–1695, Morgan Kaufmann Publishers Inc., 2009.

[14] R. E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Trans. Comput.*, vol. 35, no. 8, pp. 677–691, 1986.

[15] T. van Dijk, *Sylvan: multi-core decision diagrams*. PhD thesis, Enschede, 2016. IPA dissertation series no. 2016-09.

[16] J. Meijer and J. van de Pol, *Bandwidth and Wavefront Reduction for Static Variable Ordering in Symbolic Reachability Analysis*, pp. 255–271. Lecture Notes in Computer Science, Springer International Publishing, 2016. eemcs-eprint-27067.

[17] G. Kant, A. Laarman, J. Meijer, J. van de Pol, S. Blom, and T. van Dijk, "LTSmin: High-Performance Language-Independent Model Checking," in *Tools and Algorithms for the Construction and Analysis of Systems* (C. Baier and C. Tinelli, eds.), vol. 9035 of *Lecture Notes in Computer Science*, pp. 692–707, Springer Berlin Heidelberg, 2015.

[18] H. M. W. Verbeek, J. C. A. M. Buijs, B. F. van Dongen, and W. M. P. van der Aalst, *XES, XESame, and ProM 6*, pp. 60–75. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011.

[19] W. M. P. van der Aalst, A. Bolt, and S. J. van Zelst, "RapidProM: Mine Your Processes and Not Just Your Data," *CoRR*, vol. abs/1703.03740, 2017.

[20] T. Jouck and B. Depaire, "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), Rio de Janeiro, Brazil, September 21, 2016.*, pp. 23–27, 2016.

[21] M. Kunze, A. Luebbe, M. Weidlich, and M. Weske, *Towards Understanding Process Modeling – The Case of the BPM Academic Initiative*, pp. 44–58. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011.

[22] S. J. van Zelst, A. Bolt, and B. 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, Zaragoza, Spain, June 25-30, 2017.*, pp. 1–15, 2017.

[23] J. E. Cook and A. L. Wolf, "Software Process Validation: Quantitatively Measuring the Correspondence of a Process to a Model," *ACM Trans. Softw. Eng. Methodol.*, vol. 8, no. 2, pp. 147–176, 1999.

[24] A. Adriansyah, B. F. van Dongen, and W. M. P. van der Aalst, "Conformance Checking Using Cost-Based Fitness Analysis," in *2011 IEEE 15th International Enterprise Distributed Object Computing Conference*, pp. 55–64, 2011.

[25] W. M. P. van der Aalst, "Decomposing Petri nets for process mining: A generic approach," *Distributed and Parallel Databases*, vol. 31, no. 4, pp. 471–507, 2013.

[26] W. M. P. van der Aalst, A. Kalenkova, V. Rubin, and H. M. W. Verbeek, *Process Discovery Using Localized Events*, pp. 287–308. Cham: Springer International Publishing, 2015.

[27] J. Munoz-Gama, J. Carmona, and W. M. P. van der Aalst, "Single-Entry Single-Exit decomposed conformance checking," *Information Systems*, vol. 46, no. Supplement C, pp. 102 – 122, 2014.

[28] B. F. van Dongen, J. Carmona, T. Chatain, and F. Taymouri, "Aligning Modeled and Observed Behavior: A Compromise Between Computation Complexity and Quality," in *Advanced Information Systems Engineering - 29th International Conference, CAiSE 2017, Essen, Germany, June 12-16, 2017, Proceedings* (E. Dubois and K. Pohl, eds.), vol. 10253 of *Lecture Notes in Computer Science*, pp. 94–109, Springer, 2017.

[29] F. Taymouri and J. Carmona, "A Recursive Paradigm for Aligning Observed Behavior of Large Structured Process Models," in *Business Process Management - 14th International Conference, BPM, 2016, Rio de Janeiro, Brazil, September 18-22, 2016. Proceedings* (M. L. Rosa, P. Loos, and O. Pastor, eds.), vol. 9850 of *Lecture Notes in Computer Science*, pp. 197–214, Springer, 2016.