Computing Alignments for Partially-ordered Traces Through Petri Net Unfoldings

Ariba Siddiqui[1], Wil M. P. van der Aalst[1][0000−0002−0955−6940], and Daniel Schuster[1][0000−0002−6512−9580]

RWTH Aachen University, Aachen, Germany
ariba.siddiqui@rwth-aachen.de, wvdaalst@pads.rwth-aachen.de, schuster@pads.rwth-aachen.de

Abstract. Conformance checking techniques aim to provide diagnostics on the conformity between process models and event data. Conventional methods, such as trace alignments, assume strict total ordering of events, leading to inaccuracies when timestamps are overlapping, coarse, or missing. In contrast, existing methods that support partially ordered events rely upon the interleaving semantics of Petri nets, the reachability graphs, which suffer from the state space explosion problem. This paper proposes an improved approach to conformance checking based upon partially ordered event data by utilizing Petri net unfolding, which leverages partial-order semantics of Petri nets to represent concurrency and uncertainty in event logs more effectively. Unlike existing methods, our approach offers a streamlined one-step solution, improving efficiency in the computation of alignments. Additionally, we introduce a novel visualization technique for partially ordered unfolding-based alignments. We implement unfolding-based alignments with its user-friendly insights in a conformance analysis tool. Our experimental evaluation, conducted on synthetic and real-world event logs, demonstrates that the unfolding-based approach is particularly robust in handling high degrees of parallelism and complexity in process models.

Keywords: Conformance checking · Petri nets unfolding · Partial order semantics · Alignments · Process mining.

1 Introduction

Conformance checking, a major task within process mining, deals with comparing event data with process models, typically represented using notations like BPMN [10] or Petri Nets [17]. *Trace alignments* [2] are considered to be state-of-the-art in conformance checking. These techniques aim to search for alignments with minimum deviations, with Adriansyah et al. [2] introducing a method using a shortest-path search using A^* [16] in a Synchronous Product Net (SPN) of the trace net and the model net. Another method [15] relies on a unified representation of process models and event logs based on *event structures*, a well-known model of concurrency. The method returns a set of statements in natural language describing behavior allowed by the model but not observed in the log and vice versa.

One problem with trace alignment algorithms is that they assume totally-ordered event data and produce totally-ordered alignments. In a more recent work by Lu et al. [21], this limitation is overcome. A partially-ordered trace is converted to an occurrence net (i.e., a simple Petri net without choices), and an SPN is computed between the occurrence net and the normative model. From hereon, the shortest path computation is done on the state space of the product net (just like [2]) using the A^* algorithm and a predefined cost function. As a firing sequence with minimum cost is found, it is replayed on the net while unfolding it into a new net. That is, while replaying, whenever a place is seen for the second time (owing to a loop), it is cloned into a new place. The result is an occurrence net, which can be converted into a partially-ordered alignment (*p-alignment*). This results in a two-step computation solely because of the reliance on interleaving semantics of Petri nets, the *reachability graphs*, which additionally suffer from the well-known state space explosion [29], limiting its scalability to larger instances of problems with high concurrency.

To address the issue of state space explosion in the interleaving semantics of Petri nets, researchers in model checking [12], diagnosis [5], and planning [6,18] have extensively explored the Petri net unfolding process. This partial-order semantics, introduced in [23] and further detailed as *branching processes* in [11], offers an alternative to interleaving semantics. A branching process unfolds all partially-ordered runs of a Petri net into a tree-like structure, representing concurrency, causality, and choice points. Although comprehensive, these processes are often infinite. McMillan's seminal work [22] introduced a method to construct a finite prefix of the branching process that retains state reachability information. This prefix, which models concurrency more compactly than the reachability graphs [24], is typically much smaller. However, McMillan's algorithm can produce unnecessarily large prefixes. To address this, Esparza et al. proposed the *ERV unfolding algorithm* [13], which generalizes McMillan's algorithm using *adequate orders*. Since these prefixes encode all information on reachable states of a Petri net, they can effectively be used for shortest-path computation problems. Esparza et al. showed that the reachability problem is NP-complete relative to prefix size [14], with their *OnTheFly ERV* variant proving most efficient amongst all unfolding-based reachability techniques.

Despite advances in alignment computation, partial order-based conformance checking, and Petri net unfolding, these domains have yet to converge. This paper addresses this gap by applying Petri net unfolding to optimal alignment computation based on partially ordered event data. We apply Petri net unfolding to replace reachability graphs to compute *unfolding-based alignments*, using an algorithm we term $ERV[\lhd_c]$. Our method offers a straightforward, one-step solution, avoiding the need to create a partially ordered alignment from execution sequences artificially. A key advantage is faster computation times, particularly when there is high concurrency between events. As a second contribution, we propose a method for visualizing the results of unfolding-based alignments using a technique from [26]. This method visualizes trace variants from partially ordered event data in chevron-based views and is well-suited for graph-based align-

ments. Unlike existing methods, our approach makes the relationship between log and model more explicit. We evaluate our algorithm using synthetic event data with varying levels of parallelism, showing that unfolding-based alignments perform better under increasing parallelism. Regarding runtime, the (directed) unfolding-based approach outperforms the classic method by Lu et al. [21]. However, in lower parallelism settings, our method faces higher computational costs. Further experiments with real-world data confirm these findings, demonstrating that unfolding-based algorithms are more robust as model complexity grows.

2 Preliminaries

In process mining, an *event log* captures the occurrence of activities within a business process execution. Logs are analyzed to discover process models, check conformance, or optimize performance. Each event includes *start* and *end* timestamps, a *case identifier*, and a label for the event. Case identifiers group events into a single *case* representing a process execution. Conformance checking compares each case with the normative process model, and deviations are analyzed across the log. A key challenge is representing temporal relationships, especially for concurrently occurring events. To address this, we define *partially-ordered traces (p-traces)* to represent concurrent activities the need for total ordering.

Definition 1 (Partially-ordered Trace (P-Trace)). *For an event log E, in which all events share a common case identifier c, a partially-ordered trace is a labeled directed acyclic graph $\rho_c = ((E, \prec_{\rho_c}), l_{\rho_c})$ where E is a finite set of graph nodes and $\prec_{\rho_c}$ is a partial order over the nodes of the graph such that for any two events a and b, $a \prec_{\rho_c} b$ iff the end timestamp of a is strictly less than the start timestamp of b. $l_{\rho_c} : E \to \mathcal{L}$ is a labeling function for the nodes, where $\mathcal{L}$ is the universe of events labels of E. Each edge in a p-trace is referred to as a dependency between two events e_i and e_j, indicating that e_i has led to the execution of e_j.*

In this paper, we focus on Petri nets as the process modelling formalism. The notations of Petri nets used are largely based upon [7]. In its general, simplest form, Petri nets can be seen as graphs that consist of *places*, *transitions*, and directed edges between them. Places (depicted as circles) and transitions (depicted as squares) represent the two types of nodes of the graph. An example Petri net $\mathcal{N}_{mgmt}$ is shown in Fig. 1, modeling a software project management process with 11 places and 12 transitions.

Definition 2 (Labeled Petri Net, System Net). *A labeled Petri net (in short, net) is a bipartite graph $\mathcal{N} = (P, T, F, \lambda)$ with a disjoint finite set of nodes consisting of places P and transitions T, a set of (directed) arcs or flow relations $F \subseteq ((P \times T) \cup (T \times P))$, and a labeling function $\lambda : T \to L^{\tau}$ assigning an event label or τ (label of a silent transition) to each transition. A system net $SN = (\mathcal{N}, M_{init}, M_{final})$ is a labeled Petri net with a well-defined initial marking M_{init} and final marking M_{final}.*

Definition 3 (Pre- and Postset). *Let $\mathcal{N} = (P, T, F, \lambda)$ be a labeled Petri net and $x \in P \cup T$ be a node of $\mathcal{N}$. The set ${}^\bullet x = \{y \in P \cup T \mid (y, x) \in F\}$ is called the preset, and $x^\bullet = \{y \in P \cup T \mid (x, y) \in F\}$ is called the postset of x.*

We make the following assumptions: the sets of places are always non-empty (i.e., $P \neq \emptyset$), transitions always have incoming and outgoing edges, and the pre- and postsets of transitions are always finite.

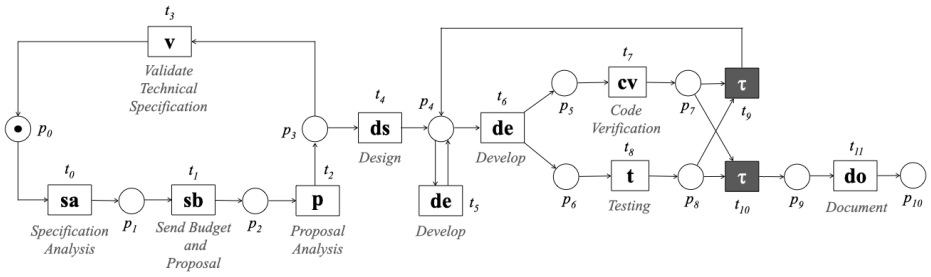


Fig. 1: An example Petri net $\mathcal{N}_{mgmt}$ of a software development management process with the initial marking $M_{init} = [p_0]$ and final marking $M_{final} = [p_{10}]$

To model the dynamic properties of a net, the notion of *marking* exists. It is a multiset over places P of a net, in which the multiplicity of a place denotes the number of *tokens* assigned to it. The net in Fig. 1 has an initial marking $m_0 = [p_0]$. A marking is said to be *reachable* if there exists a sequence of transition *firings* from the initial state of a net, where a transition firing follows the well-known firing semantics of Petri nets [24]. In this paper, we assume all nets to be 1-safe, meaning no place contains more than one token at a time.

The dynamic behavior of Petri nets can be viewed in two ways: the first, called *interleaving semantics*, considers firing sequences as possible execution paths. The second, *true concurrency semantics*, views execution as partially-ordered sequences, representing runs (hereby referred to as *distributed runs*) as partial orders. These partial orders can be modeled with simpler structure nets called *occurrence nets*, obtained by 'unfolding' system nets into a tree-like structure. Occurrence nets represent partially-ordered event sequences from the initial marking, together forming *branching processes*. Stopping the unfolding at different instances yields different processes, but *the* unique *unfolding* is the complete branching process obtained by unfolding as much as possible. The subsequent definitions in this section are largely based upon [13]. Prior to introducing occurrence nets and causal nets, we define the notions of *causality* and *conflict* as follows—1. two nodes x and y are in a *causal relation* with each other if there exists a path between x to y, 2. two nodes are in *conflict* with each other if there exist two different paths to these nodes which start at the same point and diverge immediately afterwards (although later on they can converge again).

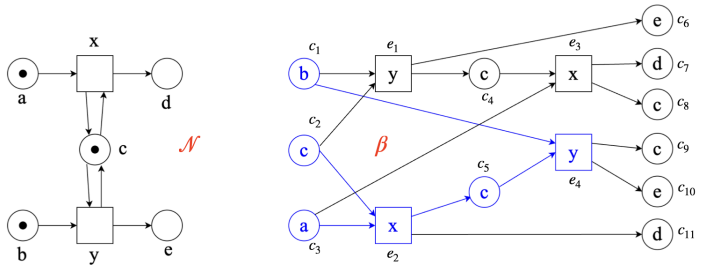


Fig. 2: A system net $\mathcal{N}$ (left) with $M_{init} = [a, b, c]$, $M_{final} = [d, c, e]$, and its branching processes β (right), which is also a complete finite prefix of the unfolding of $\mathcal{N}$.

Definition 4 (Occurrence Net, Causal Net). *Let $\mathcal{N} = (B, E, F, \lambda)$ be a labeled Petri net. We have an unlabeled net $\mathcal{O} = (B, E, F)$ consisting of the transitions, places and flow relation of $\mathcal{N}$:*

1. *$\mathcal{O}$ is an occurrence net, if the following conditions hold:*
   (a) $\forall b \in B(|{}^\bullet b| \leq 1)$, no place in $\mathcal{O}$ contains two or more incoming edges;
   (b) $\nexists e \in E$ st. it is in conflict with itself;
   (c) $F^+ \cap (F^{-1})^+ = \emptyset$, $\mathcal{O}$ is acyclic;
   (d) $\mathcal{O}$ is finitely preceded, i.e., for every $x \in B \cup E$, the set of elements $y \in B \cup E$ such that there exists a path from y to x is finite.
   Places and transitions in an occurrence net are called as conditions B and events E.
2. *$\mathcal{O}$ is a causal net, if (a)-(d) hold, in addition to:*
   (e) $\forall b \in B(|b^\bullet| \leq 1)$, no place in $\mathcal{O}$ contains two or more outgoing edges

Those labeled occurrence nets obtained by unfolding system nets are referred to as *branching processes*, formalized as below:

Definition 5 (Branching Process). *A branching process of a system net $SN = ((P, T, F, \lambda), M_{init}, M_{final})$ is a labeled occurrence net $\beta = (\mathcal{O}, h) = (B, E, F, h)$ where the labeling function $h : B \cup E \to P \cup T$ is a net homomorphism from β to SN such that: $\forall v, v' \in B \cup E.({}^\bullet v = {}^\bullet v' \wedge h(v) = h(v') \implies v = v')$. It is simply a mapping between nets that preserves the nature of nodes and the environment of nodes. It associates the conditions/events of a branching process to the places/transitions of its system net. For a formal definition of a net homomorphism, we refer to [25].*

Fig. 2 shows the branching process β of a Petri net $\mathcal{N}$. It consists of two different distributed runs corresponding to firing sequences of x followed by y (highlighted in blue) and y followed by x, respectively. Labels inside each node of β represent the places/transitions it maps to. Notice how no condition (place) in the highlighted run has more than one incoming/outgoing arc, and all events

are related in a causal or concurrency relationship. Therefore, it is a causal net. Another thing to notice in the highlighted distributed run of β is that when the run includes conditions c_9 and c_{10}, it has no outgoing arcs that enable any more transitions in the Petri net. Such a distributed run is called *complete*.

The unfolding process of a system net starts with the initial conditions and extends it iteratively with the new possible events and a condition for every output place of the newly added event. This process can typically be infinite. McMillan's [22] (and ERV [13]) algorithm builds a *complete finite prefix* of the unfolding, which terminates when the unfolding represents all of the reachable markings of the Petri net. The foundations of the algorithm are laid by the key concepts of *configurations* and *cuts*. A finite configuration C represents the set of events that have occurred so far in a distributed run. For the highlighted run in β (cf. Fig. 2), $\{e_2, e_4\}$ form a finite configuration while $\{e_1, e_2\}$ does not, since e_1 and e_2 are in conflict with each other. A configuration is related to a marking $Mark(C)$ by identifying which conditions will contain a token after firing all events in configuration C. Thus, $Mark(C)$ corresponds to a final marking in the original system net, reached by firing all transitions labeled by the events in a given finite configuration. When a finite configuration is associated with an event, it is termed as a *local configuration*.

Definition 6 (Local Configuration). *For an occurrence net $\mathcal{O} = (B, E, F)$, a local configuration $[e]$ of an event $e \in E$ is defined by $\{e' \in E \mid (e', e) \in F^*\}$*

It represents a minimal set of all the events denoted by $[e]$ that must have been fired for the event e to be enabled. For example, in β, $[e_4] = \{e_2, e_4\}$. Each local configuration is also associated with the distributed run it represents. It consists of precisely the events in $[e]$, in addition to the conditions and arcs connecting them. We term it as a *local distributed run*.

Definition 7 (Local Distributed Run). *Let $\mathcal{O} = (B, E, F)$ be an occurrence net. A local distributed run of an event $e \in E$ is a causal net $\mathcal{O}_e = (B_e, E_e, F_e)$ where:*

- *$B_e = \{c \in B \mid (c, e) \in F^*\}$ is the set of conditions,*
- *$E_e = [e]$ is the set of events, and*
- *F_e is the flow relation such that $F_e(x, y) = F(x, y), \forall x, y \in B_e \cup C_e$.*

The key to building a complete finite prefix (due to McMillan) is to identify those events where the unfolding process can be stopped without any loss of information. Those events are called *cut-off events* and can be defined in terms of *adequate order* on configurations [13,22,8]. Adequate orders are simply strict partial orders to choose between the extensions of two events when the $Mark$ of their respective local configurations is the same, and one of them precedes the other one with respect to the partial order. The one not chosen to be extended will thus be the *cut-off event*. For a strict partial order to qualify as an adequate order, it must satisfy certain properties (cf. [13] for details). Thus, the unfolding can be ceased as soon as an event e takes a prefix to a marking which has

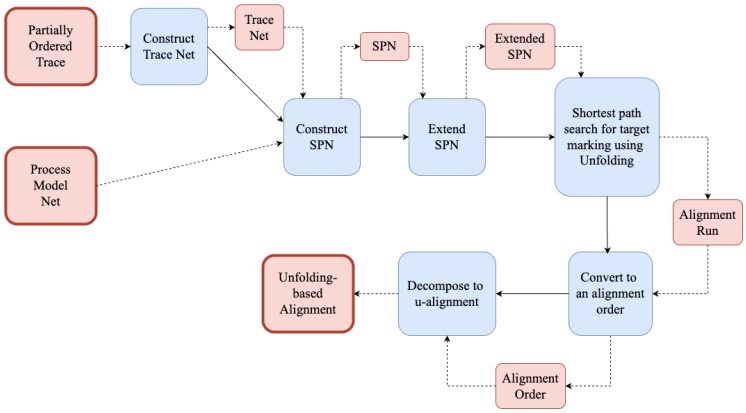


Fig. 3: Overview of the proposed method to compute unfolding-based alignments given a partially-ordered trace and a process model. Input and output of each step are shown through dashed arrows. Various steps are highlighted in blue

already been induced by a previously added event e' such that $[e'] \lhd [e]$. The event e is marked as a cutoff event. Assume an adequate order $\lhd$ where $e_1 \lhd e_2$ iff $|[e_1]| < |[e_2]|$ and a current state of marking as $\{d, c, e\}$ in $\mathcal{N}$ of Fig. 2. If there were a transition z from place d to a in $\mathcal{N}$, the event mapped to z in β would have been marked as a cut-off event. This is because it would bring the marking of $\mathcal{N}$ to $\{a, c, e\}$, already represented in β by the local distributed run of e_1, for which $|[e_1]|$ is lesser already.

3 Computation of Unfolding-Based Alignments

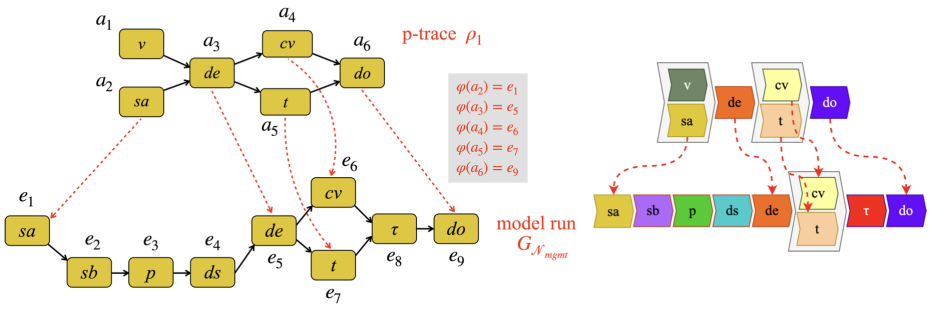
We describe our proposed approach as illustrated in Fig. 3. Given a partially ordered trace (p-trace) and a model net, we first model the p-trace as a *trace net* [21], a Petri net capturing the p-trace's behavior. Each event becomes a transition, and places represent dependencies between events. Next, we construct the SPN with the model net. The product is constructed by pairing transitions in one net with transitions in the other net that have the same label [4]. This helps to model all possible alignment moves explicitly. A formal definition of SPN is provided online (Def. 1 in Appendix, [28]). Then we extend it by adding a target transition and place, and revise the final marking to the target place. Using unfolding algorithms $ERV[\lhd_c]$ or $ERV[\lhd_h]$ (cf. Sec. 3.2 and 3.3), we unfold the extended SPN to find a minimum-cost distributed run leading to the target transition, yielding an optimal alignment run. Finally, we use this run to build an alignment order and decompose it into *unfolding-based alignments (u-alignments)*. The goal is to relate p-trace with the closest run through the model net (one with minimum deviations) in a meaningful and interpretable way.

We begin by defining u-alignments, in the context of an alignment function that relates synchronous moves when the alignment is decomposed into its log

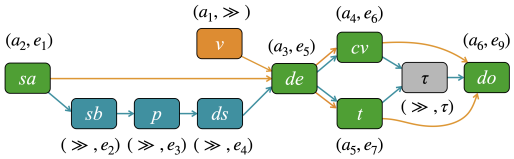
and model parts. This refinement allows for a clearer relationship between log and model visualizations.

Definition 8 (Unfolding-based Alignment (u-alignment)). *Let* $\rho_c = ((E, \prec_{\rho_c}), l_{\rho_c})$ *denote a p-trace and* $\mathcal{N} = ((P, T, F, \lambda), M_{init}, M_{final})$ *be a system net with induced partial order* $G_{\mathcal{N}} = ((T', \prec_{\mathcal{N}}), l_{\mathcal{N}})$ *over a run of* $\mathcal{N}$ *(we refer to it as a model run), where* T' *is an arbitrary set such that* $T' \cap E = \emptyset$ *and* $l_{\mathcal{N}} : T' \to rng(\lambda)$ *is a labeling function. An unfolding-based alignment (u-alignment) is a tuple* $(\rho_c, G_{\mathcal{N}}, \varphi)$ *where* $\varphi : E' \to T'$ *is an injective function, we refer to as the alignment function defined over a set* $E' \subseteq E$ *and* $T' \subseteq T$ *such that:*

1. $\forall e \in E', t \in T'$ *st.* $(l_{\rho_c}(e) = l_{\mathcal{N}}(t)) \Leftrightarrow \varphi(e) = t$ *(nodes with same activity labels are mapped together; they are synchronous), and*
2. *A directed graph consisting of nodes and edges from the p-trace* ρ_c*, model run* $G_{\mathcal{N}}$ *such that there is a single node for each pair of synchronous nodes is a strict partial order.* $\rho_c \uplus G_{\mathcal{N}}$ *denotes a partial order that combines* ρ_c *and* $G_{\mathcal{N}}$ *in such a way. We refer to it as an alignment order.*



(a) A u-alignment comprising of an arbitrary p-trace ρ_1 and a partial order $G_{\mathcal{N}_{mgmt}}$ induced by an example model run through $\mathcal{N}_{mgmt}$ Fig. 1. Mapping by the alignment function φ is shown in dashed red arrows. A chevron-based view of the alignment is visualized on the right



(b) An alignment order $\rho_1 \uplus G_{\mathcal{N}_{mgmt}}$. The partial order contains a single node (a_i, e_j) for each pair of synchronous nodes, colored in green. All other nodes and dependencies are colored according to their origin — orange if it originates from ρ_1 (a node $(a_i, \gg)$), blue if from $G_{\mathcal{N}_{mgmt}}$ (a node $(\gg, e_j)$). A grey node $(\gg, \tau)$ represents an invisible move originating from model

Fig. 4: An example of a u-alignment along with its corresponding alignment order

Fig. 4a shows an illustration of how in a u-alignment, an alignment function maps a p-trace ρ_1 to a partial order induced by an example model run $G_{\mathcal{N}_{mgmt}}$ through $\mathcal{N}_{mgmt}$. The alignment function φ is defined over nodes of p-trace $\{a_2, a_3, a_4, a_5, a_6\}$ such that $\varphi(a_2) = e_1$, $\varphi(a_3) = e_5$, $\varphi(a_4) = e_6$, $\varphi(a_5) = e_7$ and $\varphi(a_6) = e_9$. Note that φ maps only nodes with the same labels; the label for a_2 and e_1 is 'sa', and so on. Fig. 4b shows an alignment order for ρ_1 and $G_{\mathcal{N}_{mgmt}}$, a partially ordered structure of *alignment moves*. Nodes in this graph are colored according to where they originate from. An orange node represents a *log move* from the p-trace, while a blue node represents a *model move* from the model run. A green node indicates a *synchronous move*, originating from both the p-trace and the model run. A grey node denotes an invisible move from the model run. Edges represent dependencies, categorized as *model dependencies* (blue) or *log dependencies* (orange), based on their origin. Given such an alignment, the partial orders being directed acyclic graphs, can be visualized in a chevron-based view based upon recursive sequential and parallel partitioning of the graphs, a visualization technique from [26]. The result of the same is illustrated on the right of Fig. 4a. Notice how this view retains the relationship between the log and model parts through the alignment function.

By understanding how to combine a trace with a model run to create an alignment order, we can similarly decompose an alignment order into two partial orders while defining the alignment function simultaneously. This yields a u-alignment. Given an alignment order $G_\varphi = ((M, \prec_\varphi), l_\varphi)$, we denote the two decomposed partial orders as $G_{\varphi\downarrow 1}$ and $G_{\varphi\downarrow 2}$, respectively, i.e., $G_\varphi \rightarrow G_{\varphi\downarrow 1} \uplus G_{\varphi\downarrow 2}$. An alignment function φ is defined for every synchronous move. For our example in Fig. 4, decomposing the alignment order from Fig. 4b results in two partial orders isomorphic to ρ_1 and $G_{\mathcal{N}_{mgmt}}$ in Fig. 4a.

In order to compute u-alignments from an SPN, it is important to realize that each transition of an SPN of a trace net and a model net represents an alignment move [4]. Thus, distributed runs of the product net define a partially-ordered structure of alignment moves, an alignment order. Consequently, the problem of computing alignment(s) for a given p-trace and a model net can be reformulated as the problem of finding complete distributed run(s) in its SPN. Since we seek an *optimal* run corresponding to the fewest log and model moves, we use a *default cost function* [3], which assigns higher costs to log and model moves than to synchronous and invisible moves, reducing the problem to finding minimum-cost complete distributed runs in the product net.

3.1 Finding Optimal Distributed Runs

We already know that a complete finite prefix of a system net encodes all its reachable markings. We exploit this to search for a target marking corresponding to the final marking of an SPN in order to find complete distributed run(s). To enable search for a target marking in an SPN, we apply the standard encoding trick of adding a *target transition* t^* to the product net, which consumes the final marking. To preserve 1-safeness, we also add a target place p^* as a successor to t^*.

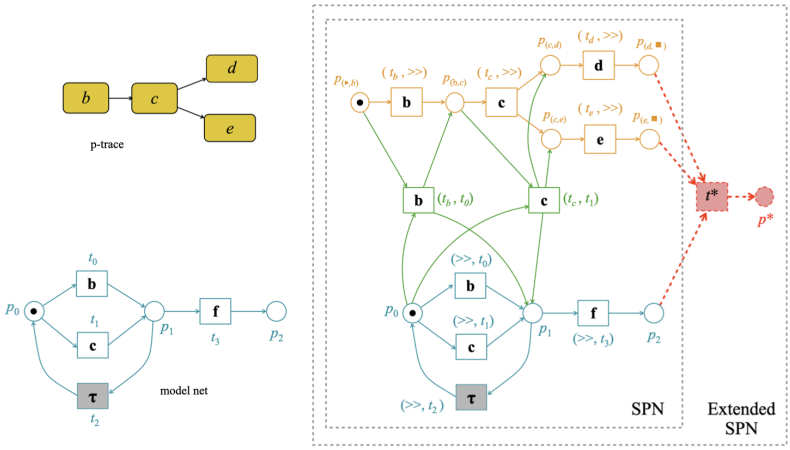


Fig. 5: An extended synchronous product net $SN^{ext}_{\otimes}$ of an SPN of a trace net and a model net. The target transition is t^*, and a target place p^* is a successor to t^*. For this system net, $M_{init,ext} = [p_{(\blacktriangleright,b)}, p_0]$ and $M_{final,ext} = [p^*]$

Definition 9 (Extended Synchronous Product Net). *Let* $SN_{\otimes} = ((P_{\otimes}, T_{\otimes}, F_{\otimes}, \lambda_{\otimes}), M_{init,\otimes}, M_{final,\otimes})$ *denote a synchronous product net of a trace net and an easy sound* [1] *model net. An extended sychronous product net of* $SN_{\otimes}$ *is another system net* $SN^{ext}_{\otimes} = ((P^{ext}_{\otimes}, T^{ext}_{\otimes}, F^{ext}_{\otimes}, \lambda^{ext}_{\otimes}), M^{ext}_{init,\otimes}, M^{ext}_{final,\otimes})$ *obtained by extending* $SN_{\otimes}$ *with a transition* t^* *and a place* p^* *such that* ${}^\bullet(t^*) = M_{final,\otimes}$, ${}^\bullet(p^*) = t^*$ *and* $(p^*)^\bullet = \emptyset$. *For such a system net,* $M^{ext}_{init,\otimes} = M_{init,\otimes}$ *and* $M^{ext}_{final,\otimes} = [p^*]$.

Fig. 5 shows an extended SPN $SN^{ext}_{\otimes}$ of an SPN $SN_{\otimes}$ after adding the target transition and place depicted in dotted red. We shall be using this product net of an exemplary trace net (induced by a p-trace) and an easy sound model net shown in Fig. 5) in the remainder of this paper.

Assume a branching process of this extended product net. We define a *target event* e^* in its branching process that maps to the target transition, i.e., for which $h(e^*) = t^*$. The encoding trick ensures that e^* is enabled only when the target marking corresponding to the final marking is reached. Therefore, its local distributed run represents a *complete* distributed run leading to e^*. We refer to such a complete distributed run as an *alignment run.*

Definition 10 (Alignment Run). *Let* $SN^{ext}_{\otimes}$ *denote an extended synchronous product net obtained by extending* $SN_{\otimes}$ *with a transition* t^* *and a place* p^*. *An alignment run* $\gamma = (B, E, F, h)$ *is a complete distributed run of* $SN^{ext}_{\otimes}$ *leading to a target event* e^* *such that the set* $\{b \in B \mid b^\bullet = \emptyset\}$ *contains a single condition* b^* *where* ${}^\bullet b^* = e^*$ *and* $h(e^*) = t^*$. *We denote the universe of all such alignment runs by* $\mathcal{U}_{SN^{ext}_{\otimes}}$.

[1] A net is *easy sound* if its complete finite prefix $\mathcal{P}$ contains at least one complete distributed run.

To ensure optimality, we use a *scoring function over finite configurations*, a monotonic function assigning additive costs. This function allows comparing local configurations of target events, where a configuration with lower cost is preferred, effectively comparing distributed runs based upon costs of their member events. The function is used to define an *optimal alignment run* in the following definition.

Definition 11 (Scoring Function Over Finite Configurations). *Let $\mathcal{C}_{all}$ denote the set of all finite configurations of a branching process $\beta = (B, E, F, h)$ of an extended SPN and cost be a cost function on transitions of the SPN such that it assigns a constant positive cost to log and model transitions, whereas synchronous transitions have a cost of 0, and invisible transitions have a negligible positive cost. The scoring function over finite configurations is defined as $s : \mathcal{C}_{all} \to \mathbb{N}_0$ such that $s(C) = \sum_{e \in C} cost(h(e)), C \in \mathcal{C}_{all}$. A scoring function is monotonic since $\forall C, C' \in \mathcal{C}_{all}, C \subseteq C' \implies s(C) \leq s(C')$.*

Definition 12 (Optimal Alignment Run). *Let $SN^{ext}_{\otimes}$ denote an extended SPN and let β denote its branching process. Let $s : \mathcal{C}_{all} \to \mathbb{N}_0$ be a scoring function over the finite configurations of β. An optimal alignment run $\gamma^{opt} \in \mathcal{U}_{SN^{ext}_{\otimes}}$ is an alignment run of $SN^{ext}_{\otimes}$ leading to a target event $(e^*)^{opt}$ such that for all the other alignment runs $\gamma \in \mathcal{U}_{SN^{ext}_{\otimes}}$ leading to target events $e^*_1, ..., e^*_n$ respectively, it holds that $s([(e^*)^{opt}]) \leq s([e^*_i]), 1 \leq i \leq n$.*

To obtain an optimal alignment run, we unfold $SN^{ext}_{\otimes}$ (using the ERV unfolding algorithm described in Sec. 3.2) until a target event is pulled out from the queue of events. At this point, the unfolding process can be terminated and a local distributed run associated with e^* is retrieved. This local distributed run is an alignment run as defined in Def. 10. Its alignment cost is given by $s([e^*])$.

Note that in the unfolding algorithm, an adequate order is used to order events in a queue. Using different adequate orders leads to different results, i.e., different complete distributed runs. Here we choose a *cost-based adequate order* that compares finite configurations based upon their costs.

Definition 13 (Cost-based Partial Order $\lhd_c$). *Let C, C' be two finite configurations of a branching process β of an extended SPN and s be the scoring function. Let $\gg$ be a total order on the events of β such that for any two events e and e', $e \gg e'$ if e was added earlier to the prefix than e'. Given a set E of events, $\phi(E)$ denotes the sequence of events that is ordered according to $\gg$. It is easy to see that $\gg$ is a well-founded relation. A cost-based partial order $\lhd_c$ is defined such that $C \lhd_c C'$ if and only if:*

- *$s(C) < s(C')$, or*
- *$s(C) = s(C')$ and $|C| < |C'|$, or*
- *$s(C) = s(C')$ and $|C| = |C'|$ and $\phi(C) \gg \phi(C')$ [2].*

[2] We say $\phi(E_1) \gg \phi(E_2)$ if $\phi(E_1)$ is lexicographically smaller than $\phi(E_2)$ with respect to the total order $\gg$

That $\lhd_c$ is irreflexive, asymmetric and transitive is obvious from the definition. Hence, $\lhd_c$ is also a strict partial order. In addition to ordering the priority queue, the chosen adequate order is used to identify cut-off events. For correctness proofs to work, i.e., for the ERV algorithm to correctly identify cut-off events, a cost-based order $\lhd_c$ defined as above must be *adequate* [13]. It can be proven that $\lhd_c$ satisfies the three properties of adequate orders (cf. Theorem 1 in Appendix, [28]) and hence can be used to correctly unfold extended SPNs with respect to the target marking.

Next, we present the $ERV[\lhd_c]$ (along with $ERV[\lhd_h]$ as discussed later in Sec. 3.3) *unfolding algorithm*, an on-the-fly variant of ERV algorithm [13] to produce optimal alignment runs. It is implemented as the *UnfoldSyncNet* procedure in Algorithm 1. It differs from the standard ERV unfolding algorithm in that it terminates when a target event is pulled out from the queue (based upon the chosen $stopAtFirst$ flag), returning an optimal alignment run as an output. Additionally, it uses $\lhd_c$ as its adequate order.

3.2 $ERV[\lhd_c]$ Unfolding Algorithm

Given an extended SPN $SN^{ext}_{\otimes} = ((P^{ext}_{\otimes}, T^{ext}_{\otimes}, F^{ext}_{\otimes}, \lambda^{ext}_{\otimes}), M^{ext}_{init,\otimes}, M^{ext}_{final,\otimes})$ and a boolean flag for $stopAtFirst$ to indicate if only one optimal alignment run is to be computed or all, we can obtain (all) optimal alignment run(s) (and thus, all optimal alignment orders) from the initial marking $M^{ext}_{init,\otimes}$ to the final marking $M^{ext}_{final,\otimes}$ using Algorithm 1.

Similar to the classic ERV unfolding algorithm, $ERV[\lhd_c]$ builds a prefix of events and conditions using a priority queue ordered by an adequate order ($\lhd_c$). Unlike the classic version, it adds steps (lines 8-13) to handle target events enabling the target marking. When such an event is retrieved, its local distributed run is stored in *alignmentRuns* (line 9). If only one alignment run is needed (line 11), the loop ends early, returning the result. Otherwise, the algorithm checks if the event's local configuration contains a cut-off event (line 14) using a lookup table $imarks$ for easy O(1) retrieval. If no cut-off exists, the prefix is extended with the event's postset conditions (Lines 15-16). Additionally, if e itself is a cutoff-event, it is added to the set of cutoff events (Lines 17-18). Any new possible extensions are calculated (line 20), added to the prefix, and $imarks$ is updated (line 21). The process repeats until the queue is empty.

The calculation of all possible extensions (lines 5 and 20) is the one crucial to implementation. It deals with the problem of finding events that are enabled by the entire set of input conditions while ensuring that they are not in causal or conflict relation with each other. Those identified events are thus added to the prefix and priority queue. Efficient conflict/causal detection while downsizing the number of combinations to check for is key to managing the combinatorial explosion inherent in this part of the algorithm. Roemer in his work [25] already proposed ways to optimize this part using intelligent data structures and procedures. In our work, we implement the same techniques.

Algorithm 1: UnfoldSyncNet - $ERV[\lhd_c]$ (and $ERV[\lhd_h]$, cf. Sec. 3.3)

Input: $SN^{ext}_{\otimes} = ((P^{ext}_{\otimes}, T^{ext}_{\otimes}, F^{ext}_{\otimes}, \lambda^{ext}_{\otimes}), M^{ext}_{init,\otimes}, M^{ext}_{final,\otimes})$, $stopAtFirst \in \mathbb{B}$
Output: $alignmentRuns \subseteq \mathcal{U}_{SN^{ext}_{\otimes}}$, $lowestCost \in \mathbb{N}_0$

```
1  cut-off, pe, finalEvents, β ← ∅        // Initialize the set of cutoff events,
     priority queue ordered by ◁_c (or ◁_h for ERV[◁_h]), the set of
     target events with minimal s([e]), and the prefix β = (B, E, F, h)
2  imarks ← {M_init,ext : ⊥}              // Lookup table of induced markings
     initialized with the initial marking induced by a dummy event ⊥
3  for p ∈ M_init,ext do                  // Add initial conditions to prefix
4  |  B ← B ∪ {a new condition mapped to place p}
5  calculate all possible extensions of B
6  while pe ≠ ∅ do
7  |  e ← Min{pe}  // Retrieve an event e s.t. [e] is minimal w.r.t. ◁_c
   |    (or ◁_h for the ERV[◁_h] variant)
8  |  if h(e) = t* then                                  // A target event found
9  |  |  alignmentRuns ← alignmentRuns ∪ {O_e}  // A local distributed run
   |  |    O_e of e (cf. Def. 7) is added
10 |  |  lowestCost ← s([e])
11 |  |  if stopAtFirst = true then
12 |  |  |  break
13 |  |  else continue
14 |  if [e] ∩ cut-off = ∅ then
15 |  |  for p ∈ h(e)• do            // Add e's postset conditions one by one
16 |  |  |  c ← a new condition mapped to p extend from event e to condition c in the
   |  |  |    prefix β
17 |  |  if e is a cut-off event w.r.t. ◁_c (or ◁_h for ERV[◁_h]) then      // check if
   |  |    Mark([e]) is already present in imarks
18 |  |  |  cut-off ← cut-off ∪ {e}
19 |  |  else
20 |  |  |  calculate all possible extensions of B
21 |  |  |  add Mark([e]) to imarks
22 return alignmentRuns, lowestCost
```

3.3 Directing the Unfolding

In the context of $ERV[\lhd_c]$ algorithm, we retrieve events from the queue based upon the total cost of mapped transitions in its local configuration, i.e., $s([e])$; equivalent to selecting events based upon the transitions fired so far in its local distributed run. For efficiency, we also consider an *underestimate* of the cost of transitions remaining till the target event is reached. Thus, when selecting events from the queue, we additionally favor those "closer" to t^*, resulting in a *directed unfolding* strategy. To this end, we use a *heuristic function* on configurations to guide the unfolding process towards a target configuration. Its value is an estimate of the distance (where the notion of distance is related to the total cost of transitions fired) between configurations. Thus, in the context of $ERV[\lhd_c]$ algorithm, orderings can be constructed upon the values of a function $f : \mathcal{C}_{all} \to \mathbb{N}_0$ composed of two parts $f(C) = s([e]) + est(C)$, where $s([e])$ is the total cost of mapped transitions of $[e]$ and $est(C)$, the heuristic value, estimates the distance from $Mark(C)$ to the target marking $[p^*]$. We use a heuristic function that exploits the marking equation [7] to underestimate the remaining cost to $[p^*]$.

Definition 14 (Heuristics-based Partial Order $\lhd_h$). *Let C, C' be two finite configurations of a branching process β of an SPN. Let $f(C) = s([e]) + est(C)$ be a function on finite configurations where s is a scoring function and est is a heuristic function. Let $\lhd_c$ be a cost-based adequate order over finite configurations. A heuristics-based partial order $\lhd_h$ is defined such that:*

$$C \lhd_h C' \text{ iff } \begin{cases} f(C) < f(C'), & \text{if } f(C) \neq f(C') \\ C \lhd_c C', & \text{otherwise.} \end{cases}$$

Given that the first part of f (in our case, s) is a monotonic function and *est* is an admissible heuristic function, Bonet et al. [6] proved that a partial order defined as above belongs to a family of *semi-adequate* orders, which still enables us to use the $ERV[\lhd_c]$ algorithm with the same definition of cut-off events. Here, we simply replace $\lhd_c$ in $ERV[\lhd_c]$ (Alg. 1) by $\lhd_h$ to obtain another algorithm variant, referred to as $ERV[\lhd_h]$ *algorithm*. The only changes in $ERV[\lhd_h]$ occur in Lines 7 and 17 where an adequate (semi-adequate, in this case) order is used.

3.4 Building and Visualizing Optimal Alignments from Optimal Alignment Runs

Building a u-alignment from an optimal alignment run, referred to as the *optimal u-alignment*, is a two-step process:

1. Construction of an optimal alignment order by using the events and conditions of the net corresponding to an optimal alignment run.
2. Decomposition of the optimal alignment order to obtain two partial orders, one corresponding to the p-trace and the other to a model run, while setting our alignment function at the same time. This gives us an optimal u-alignment, our final artefact.

In the first step, to build an optimal alignment order G_φ^{opt} from an optimal alignment run γ^{opt}, remove the target event e^*, map each event to a labeled node representing an alignment move, and create dependencies based on conditions, classifying them as log or model dependencies. In the second step, we decompose an alignment order into two partial orders while setting the alignment function. Fig. 6 shows the mechanism through an illustration. On top we see an optimal alignment run through the example extended SPN from Fig. 5 using $ERV[\lhd_c]$ or $ERV[\lhd_h]$. Below it, is the transformed optimal alignment order G_φ^{opt}. G_φ^{opt} is decomposed into different partial orders $G_{\varphi\downarrow 1}^{opt}$ and $G_{\varphi\downarrow 2}^{opt}$, i.e., $G_\varphi^{opt} \rightarrow G_{\varphi\downarrow 1}^{opt} \biguplus G_{\varphi\downarrow 2}^{opt}$. The alignment function φ relates synchronous nodes 'b' and 'c' in the two partial orders. Moreover, notice that $G_{\varphi\downarrow 1}^{opt}$ is isomorphic to the p-trace we computed our alignment for (observe the p-trace in Fig. 5). Thus, our final u-alignment is $(G_{\varphi\downarrow 1}^{opt}, G_{\varphi\downarrow 2}^{opt}, \varphi)$.

The decomposed partial orders $G_{\varphi\downarrow 1}^{opt}$ and $G_{\varphi\downarrow 2}^{opt}$ are visualized into chevron-based views as exemplified in Fig. 4a. This visualization method effectively highlights alignment between log and model events but falls short in identifying

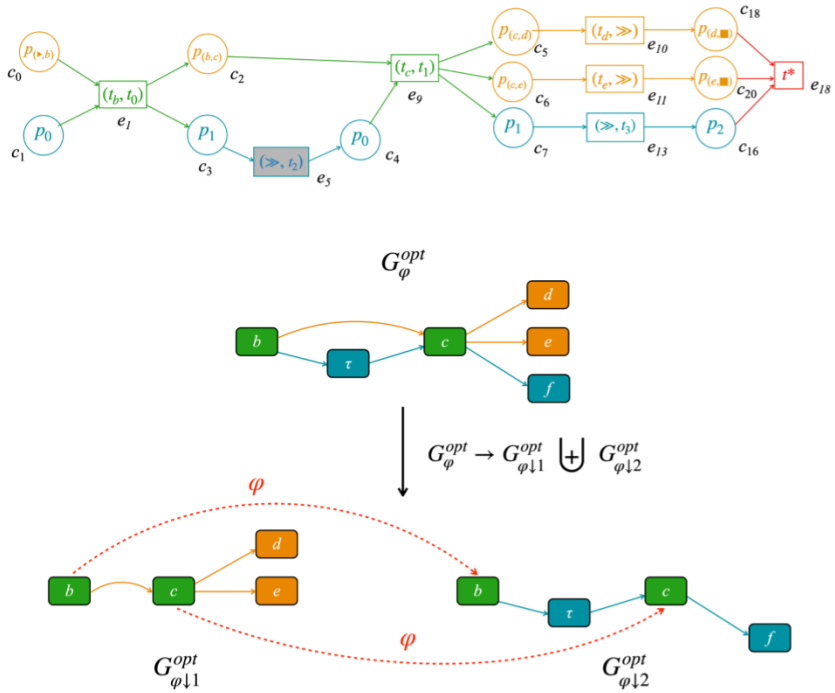


Fig. 6: Optimal alignment order (middle), derived from the alignment run (top) of the extended SPN $SN_{\otimes}^{ext}$, with partial order decomposition showing the u-alignment (bottom). $G_{\varphi\downarrow 1}^{opt}$ is isomorphic to the p-trace and $G_{\varphi\downarrow 2}^{opt}$ is the partial order from the model net run (cf. Fig. 5). Dotted red arrows show the mapping of the alignment function.

misalignments, particularly regarding dependencies. When comparing a model with a log variant, we aim to identify events present in reality but missing in the model (and vice versa) and to detect violations in event orders. These observations allow us to classify misalignments into two main categories. In the following, we assume a u-alignment $(G_{\varphi\downarrow 1}^{opt}, G_{\varphi\downarrow 2}^{opt}, \varphi)$ where $\prec_{\varphi\downarrow 1}^{opt-}$, $\prec_{\varphi\downarrow 2}^{opt-}$ are the edge relations of $G_{\varphi\downarrow 1}^{opt-}$ and $G_{\varphi\downarrow 2}^{opt-}$, respectively, where G^- denotes the transitive reduction of a DAG G.

1. **Missing events and dependencies**: Events and dependencies in the log but not in the model. Missing dependencies are relations $(m_1, m_2) \in \prec_{\varphi\downarrow 1}^{opt-}$ with $(m_1, m_2) \notin \prec_{\varphi\downarrow 2}^{opt-}$, representing log dependencies without corresponding model dependencies. Missing events are log moves of the form $(m_1, \gg)$.
2. **Undesired events and dependencies**: Events and dependencies in the log but not in the model. Missing dependencies are relations $(m_1, m_2) \in \prec_{\varphi\downarrow 2}^{opt-}$ with $(m_1, m_2) \notin \prec_{\varphi\downarrow 1}^{opt-}$, representing log dependencies without corresponding model dependencies. Missing events are log moves of the form $(\gg, m_2)$.

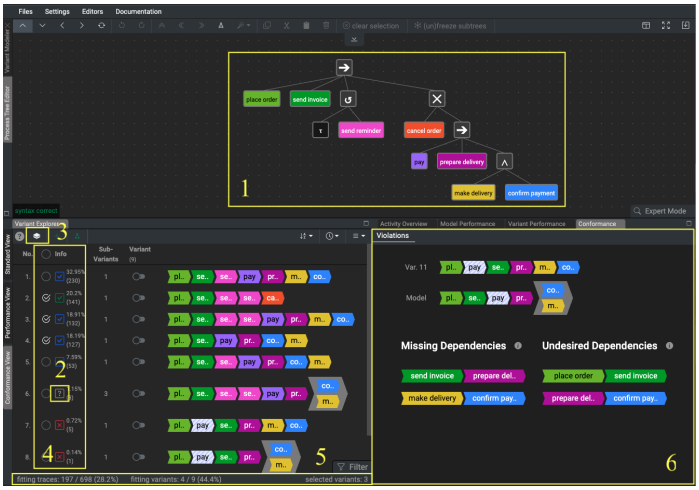


Fig. 7: Overview of the Conformance Analysis interface in Cortado

In $G^{opt}_{\varphi\downarrow 1}$ and $G^{opt}_{\varphi\downarrow 2}$ from Fig. 6, there are in total three missing dependencies corresponding to all dependencies visible in $G^{opt}_{\varphi\downarrow 1}$, three undesired dependencies corresponding to all dependencies in $G^{opt}_{\varphi\downarrow 2}$. Moreover, there are two missing events corresponding to 'd' and 'e' (orange nodes) and two undesired events corresponding to 'τ' and 'f' (blue nodes).

4 Tool Support

The $ERV[\lhd_c]$ and $(ERV[\lhd_h])$ algorithms have been implemented[3] in *Cortado* [27], a tool for interactive process discovery. Cortado traditionally checks conformance by computing all possible sequentializations of each partially ordered trace variant and aligning each sequence using the classic sequential alignment method [2]. The final alignments for each variant are aggregates of the sub-alignments. Our implementation replaces this sequentialization-based approach with unfolding-based alignments, adding new features.

We present a visual overview of the modified Conformance Analysis interface in Fig. 7. Once an initial model based on selected variants is available, users can compute unfolding-based alignments for new variants. The model, shown in the process tree editor (1), allows alignment computation by clicking one of the two buttons to (re)calculate conformance statistics (2), (3). After computation, one of three statuses appears (4): 1. variant fits the model, 2. variant does not fit, or 3. variant fits but with order deviations (our contribution). Conformance statistics is shown at the bottom (5). For a more detailed analysis, the Violations tab (6) in 'Conformance View' displays chevron-based concurrency visualizations

[3] `https://github.com/ariba-work/cortado`

of each selected variant and the lowest cost model run corresponding to that variant. Missing events are highlighted with stripes, and synchronous events can be paired by hovering over them. Missing and undesired dependencies are listed as pairs of chevrons, showing where dependencies should exist but don't in the model (e.g., 'send invoice' followed by 'prepare delivery') or vice versa.

5 Evaluation

We evaluate $ERV[\triangleleft_c]$ and $ERV[\triangleleft_h]$ against the classic approach to partial alignments based on A^* algorithm. Since we are particularly interested in the performance of our algorithm in settings of high parallelism, we split our experiments into two: 1). where we generate data artificially in order to control the degree of parallelism and compare different algorithms in the respective settings (Sec. 5.1), and 2). where we evaluate using real-life event logs (Sec. 5.2). In each part, we present our experimental setup and discuss the results. Both $ERV[\triangleleft_c]$ and $ERV[\triangleleft_h]$ are implemented in a forked repository[4] of the open-source software tool for process mining, *Cortado* [27]. Furthermore, to compare our results with the classic approach to partial alignments (referred to as *Classic PA* from hereon) [21], we implement their algorithm in a separate Github repository[5].

5.1 Using Artificial Event Logs

To examine the impact of parallelism on computation time, we conducted an experiment to test our hypothesis: as concurrency in models increases, the runtime overhead for exploring interleavings in *Classic PA*'s reachability graph grows larger than the overhead for calculating extensions in $ERV[\triangleleft_c]$ and $ERV[\triangleleft_h]$. Consequently, our proposed algorithms are expected to outperform *Classic PA* at higher parallelism levels.

Experimental Setup The experimental workflow involves generating 8 process trees with varying parallelism levels (0 % to 70%) using *PTAndLogGenerator* [19] such that all trees depict block-structured Petri nets without loops and/or duplicate labels. We then convert the trees into Petri nets, and reduce silent transitions, using different ProM[6] plugins. Event logs with 500 traces are simulated for each net using the plugin *StochasticPetriNets*. Noise in levels of 0 % to 50 % is inserted using an existing noise-insertion technique proposed in [1]. Finally, conformance checking is performed using $ERV[\triangleleft_c]$, $ERV[\triangleleft_h]$, and A^*-based *Classic PA* on each of the simulated event log.

[4] `https://github.com/ariba-work/cortado`
[5] `https://github.com/ariba-work/classic-pa`
[6] `https://promtools.org/`

Results In Fig. 8, we compare $ERV[\triangleleft_c]$, $ERV[\triangleleft_h]$, and *Classic PA* for varying noise and parallelism levels, focusing on average computation times. Results are shown for parallelism levels of 30%, 50%, and 70%, where significant differences were observed. The x-axis represents noise percentage in event logs, and the y-axis shows computation times in milliseconds, with linear regression lines illustrating trends. At 70% parallelism, $ERV[\triangleleft_h]$ outperforms other variants, showing robustness. On the other hand, $ERV[\triangleleft_h]$ starts to slow down even at 50%, due to larger prefixes generated, caused by increased events in high-parallelism models and the lack of heuristics to limit prefix expansion. At 30% parallelism, all variants perform almost similarly, with ERV variants excelling under higher noise levels. These findings support the hypothesis that unfolding-based alignments are more robust as parallelism increases.

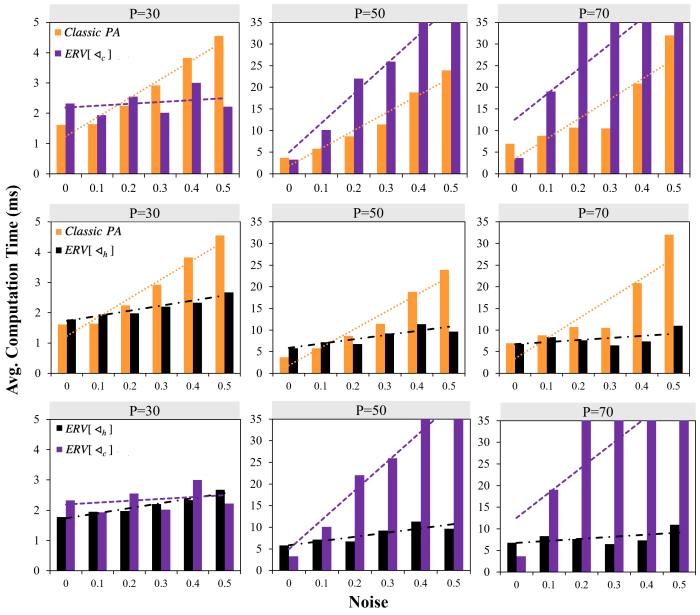


Fig. 8: Average computation time (ms) per percentage of noise, plotted per level of parallelism (30%, 50% and 70%). The level of parallelism is shown on top of the plots. In each row, the comparison is shown for two algorithms at a time.

5.2 Using Real Life Event Logs

Experimental Setup For this experiment, we selected the BPI Challenge 2012 (BPIC 12) [9] dataset, which contains partial event data suitable for validating our approach. The BPIC 12 dataset contains 262,200 events across 13,087 cases and 4,366 variants. It includes both start- and end-timestamps, with case lengths ranging from 3 to 175 events.

Table 1: Statistics of the models discovered by Infrequent Inductive Miner. Column names are abbreviated (#Pl: number of places, #Tr: number of transitions, #Si: number of silent transitions, ECyM: Extended Cyclomatic Metric)

BPIC 12 Model	#Pl	#Tr	#Si	ECyM
noise thr. 0.05	37	50	33	1899
noise thr. 0.5	26	27	11	289

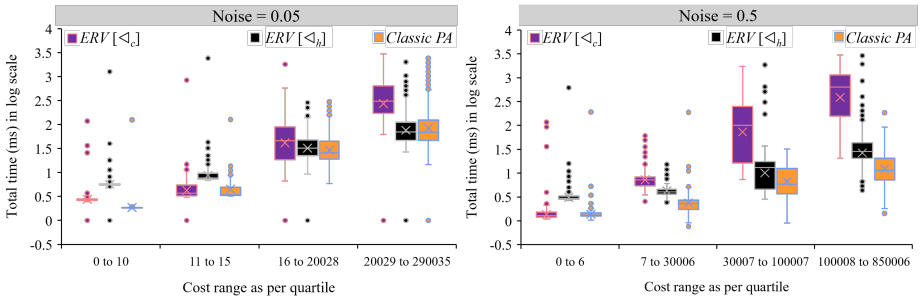


Fig. 9: Runtime comparison regarding deviation costs for different algorithms applied on models with different noise thresholds for BPIC 12 event log. Timeout for individual alignments for single traces is set to 3 seconds. $ERV[\triangleleft_c]$ generally performs the worst, while $ERV[\triangleleft_h]$ is more robust at the lowest noise threshold.

For conformance checking experiments, process models were discovered using the *Infrequent Inductive Miner* algorithm in ProM with noise thresholds of 5% and 50% where higher thresholds simplify models by filtering out rare traces. Tab. 1 shows the number of places, transitions, silent transitions and additionally the *Extended Cyclomatic Metric* (ECyM) [20] of the mined models. This metric is calculated using the reachability graph of Petri nets where higher values indicate more reachability paths.

Results In Fig. 9, we compare the runtime of $ERV[\triangleleft_c]$, $ERV[\triangleleft_h]$, and *Classic PA* across cost ranges and noise thresholds. Box plots show that runtime increases approximately exponentially with deviation costs, with lower noise threshold models taking longer. In both plots, it is apparent that *Classic PA* performs better for lower deviation costs. At 0.5 noise threshold, *Classic PA* still outperforms both *ERV* variants at higher deviation costs. This can be attributed to the higher number of non-synchronous transitions in the SPN of a higher noise threshold model, which necessitates exploring more equal-cost events. This results in broader prefixes and increased computation times. At 0.05 noise, *Classic PA* performs better for low deviation costs (0-10), while $ERV[\triangleleft_h]$ is slower but excels for higher costs. The difference between $ERV[\triangleleft_h]$ and *Classic PA* is not *so* apparent in this plot since runtime in the plots discussed so far does not take into account the traces for which alignment compu-

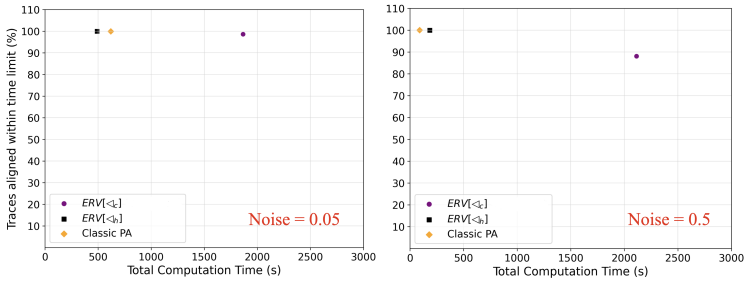


Fig. 10: Percentage of traces aligned within time limit by different algorithms, along with the total computation time. The time limit applies to individual traces, and total time is for aligning the entire BPIC 12 event log with its model at various noise thresholds.

tations timed out (> 3 seconds). However, the total computation for this model differs by nearly 130 seconds. As shown in Fig. 10, $ERV[\lhd_h]$ and *Classic PA* align nearly all traces across thresholds, while $ERV[\lhd_c]$ frequently times out at higher thresholds.

6 Conclusion

This paper presented Petri net unfolding as an efficient solution for computing alignments from partially-ordered event data, overcoming the state space explosion of traditional reachability graph-based methods. Our approach simplifies alignment computation into a single step, avoiding the need for reconstructing alignments from totally-ordered sequences. We also introduce a graph-based visualization for unfolding-based alignments, enhancing the interpretability of log-model relationships. Experimental results show that the computation of unfolding-based alignments outperforms traditional alignment methods in high-parallelism and complex models. While there is some computational overhead due to the calculation of possible extensions in low-parallelism settings, the runtime and robustness of our proposed algorithms in handling concurrency validate its effectiveness. Moreover, with our visualization technique, it is possible to obtain diagnostics over missing and undesired events and dependencies on partially-ordered alignments that have not been possible otherwise. Implemented in *Cortado*, this work provides both theoretical and practical contributions to conformance analysis. Future work will focus on optimizing $ERV[\lhd_c]$ and $ERV[\lhd_h]$, exploring heuristics and caching techniques.

References

1. van der Aa, H., Rebmann, A., Leopold, H.: Natural language-based detection of semantic execution anomalies in event logs. Inf. Syst. **102**, 101824 (2021), `https://doi.org/10.1016/j.is.2021.101824`

2. van der Aalst, W.M.P., Adriansyah, A., van Dongen, B.F.: Replaying history on process models for conformance checking and performance analysis. WIREs Data Mining Knowl. Discov. **2**(2), 182–192 (2012), https://doi.org/10.1002/widm.1045
3. Adriansyah, A., van Dongen, B.F., van der Aalst, W.M.: Memory-efficient alignment of observed and modeled behavior. BPM Center Report **3**, 1–44 (2013)
4. Adriansyah, A., Munoz-Gama, J., Carmona, J., van Dongen, B.F., van der Aalst, W.M.P.: Alignment Based Precision Checking. In: Rosa, M.L., Soffer, P. (eds.) Business Process Management Workshops - BPM 2012 International Workshops, Tallinn, Estonia, September 3, 2012. Revised Papers. Lecture Notes in Business Information Processing, vol. 132, pp. 137–149. Springer (2012), https://doi.org/10.1007/978-3-642-36285-9_15
5. Benveniste, A., Fabre, E., Haar, S., Jard, C.: Diagnosis of asynchronous discrete-event systems: a net unfolding approach. IEEE Trans. Autom. Control. **48**(5), 714–727 (2003), https://doi.org/10.1109/TAC.2003.811249
6. Bonet, B., Haslum, P., Hickmott, S.L., Thiébaux, S.: Directed Unfolding of Petri Nets. Trans. Petri Nets Other Model. Concurr. **1**, 172–198 (2008), https://doi.org/10.1007/978-3-540-89287-8_11
7. Carmona, J., van Dongen, B.F., Solti, A., Weidlich, M.: Conformance Checking - Relating Processes and Models. Springer (2018), https://doi.org/10.1007/978-3-319-99414-7
8. Chatain, T., Khomenko, V.: On the well-foundedness of adequate orders used for construction of complete unfolding prefixes. Inf. Process. Lett. **104**(4), 129–136 (2007), https://doi.org/10.1016/j.ipl.2007.06.002
9. van Dongen, B.: BPI Challenge 2012 (2012). https://doi.org/10.4121/UUID:3926DB30-F712-4394-AEBC-75976070E91F, https://data.4tu.nl/articles/_/12689204/1
10. Dumas, M., Rosa, M.L., Mendling, J., Reijers, H.A.: Fundamentals of Business Process Management, Second Edition. Springer (2018), https://doi.org/10.1007/978-3-662-56509-4
11. Engelfriet, J.: Branching Processes of Petri Nets. Acta Informatica **28**(6), 575–591 (1991), https://doi.org/10.1007/BF01463946
12. Esparza, J.: Model Checking Using Net Unfoldings. Sci. Comput. Program. **23**(2-3), 151–195 (1994), https://doi.org/10.1016/0167-6423(94)00019-0
13. Esparza, J., Römer, S., Vogler, W.: An Improvement of McMillan's Unfolding Algorithm. Formal Methods Syst. Des. **20**(3), 285–310 (2002), https://doi.org/10.1023/A:1014746130920
14. Esparza, J., Schröter, C.: Unfolding Based Algorithms for the Reachability Problem. Fundam. Informaticae **47**(3-4), 231–245 (2001), http://content.iospress.com/articles/fundamenta-informaticae/fi47-3-4-05
15. García-Bañuelos, L., van Beest, N.R., Dumas, M., Rosa, M.L., Mertens, W.: Complete and interpretable conformance checking of business processes. IEEE Transactions on Software Engineering **44**(3), 262–290 (2018). https://doi.org/10.1109/TSE.2017.2668418
16. Hart, P.E., Nilsson, N.J., Raphael, B.: A formal basis for the heuristic determination of minimum cost paths. IEEE Trans. Syst. Sci. Cybern. **4**(2), 100–107 (1968), https://doi.org/10.1109/TSSC.1968.300136
17. van Hee, K.M., Sidorova, N., van der Werf, J.M.E.M.: Business Process Modeling Using Petri Nets. Trans. Petri Nets Other Model. Concurr. **7**, 116–161 (2013), https://doi.org/10.1007/978-3-642-38143-0_4

18. Hickmott, S.L., Rintanen, J., Thiébaux, S., White, L.B.: Planning via Petri Net Unfolding. In: Veloso, M.M. (ed.) IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007. pp. 1904–1911 (2007), http://ijcai.org/Proceedings/07/Papers/307.pdf
19. Jouck, T., Depaire, B.: Generating Artificial Data for Empirical Analysis of Control-flow Discovery Algorithms - A Process Tree and Log Generator. Bus. Inf. Syst. Eng. **61**(6), 695–712 (2019), https://doi.org/10.1007/s12599-018-0541-5
20. Lassen, K.B., van der Aalst, W.M.P.: Complexity metrics for workflow nets. Inf. Softw. Technol. **51**(3), 610–626 (2009). https://doi.org/10.1016/J.INFSOF.2008.08.005, https://doi.org/10.1016/j.infsof.2008.08.005
21. Lu, X., Fahland, D., van der Aalst, W.M.P.: Conformance Checking Based on Partially Ordered Event Data. In: Fournier, F., Mendling, J. (eds.) Business Process Management Workshops - BPM 2014 International Workshops, Eindhoven, The Netherlands, September 7-8, 2014, Revised Papers. Lecture Notes in Business Information Processing, vol. 202, pp. 75–88. Springer (2014), https://doi.org/10.1007/978-3-319-15895-2_7
22. McMillan, K.L.: Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In: von Bochmann, G., Probst, D.K. (eds.) Computer Aided Verification, Fourth International Workshop, CAV '92, Montreal, Canada, June 29 - July 1, 1992, Proceedings. Lecture Notes in Computer Science, vol. 663, pp. 164–177. Springer (1992), https://doi.org/10.1007/3-540-56496-9_14
23. Nielsen, M., Plotkin, G.D., Winskel, G.: Petri Nets, Event Structures and Domains, Part I. Theor. Comput. Sci. **13**, 85–108 (1981), https://doi.org/10.1016/0304-3975(81)90112-2
24. Reisig, W.: Understanding Petri Nets - Modeling Techniques, Analysis Methods, Case Studies. Springer (2013), https://doi.org/10.1007/978-3-642-33278-4
25. Römer, S.: Theorie und Praxis der Netzentfaltungen als Grundlage für die Verifikation nebenläufiger Systeme. Ph.D. thesis, Technical University Munich, Germany (2000), http://tumb1.biblio.tu-muenchen.de/publ/diss/in/2000/roemer.pdf
26. Schuster, D., Schade, L., van Zelst, S.J., van der Aalst, W.M.P.: Visualizing Trace Variants from Partially Ordered Event Data. In: Munoz-Gama, J., Lu, X. (eds.) Process Mining Workshops - ICPM 2021 International Workshops, Eindhoven, The Netherlands, October 31 - November 4, 2021, Revised Selected Papers. Lecture Notes in Business Information Processing, vol. 433, pp. 34–46. Springer (2021), https://doi.org/10.1007/978-3-030-98581-3_3
27. Schuster, D., van Zelst, S.J., van der Aalst, W.M.P.: Cortado - An Interactive Tool for Data-Driven Process Discovery and Modeling. In: Buchs, D., Carmona, J. (eds.) Application and Theory of Petri Nets and Concurrency - 42nd International Conference, PETRI NETS 2021, Virtual Event, June 23-25, 2021, Proceedings. Lecture Notes in Computer Science, vol. 12734, pp. 465–475. Springer (2021), https://doi.org/10.1007/978-3-030-76983-3_23
28. Siddiqui, A., van der Aalst, W.M.P., Schuster, D.: Computing alignments for partially-ordered traces through petri net unfoldings (2025), https://arxiv.org/abs/2504.00550
29. Valmari, A.: The state explosion problem, pp. 429–528. Springer Berlin Heidelberg, Berlin, Heidelberg (1998). https://doi.org/10.1007/3-540-65306-6_21, https://doi.org/10.1007/3-540-65306-6_21