Complete and Interpretable Conformance Checking of Business Processes

Luciano García-Bañuelos, Nick R.T.P. van Beest, Marlon Dumas and Marcello La Rosa

Abstract—This article presents a method for checking the conformance between an event log capturing the actual execution of a business process, and a model capturing its expected or normative execution. Given a business process model and an event log, the method returns a set of statements in natural language describing the behavior allowed by the process model but not observed in the log and vice versa. The method relies on a unified representation of process models and event logs based on a well-known model of concurrency, namely event structures. Specifically, the problem of conformance checking is approached by folding the input event log into an event structure, unfolding the process model into another event structure, and comparing the two event structures via an error-correcting synchronized product. Each behavioral difference detected in the synchronized product is then verbalized as a natural language statement. An empirical evaluation shows that the proposed method scales up to real-life datasets while producing more concise and higher-level difference descriptions than state-of-the-art conformance checking methods.

Index Terms—process mining, conformance checking, process model, event log, event structure, Petri net.

1 Introduction

PROCESS mining [1] is a family of methods concerned with the analysis of event logs produced by software systems that support the execution of business processes. Process mining methods allow analysts to understand how a business process is actually executed, and to detect and analyze deviations with respect to performance objectives or normative pathways.

The main input of a process mining method is an *event log* of a business process. An event log is a set of *traces*, each consisting of the sequence of *event records* produced by one execution of the process (a.k.a. a *case*). An event in a trace denotes the start, end, abortion or other relevant state change of a task. As a minimum, an event record contains a timestamp, an identifier of the case to which the event refers, and an *event class*, that is, a reference to a task in the process under observation.

This article is concerned with a recurrent process mining operation, namely *business process conformance checking* [1]. Given an event log recording the actual execution of a business process, and given a process model capturing its expected or normative executions, the goal of conformance checking is to pinpoint and to describe any differences between the behavior observed in the event log and the behavior captured in the process model.

Previous approaches to business process conformance checking are designed to identify the number and the location of the differences between the model and the traces in the log, rather than providing a diagnosis that would allow analysts to understand these differences. For example, these approaches can identify that there is a state in the process model where the model has additional behavior not observed in the log, but without describing this additional behavior. Similarly, these approaches can find points in a trace where the behavior observed in the trace deviates from the model, but without explaining what behavior the trace has that the model does not.

This article addresses these limitations by proposing a method for business process conformance checking that: (i) identifies all differences between the behavior in the model and the behavior in the log; and (ii) describes each difference via a natural language statement capturing task occurrences, behavioral relations or repeated behavior captured in the model but not observed in the log, or vice-versa.

The method relies on a well-known model of concurrency, namely prime event structures, extended with a notion of "repeated behavior". In this context, event structures serve as a unified representation of the input process model and the event log [2]. In other words, both the input process model and the event log are transformed into event structures. The two resulting event structures are then compared via an error-correcting (partial) synchronized product that identifies all the behavioral differences between model and log. This synchronized product is used as a basis for enumerating the behavioral differences and verbalizing them as natural language statements. The choice of event structures is driven by the fact that they allow us to characterize the detected differences in terms of behavioral relations corresponding to well-accepted elementary workflow patterns [3], namely causality (sequence pattern), concurrency (parallel split & syncronization patterns), conflict (exclusive choice and merge patterns) and repetition (cycles).

As a running example, Fig 1 presents a model of a loan application process using the Business Process Model and Notation (BPMN). The process starts with the receipt of a loan application. Two tasks are performed in parallel – "Check credit history" and "Check income sources". Next the application is assessed, leading to two possible branches.

- *L. García-Bañuelos and M. Dumas are with the University of Tartu, Estonia.*
  E-mail: {luciano.garcia, marlon.dumas}@ut.ee
- *N.R.T.P. van Beest is with NICTA Queensland, Australia.*
  E-mail: nick.vanbeest@nicta.com.au
- *M. La Rosa is with the Queensland University of Technology, Australia.*
  E-mail: m.larosa@qut.edu.au

In one branch, a credit offer is made to the customer and the process ends. In the other, a negative decision is communicated to the customer. In some cases, the customer is asked to provide additional information. Once the customer provides this information, the application is assessed again.

Consider now a log {ABCDEH, ACBDEH, ABDEH, ABCDFH, ACBDFH, ABDFH} where, for convenience, traces are represented as words over the single-letter labels A-H shown in the top-right corner of each model element in Fig 1. Given this log, the proposed method identifies the following differences: (i) task C is optional in the log; (ii) the cycle including IGDF is not observed in the log. The first statement characterizes the behavior observed in the log but not in the model, while the second characterizes the behavior captured in the model but not observed in the log.

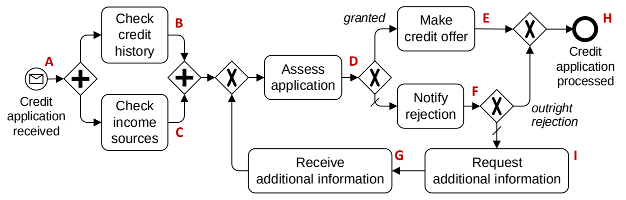


Fig. 1: Example: loan application process model

The rest of the article is structured as follows. Section 2 discusses previous work on conformance checking. Section 3 introduces event structures so as to make the article self-contained. Section 4 gives an overview of the proposed conformance checking method, which relies on the construction of event structures from process models and from event logs (Section 5), a partially synchronized product of event structures (Section 6) and a method for extracting and verbalizing differences from the partially synchronized product (Section 7). Finally, Section 8 presents an empirical evaluation while Section 9 summarizes the contributions and outlines directions for future work.

2 Background and Related Work

The purpose of conformance checking is to identify two types of discrepancies:

1) Unfitting log behavior: behavior observed in the log that is not allowed by the model.
2) Additional model behavior: behavior allowed in the model but never observed in the log.

The identification of unfitting log behavior has been approached using two types of methods: *replay* and *trace alignment*.

Replay methods take as input one trace at a time and determine the maximal prefix of the trace (if any) that can be parsed by the model. When it is found that a prefix can no longer be parsed by the model, this "parsing error" is corrected either by skipping the event and continuing with the next one, or by changing the state of execution of the process model to one where the event in question can be replayed. A representative replay method is *token fitness* [4], which replays each trace in the log against the process model (represented as a Petri net) and identifies two types of errors: (i) *missing tokens*: how many times a token needs to be added to a place in the Petri net in order to correct a replay (parsing) error; and (ii) *remaining tokens*: how many tokens remain in the Petri net once a trace has been fully replayed. De Medeiros [5] proposed two extensions of this technique: *continuous parsing* and *improved continuous semantics fitness*. These extensions rely on the same principle, but they sacrifice completeness of the output in order to gain in performance. Another extended replay method has been proposed by vanden Broucke et al. [6]. This method starts by decomposing the process model into single-entry single-exit (SESE) regions, so that the replay can be done on each region separately. This decomposition allows the replay to be performed independently in each region, thus making it more scalable and suitable for real-time conformance analysis. It also produces more localized feedback, since each replay error can be traced back to a specific region. An analysis of different approaches to decompose the model into regions for the purpose of measuring unfitting log behavior is provided in [7].

A general limitation of replay methods is that error recovery is performed locally each time that an error is encountered. Hence, these methods might not identify the minimum number of errors that can explain the unfitting log behavior. This limitation is addressed by *trace alignment fitness* [8], [9]. This latter method identifies, for each trace in the log, the closest corresponding trace parsed by the model and computes an alignment showing the points of divergence between these two traces. The output is a set of pairs of "aligned traces". Each pair shows a trace in the log that does not match exactly a trace in the model, together with the corresponding closest trace(s) produced by the model. For example given the model shown in Fig. 1 and log {ABCDEH, ACBDEH, ABDEH, ABCDFH, ACBDFH, ABDFH}, trace alignment produces two aligned traces: one between trace ABDEH of the log and trace ABCDEH of the model; and another between trace ABDFH of the log and trace ABCDFH of the model. From this, the user needs to infer that the unfitting log behavior is that task C is optional in the log but always executed in the model.

The number of aligned traces produced by the above method is often too large to be explored exhaustively. Visualizations are proposed to cope with large sets of aligned traces. Fundamentally though, the limitation of trace alignment fitness (also shared with replay methods) is that it identifies differences at the level of individual traces rather than at the level of behavioral relations observed in the log but not captured in the model.

Approaches to identify additional behavior include those based on *negative events* and those based on *prefix automata*. An exemplar of the former approach is *negative event precision* [10]. This approach works by inserting inexistent (so-called *negative*) events to enhance the traces in the log. A negative event is inserted after a given prefix of a trace if this event is never observed preceded by that prefix anywhere in the log. For example, if event c is never observed after prefix ab, then c can be inserted as a negative event after ab. The traces extended with negative events are then replayed on the model. If the model can parse some of the negative events, it means that the model has additional behavior. This approach to detect additional behavior is however heuristic: it does not guarantee that all additional behavior is identified. An extension of this approach [11] addresses its scalability limitations and can also better deal with noisy logs, but again it does not guarantee that all additional behavior is identified.

An approach to detect the presence of additional model behavior based on prefix automata is outlined in [12]. The first step in this approach is to generate a prefix automaton that fully represents the entire log. Each state in this automaton corresponds to a unique trace prefix. For each state S_a in the automaton, the corresponding trace prefix is replayed in the model in order to identify a matching state S_m in the model. The set of tasks enabled in S_m is then determined. If there is a task enabled in state S_m in the model but not in state S_a in the automaton, this is marked as additional model behavior by adding a so-called "escaping edge" to state S_a of the automaton. This edge is labelled with the task in question and considered as a sink state in the automaton. The edge represents the fact that in state S_a, there is additional behavior in the model that is not observed in the log. This basic approach suffers from two limitations: (i) it cannot handle tasks with duplicate labels in the model nor tasks without labels (so-called *invisible tasks*), which are needed to capture decisions based on the evaluation of data conditions; and (ii) it assumes that all traces in the log fit the model. These limitations are addressed in [13]. The idea of this latter approach is to first calculate an alignment between the traces in the log and traces in the model, using the trace alignment technique mentioned above. This leads to a log with aligned traces, which include invisible tasks. These aligned traces and any prefix thereof can always be replayed by the model. The prefix automaton is then computed from the model-projection of the aligned traces rather than from the original traces. The automaton is then used to detect "escaping edges" in the same way as described above.

The approaches described in [12] and [13] are able to pinpoint states in the model where behavior is allowed that is not present in the log. However, they cannot characterize the additional allowed behavior, beyond stating that the additional behavior starts with the execution of a given task. For example, given the model in Fig. 1 and log {ABCDEH, ACBDEH, ABDEH, ABCDFH, ACBDFH, ABDFH}, these approaches identify an escape edge after a prefix in the automaton that finishes with "Notify Rejection". However, they do not detect that there is repetitive behavior in the model whereas there is no such repeating behavior in the log (e.g. in the model task "Assess application" can be repeated whereas this repetition is not observed in the log).

In summary, existing approaches to detect unfitting log behavior operate at the level of individual traces and hence provide low-level feedback (parsing errors on pairs of aligned traces). Meanwhile, approaches to detect additional model behavior are able to pinpoint the presence of additional behavior but cannot describe it. In particular, they are not able to detect where the additional behavior ends. The method proposed in this article addresses these limitations by detecting all unfitting and additional behavior and describing it in terms of events (tasks) and relations captured in the model but not observed in the log and vice-versa.

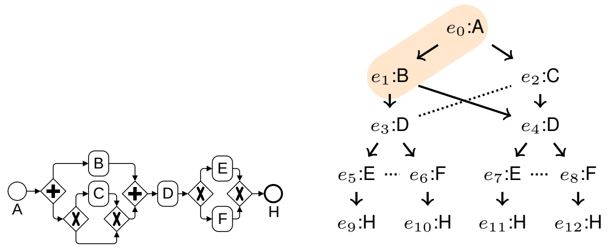
We previously sketched the idea of using event structures for conformance checking in an extended abstract [14]. The present article develops this idea in detail with definitions and algorithms, and provides an empirical evaluation.

3 Event Structures

A Prime Event Structure (PES) [15] is a graph of events, where an event e represents the occurrence of an action (e.g. a task) in the modeled system (e.g. a business process). If a task occurs multiple times in an execution, each occurrence is represented by a different event. The order of occurrence of events is defined via three binary relations: i) *Causality* ($e < e'$) indicates that event e is a prerequisite for e'; ii) *Conflict* ($e \# e'$) implies that e and e' cannot occur in the same execution; iii) *Concurrency* ($e \parallel e'$) indicates that no order can be established between e and e'. Formally:

Definition 1. A *Labeled Prime Event Structure* is the tuple $\mathcal{E} = (E, \leq, \#, \lambda)$ where E is the set of events, $\leq \subseteq E \times E$ is a partial order, referred to as *causality*, $\# \subseteq E \times E$ is an irreflexive, symmetric relation, referred to as *conflict*, and $\lambda : E \rightarrow \mathcal{L} \cup \{\tau\}$ is a labeling function.
The irreflexive version of causality is denoted as $<$. The *concurrency relation*, in turn, can be derived from causality and conflict relations, i.e. $\parallel \triangleq E \times E \backslash (< \cup <^{-1} \cup \#)$. Moreover, conflict is "inherited" via the causality relation, i.e. $e \# e' \wedge e' \leq e'' \Rightarrow e \# e''$ for $e, e', e'' \in E$.

Fig. 2 presents a variant of the process model introduced in Fig. 1[1] and its corresponding PES $\mathcal{E}^1$. In the PES, nodes are labelled by an event identifier and a task label, e.g. "e_2:C" tells us that event e_2 represents an occurrence of task "C". For brevity, we will often omit the event label. We distinguish between observable and silent (or τ) events. In the following, we write $C|_\lambda$ to denote the restriction of configuration C to its subset of observable events, i.e. $C|_\lambda \triangleq \{e \in C \mid \lambda(e) \neq \tau\}$. Causal dependencies are drawn as solid arcs, whereas conflict relations as dotted edges. In order to simplify the graphical representation of an event structure, transitive causal and hereditary conflict relations are not drawn. Every two events that appear neither directly nor transitively connected are considered to be concurrent.



(a) BPMN (b) Prime event structure $\mathcal{E}^1$

Fig. 2: Sample process model and event structure

An execution context (i.e. a "state") in an event structure is described in terms of sets of events that can occur together in an execution of the underlying system. Such a set of events is called a *configuration*. Formally, we say that a set of events $C \subseteq E$ is a configuration iff (i) C is causally closed: for each event $e \in C$, the configuration C also contains all causal predecessors of e, i.e. $\forall e' \in E, e \in C : e' \leq e \Rightarrow e' \in C$, and (ii) C is conflict free: C does not contain any pair

1. In this variant of the process, task C can be skipped – e.g. "Check income sources" may not be required for existing customers – and applicants cannot request for reviewing a rejected application.

of events in mutual conflict, i.e. $\forall e, e' \in C \Rightarrow \neg(e \# e')$. An event e is an extension of a configuration C, denoted $C \oplus e$, if and only if $C \cup \{e\}$ is also a configuration. We denote by $\mathcal{F}(\mathcal{E})$ the set of all the configurations of $\mathcal{E}$ and by $\mathcal{F}_{\mathrm{m}}(\mathcal{E})$ the set of configurations that are maximal with respect to set inclusion. Moreover, we define the concept of *set of possible extensions of a configuration* C as $\mathrm{PE}(C) \triangleq \{e \mid C \oplus e\}$.

For example, let C_1 be the set of events $\{e_0, e_1\}$ – highlighted in Fig. 2(b). Intuitively, the configuration C_1 of $\mathcal{E}^1$ represents the state of computation in which tasks "A" and "B" have occurred. Moreover, given the configuration C_1 we say that the computation can evolve by executing an event from the set $\{e_2, e_3\}$, given that this set of events corresponds to the possible extensions of C_1, that is $\mathrm{PE}(C_1)$. Now, if consider the occurrence event e_3, we would have to consider a new configuration, say $C_2 = \{e_0, e_1, e_3\}$, which we can also denote as $C_1 \oplus e_3$. Note that in context of configuration C_2 the occurrence of e_2 is no longer possible because the event e_3 is in conflict with e_2. Finally, in this same example the set of maximal configurations is $\mathcal{F}_{\mathrm{m}}(\mathcal{E}^1) = \{\{e_0, e_1, e_3, e_5, e_9\}, \{e_0, e_1, e_3, e_6, e_{10}\}, \{e_0, e_1, e_2, e_4, e_7, e_{11}\}, \{e_0, e_1, e_2, e_4, e_8, e_{12}\}\}$.

We use the term *local configuration* of an event e to refer to $\lceil e \rceil \triangleq \{e' \mid e' \leq e\}$, and the term *strict causes* of an event to refer to $\lceil e) \triangleq \lceil e \rceil \setminus \{e\}$. Finally, we say that events e_1 and e_2 are in *immediate conflict*, denoted $e_1 \#_\mu e_2$, if and only if $e_1 \# e_2$ and they are both possible extensions of the same configurations. Formally, the latter property can be verified by checking if $\lceil e_1) \cup \lceil e_2 \rceil$ and $\lceil e_1 \rceil \cup \lceil e_2)$ are both configurations or not.

4 Conformance Checking Method

The proposed method takes as input a process model captured in the standard BPMN language and an event log (cf. Fig. 3). In order to leverage Petri net-based techniques for constructing event structures, the input process model is first converted into a Petri net using the transformation proposed in [16]. The resulting Petri net is then unfolded into a prime event structure (cf. PES_m in Fig. 3) using Petri net unfolding techniques [17]. Each event in the resulting PES corresponds to an occurrence of a task in the input process model. The procedure for constructing an event structure from a Petri net is further outlined in Section 5.1.

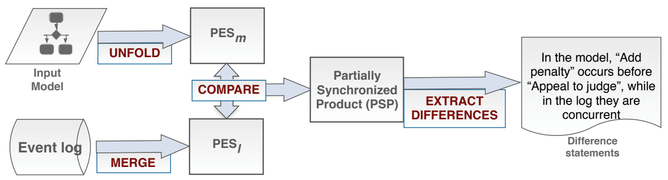

Input Model
UNFOLD
PES_m
COMPARE
Event log
MERGE
PES_l
Partially Synchronized Product (PSP)
EXTRACT DIFFERENCES
In the model, "Add penalty" occurs before "Appeal to judge", while in the log they are concurrent
Difference statements

Fig. 3: Overall view of the method

Meanwhile, the input event log is transformed into another prime event structure (cf. PES_l in Fig. 3) by first transforming the set of traces in the log into a set *partially-ordered runs* and then "prefix-merging" the resulting set of runs. The procedure for constructing an event structure from a log is elaborated upon in Section 5.2.

Given the prime event structures PES_m and PES_l obtained from the model and the log respectively, we compute a so-called Partially Synchronized Product (PSP) of the two event structures. In a nutshell, a PSP is a representation of a synchronized traversal of two input PESs, such that when a discrepancy between the PESs is detected, it is explicitly recorded and the traversal resumes from a"suitable" configuration in each of the two PESs. The procedure for calculating the PSP is presented in Section 6.

If two event structures PES_m and PES_l have a behavioral difference of type unfitting log behavior, this difference will be captured in a node of the PSP. Thus, we can enumerate all unfitting log behavior by traversing their PSP. To expose additional model behavior, we define a notion of coverage of a PES extracted from a model by a PES extracted from a log. The parts of PES_m not covered by PES_l can then be isolated and enumerated.

Differences between two event structures can be of several types. For example, one type of difference is that a task t is always executed according the model, but it is skipped in some traces in the log. Another type of difference occurs when two tasks t_1 and t_2 are causally related in the model (e.g. t_1 occurs always before t_2), but the corresponding events appear in any order in the log (i.e. sometimes t_1 occurs before t_2, sometimes the other way around). In order to generate an interpretable difference diagnosis from the PSP, we define a set of disjoint and complete mismatch patterns, as well as rules to verbalize each mismatch pattern as a natural language statement. The patterns and their verbalization are presented in Section 7.

5 Construction of Event Structures

This section shows how event structure are derived from a Petri net on the one hand, and from an event log on the other.

5.1 From Petri nets to PES

A Petri net is a bipartite graph, consisting of transitions (rectangles), places (hollow circles), tokens (filled circles) and arcs. A transition represents a system action (e.g. a task). Each transition has a set of input places and a set of output places. At a given point in the execution of a Petri net, a place can hold a number of tokens. The distribution of tokens across places on the net is called a *net marking*. A transition is enabled and can "fire" when all its input places has at least one token. When a transition fires, one token is removed from each input place and one token is put into each output place. A Petri net is called *safe* iff in every possible marking each place holds at most one token.

Definition 2. A tuple (P, T, F, λ) is a *labeled Petri net*, where P is a set of *places*, T is a set of *transitions*, with $P \cap T = \emptyset$, $F \subseteq (P \times T) \cup (T \times P)$ is a set of arcs, and $\lambda : P \cup T \to \mathcal{L} \cup \{\tau\}$ a labeling function. A *net marking* $M : P \to \mathbb{N}_0$ is a function that associates a place $p \in P$ with a natural number (viz., place tokens). A *marked net* $N = (P, T, F, M_0)$ is a Petri net (P, T, F) together with an *initial marking* M_0.

Fig. 4 presents a Petri net corresponding to the BPMN process model in Fig. 2(a). Each task, start and end event is mapped into a transition, which carries the same label as the corresponding BPMN construct. The Petri net additionally

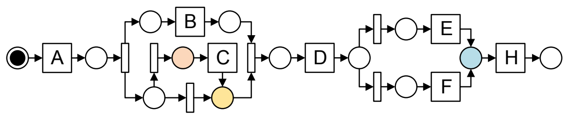


Fig. 4: Petri net for BPMN model in 2(a)

contains some unlabeled transitions. These transitions correspond to parallel gateways in the BPMN process model as well as branches stemming out of decision gateways. The materialization of gateways and decision branches as unlabelled (a.k.a. silent or τ) transitions is an artifact of the transformation from BPMN to Petri nets [16]. These unlabeled transitions will be eliminated during the construction of the event structure as discussed later.

An alternative approach to represent the dynamics of a system (e.g. a business process) is by means of a Petri net that explicitly represents all partially-ordered runs of the system. A run of a system is a partially-ordered set of events that can occur in one execution thereof. When all the runs are prefix-merged into a single Petri net, the latter is called a branching process [18]. Fig. 5 presents the branching process of the marked net in Fig. 4.

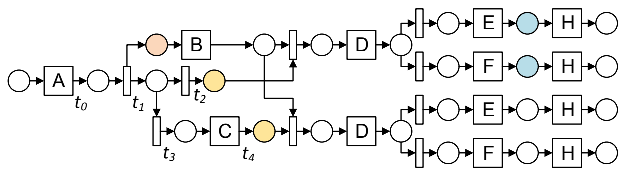


Fig. 5: Branching process of the marked net in Fig. 4

Branching processes are intimately related with prime event structures because they explicitly represent the same set of behavior relations. This fact is formally captured in the following definition.

Definition 3. Let $N = (P, T, F)$ be a net and $x, y \in P \cup T$ two nodes in N.

- x and y are *causal*, written $x <_N y$, iff $(x, y) \in F^+$,
- x and y are in *conflict*, denoted $x \#_N y$, iff $t, t' \in T$: $t \neq t' \wedge \bullet t \cap \bullet t' \neq \emptyset \wedge (t, x), (t', y) \in F^*$,
- x and y are *concurrent*, denoted $x \parallel_N y$, iff neither $x <_N y$, nor $y <_N x$, nor $x \#_N y$.

Given with the above, an event structure can be derived from a marked net by exploiting the fact that the configurations of a branching process can be mapped one-on-one to configurations of a corresponding event structure. Specifically, given a branching process $BP = (P, T, F, \lambda)$ of a marked net N. the event structure $\mathcal{E}$ of N is defined as $\mathcal{E}(N) \triangleq (T, \leq_{BP} \cap (T \times T), \#_{BP} \cap (T \times T), \lambda|_T)$. The latter definition maps both observable and silent transitions to events in the event structure. In [19], the authors proved that silent events can be abstracted away (i.e. removed) in a behavior-preserving manner, under a well-known notion of behavioral equivalence, namely visible-pomset equivalence.[2] The PES presented in Fig. 2(b) is the one that corresponds to the branching process in Fig. 6 after removing all silent events. Accordingly, in the rest of the paper we assume, without loss of generality, that the event structures we manipulate do not have silent transitions.

The branching process of a Petri net with cycles may be infinite. In [17], McMillan showed that for safe nets a prefix of a branching process fully encodes the behavior of the original net. Such prefix of a branching process is referred to as the *complete prefix unfolding* of a net. For example, the complete prefix unfolding corresponding to the marked net in Fig. 4 (and its branching process in Fig. 5) is given in Fig. 6.

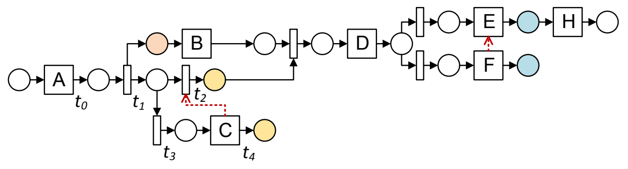


Fig. 6: Complete prefix unfolding of the marked net in Fig. 4

To illustrate the intuition a complete prefix unfolding, let us consider the local configurations $\lceil t_2 \rceil = \{t_0{:}\mathsf{A}, t_1{:}\tau, t_2{:}\tau\}$ and $\lceil t_4 \rceil = \{t_0{:}\mathsf{A}, t_1{:}\tau, t_3{:}\tau, t_4{:}\mathsf{C}\}$. Clearly, the "future" of event t_2, denoted $\lceil t_2 \rceil\Uparrow$, is isomorphic to that of t_4. Indeed, the firing the transitions that correspond with the events in $\lceil t_2 \rceil$ would lead to a marking where places colored orange and yellow in Fig. 4 would hold a token each, which would be the same making that would produce the firing of the transitions that correspond with set of events in $\lceil t_4 \rceil$. Therefore, we can safely stop unfolding the branching process once we reach $t_4{:}\mathsf{C}$ provided that we continue unfolding from $t_1{:}\tau$ and onwards. In this context, the pair (t_4, t_2) is called a *cc-pair*, standing for cutoff-corresponding pair. Moreover, given a cc-pair (e, f) we will denote the isomorphism from $\lceil e \rceil\Uparrow$ to $\lceil f \rceil\Uparrow$ by $\mathcal{I}_{(\lceil e \rceil, \lceil f \rceil)}$. In the graphical representation of unfoldings and PESs (as introduced later), a cc-pair is indicated via a dashed red arrow from the cutoff event to the corresponding event (cf. for example the dashed red arc between t_4 and t_2 in Fig. 6).

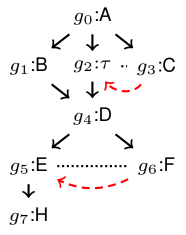


Fig. 7: PES prefix unfolding $\mathcal{E}_2$ of net in Fig. 4

To represent the behavior specified by a BPMN process model, we will use the prime event structure derived from the complete prefix unfolding of the model's Petri net. The latter is herein called the *PES prefix unfolding* of a model. The computation of a PES prefix unfolding is the same as for a regular prime event structure except for the following: (i) we keep track of cc-pairs, and (ii) for convenience, we do not abstract away a silent event when such event is either a cutoff or a corresponding event. For example, Fig. 7 presents the PES prefix unfolding corresponding with the marked net in Fig. 4 and, hence, with the process model in Fig. 2(a).

Reasoning about possible executions of a PES prefix unfolding is not convenient because some configurations are not explicitly represented. To make it more convenient to explore the configurations of a PES prefix unfolding, we adapt to our setting the "shift" operation on net unfoldings

2. This result holds on condition that every sink event in the event structure is a labeled (non-silent) event – something that we can easily ensure by adding (when needed) a "fake" labelled final event to the Petri net from which the event structure is generated.

introduced in [20]. Intuitively, given a cc-pair (e, f), since the futures of $\lceil e \rceil$ and $\lceil f \rceil$ are isomorphic, we can "shift" from one configuration to the other. In other words, the shift operation is a "step" function that allows us to move from one configuration to another. This intuition is captured in the following definition:

Definition 4. Let (e, f) be a cc-pair of the PES prefix $\mathcal{E}_f$ and $\mathcal{I}_{(\lceil e \rceil, \lceil f \rceil)}$ the isomorphism from $\lceil e \rceil \Uparrow$ to $\lceil f \rceil \Uparrow$. Moreover, let C be a configuration of $\mathcal{E}$. The (e, f)-shift of C, denoted $\mathcal{S}_{(e,f)}(C)$, is defined as follows:

$$\mathcal{S}_{(e,f)}(C) = \lceil f \rceil \cup \mathcal{I}_{(\lceil e \rceil, \lceil f \rceil)}(C \setminus \lceil e \rceil)$$

We say that $\mathcal{S}_{(e,f)}(C)$ is a *backward shift* iff $\lceil f \rceil \subset \lceil e \rceil$, that is, the corresponding event f is included in the local configuration of the cutoff event e, otherwise $\mathcal{S}_{(e,f)}(C)$ is called a *forward shift*. Moreover, an event e is said a *backward cutoff event* iff it entails backward shift. Intuitively, a backward shift "moves back" to a configuration that has already be observed in the past of the run.

With abuse of notation, we will use the following variant:

$$\overline{\mathcal{S}}_e(C) = \begin{cases} C & \text{if } e \text{ is not cutoff event} \\ \mathcal{S}_{(e,\mathrm{corr}(e))}(C) & \text{otherwise} \end{cases}$$

Consider for example configuration $C_1 = \{g_0{:}\mathsf{A}, g_1{:}\mathsf{B}, g_3{:}\mathsf{C}\}$ of event structure $\mathcal{E}_2$ in Fig. 7. C_1 contains the cutoff event g_3, which is associated with the cc-pair (g_3, g_2). Given that $\mathcal{S}_{(g_3,g_2)}(C_1) = \{g_0{:}\mathsf{A}, g_1{:}\mathsf{B}, g_2{:}\tau\}$, we infer that $g_2{:}\mathsf{D}$ is a possible extension of C_1.

Esparza [20] shows that any property that holds over a branching process (and thus over a full PES) also holds on its prefix unfolding by applying a sequence of shift operators. In other words, if we wish to compare a full PES with a PES prefix unfolding, we can apply shift operations on the PES prefix in order to materialize behavior that is not explicitly represented. This latter observation is used later when simultaneously traversing a PES prefix derived from a process model and a full PES derived from an event log.

The extraction of a complete prefix unfolding from a Petri net (and the size of the prefix unfolding itself) is exponential on the size of the net [20]. This entails in turn that the derivation of the event structure from an input BPMN model is worst-case exponential.

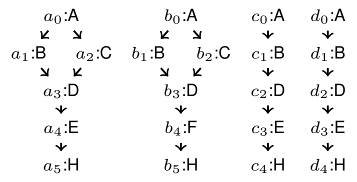
5.2 From log to PES

In previous work [21], we presented a method to generate a PES from an event log. The method consists of two steps. First the event log, seen as a set of traces, is transformed into a set of partially-ordered runs by invoking a *concurrency oracle*, that is a function that given a log returns a set of pairs of event labels that are in a concurrency relation. Each trace is turned into a run by relaxing the total order induced by the trace into a partial order such that two events are not causally related if the concurrency oracle has determined that they occur concurrently. The choice of concurrency oracle is left open. Existing concurrency oracles such as those proposed in the $\alpha+$ process mining algorithm [22] or in [23] can be used for this purpose.

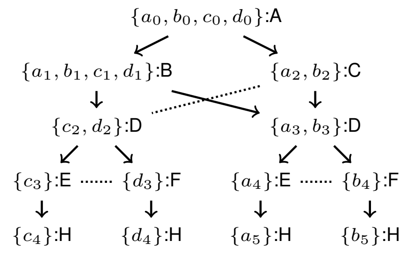
Second, the set of runs are merged into an event structure in a lossless manner, meaning that the set of maximal configurations of the resulting event structure is exactly equal to the set of runs. In this way and modulo the accuracy of the concurrency oracle, we ensure that the resulting event structure is a lossless representation of the input log.

Trace	Ref	N
A B C D E H	t_1	3
A B C D F H	t_2	3
A C B D E H	t_3	2
A C B D F H	t_4	2
A B D E H	t_5	3
A B D F H	t_6	3

(a) Event log



(b) Runs



(c) Induced PES

Fig. 8: Example of construction of a PES from a set of traces

For example, consider the log given in Fig. 8(a). This event log consists of 16 traces, including 3 instances of distinct trace t_1 (as specified in column "N"), 3 instances of t_2, so on and so forth. Using the concurrency oracle of the $\alpha+$ algorithm we conclude that event classes B and C are concurrent, thus we construct the set of runs in Fig. 8(b). In this latter figure, the notation $e{:}\mathsf{A}$ indicates that event e represents an occurrence of event class A in the original log. By merging together events with the same label and the same history (i.e. same prefix), we obtain the PES in Fig. 8(c). In this figure, the notation $\{e_1, e_2 \ldots e_i\}{:}\mathsf{A}$ indicates that events $\{e_1, e_2 \ldots e_i\}$ represent occurrences of event class A in different runs.

For a detailed presentation of the algorithms for transforming traces into runs and merging runs into event structures, the reader is referred to [21]. This latter paper also shows that the complexity of this transformation is $O(|\sigma_m|^3)$, where $|\sigma_m|$ is the length of the longest trace in the event log.

6 Partially Synchronized Product (PSP)

The *Partially Synchronized Product* (PSP) of two event structures [19] is a state machine in which the states correspond to pairs of configurations visited during an error-correcting synchronized traversal of the two input event structures, starting from their empty configurations and ending with all pairs of maximal configurations of the two event structures. A technique for constructing a PSP of two acyclic PESs (without cc-pairs) has been proposed in [19]. In this section, we extend the notion of PSP and the PSP construction technique proposed in [19] in order to handle the case where one of the input event structures is the PES prefix of a process model (and thus contains cc-pairs), and the other is a PES derived from an event log as discussed in Section 5.2.

To illustrate the notion of PSP, consider the pair of event structures shown in Fig. 9. The synchronized product starts with the empty configurations. In this initial state, events a_0 from $\mathcal{E}^a$ and b_0 and b_1 from $\mathcal{E}^b$ are enabled. Since a_0 and b_0 carry the same label (i.e. A), an event match is asserted in the PSP via a so-called "match" operation. This operation leads to a state corresponding to the pair of configurations containing the occurrences of a_0 and b_0. In this state of the PSP, events a_1 from $\mathcal{E}^a$ and b_1 from $\mathcal{E}^b$ are enabled. Although events a_1 and b_1 carry the same label (i.e. B), they cannot be matched because of the discrepancy in the causal relation of the PESs: in $\mathcal{E}^a$ it holds that $a_0 < a_1$ whereas in $\mathcal{E}^b$ it holds that $b_0 \parallel b_1$. In other words, there is an error in the synchronized simulation of the two event structures. To recover from this error, events a_1 and b_1 are declared as "hidden" in the PSP and the synchronized simulation can proceed. This example illustrates two requirements for two events to be matched in the PSP, namely that an event matching must be label preserving (i.e. both events must have the same label) and order-preserving (i.e. event matchings in the PSP are consistent with the causal relation of the input PESs).

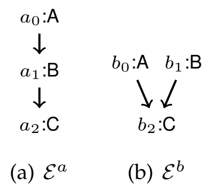


Fig. 9: Sample PES

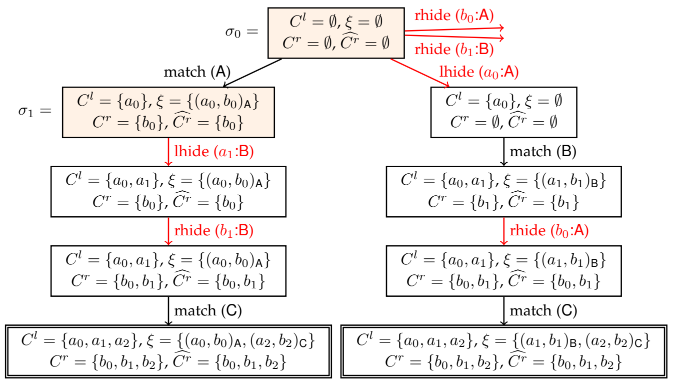

$\sigma_0 =$
$C^l = \emptyset, \xi = \emptyset$
$C^r = \emptyset, \widehat{C^r} = \emptyset$
rhide (b_0:A)
rhide (b_1:B)
match (A)
lhide (a_0:A)
$\sigma_1 =$
$C^l = \{a_0\}, \xi = \{(a_0, b_0)_\mathsf{A}\}$
$C^r = \{b_0\}, \widehat{C^r} = \{b_0\}$
$C^l = \{a_0\}, \xi = \emptyset$
$C^r = \emptyset, \widehat{C^r} = \emptyset$
lhide (a_1:B)
match (B)
$C^l = \{a_0, a_1\}, \xi = \{(a_0, b_0)_\mathsf{A}\}$
$C^r = \{b_0\}, \widehat{C^r} = \{b_0\}$
$C^l = \{a_0, a_1\}, \xi = \{(a_1, b_1)_\mathsf{B}\}$
$C^r = \{b_1\}, \widehat{C^r} = \{b_1\}$
rhide (b_1:B)
rhide (b_0:A)
$C^l = \{a_0, a_1\}, \xi = \{(a_0, b_0)_\mathsf{A}\}$
$C^r = \{b_0, b_1\}, \widehat{C^r} = \{b_0, b_1\}$
$C^l = \{a_0, a_1\}, \xi = \{(a_1, b_1)_\mathsf{B}\}$
$C^r = \{b_0, b_1\}, \widehat{C^r} = \{b_0, b_1\}$
match (C)
match (C)
$C^l = \{a_0, a_1, a_2\}, \xi = \{(a_0, b_0)_\mathsf{A}, (a_2, b_2)_\mathsf{C}\}$
$C^r = \{b_0, b_1, b_2\}, \widehat{C^r} = \{b_0, b_1, b_2\}$
$C^l = \{a_0, a_1, a_2\}, \xi = \{(a_1, b_1)_\mathsf{B}, (a_2, b_2)_\mathsf{C}\}$
$C^r = \{b_0, b_1, b_2\}, \widehat{C^r} = \{b_0, b_1, b_2\}$

Fig. 10: Fragment of the PSP of $\mathcal{E}^a$ and $\mathcal{E}^b$

Fig. 10 presents a fragment of the PSP of event structures $\mathcal{E}^a$ and $\mathcal{E}^b$ shown in Fig. 9. In the general case, the PSP of a pair of event structures is not commutative. Therefore, we fix the following convention: the left-hand side PES is the one that derived from an event log, while the PES derived from a process model is always at the right hand side. Correspondingly, we use "lhide" to denote the hiding of an event from the PES at the left-hand side and "rhide" when the hidden event comes from the other PES.

Note that the fragment of the PSP shown in Fig. 10 has two branches. The leaf state in each brach corresponds to states that can no longer be extended, because they state refers to maximal configurations. Informally, the left-hand side branch states that, without considering the occurrence of events a_1 and b_1, in both PESs we observe the execution of a task A followed by the execution C. If we consider the other branch, we would have a somehow symmetric conclusion. We note also that we could also find in the PSP a sequence of PSP operations hiding all the events from both event structures. However, such sequence would be not be informative. Instead, we are interested only in sequences of PSP operations that maximize the number of event matches or, symmetrically minimize the number of hide operations, which we will call optimal event matchings.

From a conceptual point of view, a PSP is a directed acyclic graph. A node in a PSP represents a state in the synchronized simulation. An arc in a PSP, on the other hand, represents a transition between states in the simulation and is labeled by the type of operation that was used in the transition. Formally, a PSP is denoted as a tuple (Σ, OP, A), where Σ is a set of states; OP is the set of operations in a PSP, i.e. $OP \triangleq \{\text{match}, \text{lhide}, \text{rhide}\}$; and $A \subseteq \Sigma \times OP \times \Sigma$ is the set of directed arcs.A state in the PSP stores the configurations of the input event structures, as a way to keep track of the moves in the synchronized simulation. Given a pair of event structures $\mathcal{E}^l$ and $\mathcal{E}^r$, a state $\sigma \in \Sigma$ of a PSP is defined as a tuple $(C^l, \xi, C^r, \widehat{C^r})$. There, C^l and $\widehat{C^r}$ represent configurations from $\mathcal{F}(\mathcal{E}^l)$ and $\mathcal{F}(\mathcal{E}^r)$, respectively. Since $\mathcal{E}^r$ is a PES prefix, $\widehat{C^r}$ may be the result of a shift operation. In the tuple, C^r is a multiset of events from $\mathcal{E}^r$, which records not only the set of events but also the number of event occurrences during the synchronized simulation. Finally, $\xi \subseteq E(\mathcal{E}^l) \times E(\mathcal{E}^r)$ holds a set of event pairs, representing the event matches that have been observed in a path from the root state of the PSP to state σ.

We approached the problem of computing the set of optimal event matchings in the PSP with Algorithm 1, which is based on the well-known A* heuristic search [24].The problem at hand can be naturally mapped into a multi-objective heuristic search. However, as for other domains, the memory requirements of a multi-objective approach are high. As a result, Algorithm 1 is designed as a single-objective heuristic search, which matches one maximal configuration from the event log at a time. The result of this algorithm is later combined to produce the entire PSP. With abuse of notation, we use $C \xrightarrow{e} C'$ to denote the extension of the configuration C with event e, such that $C' = C \oplus e$.

Given two event structures $\mathcal{E}^l$ and $\mathcal{E}^r$, and a maximal configuration C^l_m of $\mathcal{E}^l$, Algorithm 1 proceeds as follows. It starts by considering the root state (all configurations are set to empty set) in line 2 and enters a while loop in line 4, which will be repeated as long as there is an unprocessed state in OPEN. In line 5, a state σ is taken from OPEN such that σ has minimum cost φ. The cost function φ will be discussed later. In lines 8-12, the algorithm identifies the set of event matches. To this end, given the configurations C^l and $\widehat{C^r}$ in the state σ, we iterate over the set of possible extensions for both configurations. When a pair of events is found to be label-preserving and order-preserving, a new state σ' is instantiated and an arc $(\sigma, \text{match}, \sigma')$ is added to the PSP in line 10. Label preservation is straightforwardly checked by comparing the event labels, i.e. $\lambda^l(e) = \lambda^r(f)$. Order preservation, on the other hand, is checked by calling the function FINDCAUSALINC: this function returns $\perp$ a pair of events is found order-preserving in state σ. The function FINDCAUSALINC will be further discussed later in this Section. Then, the new state σ' is added to OPEN. Note that the configuration $\widehat{C^r}'$ is updated accordingly (i.e. it is shifted) when the event f is a cutoff event. Then, a

Algorithm 1 Partially synchronised product

```
1:  function BUILDPSP($\mathcal{E}^l$, $\mathcal{E}^r$, $C^l_{\mathrm{m}}$)
2:      OPEN $\leftarrow \{(\emptyset, \emptyset, \emptyset, \emptyset)\}$
3:      Initialize PSP
4:      while OPEN $\neq \emptyset$ do
5:          Choose $\sigma = (C^l, \xi, C^r, \widehat{C^r}) \in$ OPEN, with min. $\varphi(\sigma, C^l_{\mathrm{m}})$
6:          OPEN $\leftarrow$ OPEN $\setminus \{\sigma\}$
7:          return (PSP, $\sigma$) if $C^l \in \mathcal{F}_{\mathrm{m}}(\mathcal{E}^l) \wedge \widehat{C^r} \in \mathcal{F}_{\mathrm{m}}(\mathcal{E}^r)$
8:          for each $\left( \begin{array}{l} C^l \xrightarrow{e} C^{l\prime}, \widehat{C^r} \xrightarrow{f} \widehat{C^{r\prime}}, \text{s.t.} \\ e \in C^l_m \wedge \lambda^l(e) = \lambda^r(f) \wedge \\ \text{FINDCAUSALINC}(\text{PSP}, \sigma, e, f) = \perp \end{array} \right)$ do
9:              $\sigma' \leftarrow (C^{l\prime}, \xi \cup \{(e, f)\}, C^r \cup \{f\}, \overline{\mathcal{S}}_f(\widehat{C^{r\prime}}))$
10:             ADDARC(PSP, ($\sigma$, "match", $\sigma'$))
11:             PUSH(OPEN, $\sigma'$)
12:         end for
13:         for each $C^l \xrightarrow{e} C^{l\prime}$, s.t. $e \in C^l_m$ do
14:             $\sigma' \leftarrow (C^{l\prime}, \xi, C^r, \widehat{C^r})$
15:             ADDARC(PSP, ($\sigma$, "lhide", $\sigma'$))
16:             PUSH(OPEN, $\sigma'$)
17:         end for
18:         for each $\widehat{C^r} \xrightarrow{f} \widehat{C^{r\prime}}$ do
19:             $\sigma' \leftarrow (C^l, \xi, C^r \cup \{f\}, \overline{\mathcal{S}}_f(\widehat{C^{r\prime}}))$
20:             ADDARC(PSP, ($\sigma$, "rhide", $\sigma'$))
21:             PUSH(OPEN, $\sigma'$)
22:         end for
23:     end while
24: end function
```

hide operation is processed, i.e. a new state and an arc are added to the PSP, for each possible extension found in the configuration C^l (lines 13-17) and C^r (lines 18-22). Note that we restrict the processing of events from $\mathcal{E}^l$ to those that are part of C^l_m. This is done by checking $f \in C^l_{\mathrm{m}}$ in lines 8 and 13. In this way, the search is explicitly directed to find an optimal matching for C^l_{m}. Moreover, the algorithm will stop in line 7 when it first reaches a state where C^l_{m} has been matched.

We now turn our attention to the problem of checking if a candidate event match is order-preserving or not. One of the key difficulties in this context is to properly consider the shift operations that have occurred in the path that leads to a given state in the PSP. Algorithm 2 provides a solution to this problem.

Algorithm 2 Find causally inconsistent events in the PSP w.r.t. a given pair of events

```
1:  function FINDCAUSALINC(PSP, $\sigma$, $e$, $f$)
2:      cutoffs $\leftarrow \emptyset$
3:      while $\exists(\sigma_{\text{pred}},\ op,\ \sigma) \in A(\text{PSP})$ do
4:          $(e', f') \leftarrow$ GETDELTAEVENTS($\sigma$, $\sigma_{\text{pred}}$)
5:          if $f'$ is cutoff event then
6:              cutoffs $\leftarrow$ cutoff $\circ f'$
7:              $f \leftarrow \mathcal{I}_{f'}^{-1}[f]$
8:          end if
9:          if $op$ is "match" then
10:             return $(e, e', f, f', \text{cutoffs})$ if $\neg(e' < e \Leftrightarrow f' < f)$
11:         end if
12:         $\sigma \leftarrow \sigma_{\text{pred}}$
13:     end while
14:     return $\perp$
15: end function
```

The algorithm backward-traverses the PSP from a given state to the root state, checking if the input pair of events are causally consistent with the matched events as recorded in a path of the PSP. If a pair of events is found that is causally inconsistent, the algorithm returns a tuple containing the input events (possibly updated to compensate the effect of shift operations), the causally inconsistent events, and the sequence of cutoff events that are traversed during the search. This algorithm is used later in the characterization of behavior mismatches as explained in Section 7.

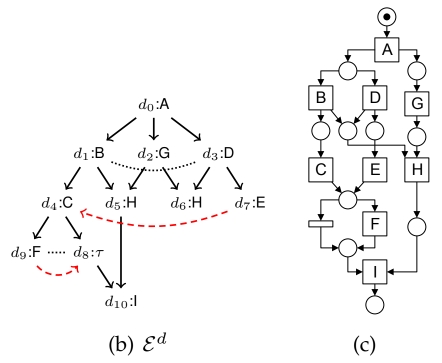


Fig. 11: Sample PESs ($\mathcal{E}^d$ is the PES prefix of (c))

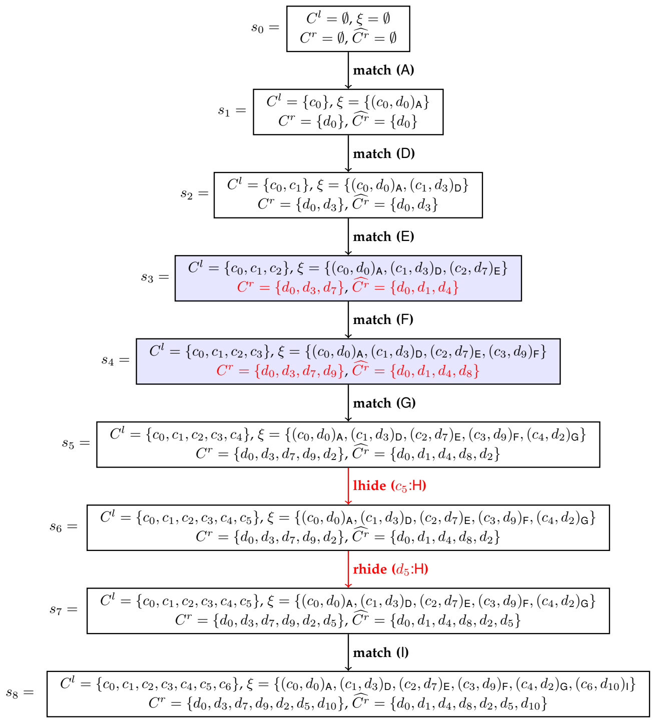

$s_0 =$ $C^l = \emptyset, \xi = \emptyset$ $C^r = \emptyset, \widehat{C^r} = \emptyset$
match (A)
$s_1 =$ $C^l = \{c_0\}, \xi = \{(c_0, d_0)_{\mathsf{A}}\}$ $C^r = \{d_0\}, \widehat{C^r} = \{d_0\}$
match (D)
$s_2 =$ $C^l = \{c_0, c_1\}, \xi = \{(c_0, d_0)_{\mathsf{A}}, (c_1, d_3)_{\mathsf{D}}\}$ $C^r = \{d_0, d_3\}, \widehat{C^r} = \{d_0, d_3\}$
match (E)
$s_3 =$ $C^l = \{c_0, c_1, c_2\}, \xi = \{(c_0, d_0)_{\mathsf{A}}, (c_1, d_3)_{\mathsf{D}}, (c_2, d_7)_{\mathsf{E}}\}$ $C^r = \{d_0, d_3, d_7\}, \widehat{C^r} = \{d_0, d_1, d_4\}$
match (F)
$s_4 =$ $C^l = \{c_0, c_1, c_2, c_3\}, \xi = \{(c_0, d_0)_{\mathsf{A}}, (c_1, d_3)_{\mathsf{D}}, (c_2, d_7)_{\mathsf{E}}, (c_3, d_9)_{\mathsf{F}}\}$ $C^r = \{d_0, d_3, d_7, d_9\}, \widehat{C^r} = \{d_0, d_1, d_4, d_8\}$
match (G)
$s_5 =$ $C^l = \{c_0, c_1, c_2, c_3, c_4\}, \xi = \{(c_0, d_0)_{\mathsf{A}}, (c_1, d_3)_{\mathsf{D}}, (c_2, d_7)_{\mathsf{E}}, (c_3, d_9)_{\mathsf{F}}, (c_4, d_2)_{\mathsf{G}}\}$ $C^r = \{d_0, d_3, d_7, d_9, d_2\}, \widehat{C^r} = \{d_0, d_1, d_4, d_8, d_2\}$
lhide (c_5:H)
$s_6 =$ $C^l = \{c_0, c_1, c_2, c_3, c_4, c_5\}, \xi = \{(c_0, d_0)_{\mathsf{A}}, (c_1, d_3)_{\mathsf{D}}, (c_2, d_7)_{\mathsf{E}}, (c_3, d_9)_{\mathsf{F}}, (c_4, d_2)_{\mathsf{G}}\}$ $C^r = \{d_0, d_3, d_7, d_9, d_2\}, \widehat{C^r} = \{d_0, d_1, d_4, d_8, d_2\}$
rhide (d_5:H)
$s_7 =$ $C^l = \{c_0, c_1, c_2, c_3, c_4, c_5\}, \xi = \{(c_0, d_0)_{\mathsf{A}}, (c_1, d_3)_{\mathsf{D}}, (c_2, d_7)_{\mathsf{E}}, (c_3, d_9)_{\mathsf{F}}, (c_4, d_2)_{\mathsf{G}}\}$ $C^r = \{d_0, d_3, d_7, d_9, d_2, d_5\}, \widehat{C^r} = \{d_0, d_1, d_4, d_8, d_2, d_5\}$
match (I)
$s_8 =$ $C^l = \{c_0, c_1, c_2, c_3, c_4, c_5, c_6\}, \xi = \{(c_0, d_0)_{\mathsf{A}}, (c_1, d_3)_{\mathsf{D}}, (c_2, d_7)_{\mathsf{E}}, (c_3, d_9)_{\mathsf{F}}, (c_4, d_2)_{\mathsf{G}}, (c_6, d_{10})_{\mathsf{I}}\}$ $C^r = \{d_0, d_3, d_7, d_9, d_2, d_5, d_{10}\}, \widehat{C^r} = \{d_0, d_1, d_4, d_8, d_2, d_5, d_{10}\}$

Fig. 12: Excerpt of the PSP of $\mathcal{E}^c$ and $\mathcal{E}^d$

To illustrate Algorithm 2, we will use the pair of PESs in Fig. 11. Here, $\mathcal{E}^d$ is the PES prefix of the Petri net shown in Fig. 11(c). Assuming that the PSP, shown in Fig. 12, has been computed up to state s_2, events c_2 of $\mathcal{E}^c$ and d_7 of $\mathcal{E}^d$ are enabled. As a result, function FINDCAUSALINC is called by Algorithm 1 (line 8). The input PSP in this call refers to the excerpt of the PSP in Fig. 11 up to state s_3, σ refers to state s_3, e refers to c_2 and f to d_7. In line 2, variable cutoffs –

which stores the sequence of cutoff events found during the traversal – is initialized with an empty sequence. The while loop in lines 3-13 traverses the PSP backwards, processing one arc from the input PSP at a time. In the first iteration, the algorithm analyzes arc $(s_1, \text{"match"}, s_2)$. Next, in line 4, function GETDELTAEVENTS is called to determine the set of events involved in the operation associated with arc $(s_2, \text{"match"}, s_3)$. This function GETDELTAEVENTS can be straightforwardly implemented by computing the difference of C^l and C^r in the states s_3 and s_2. In this case, the function returns and set $e' = c_2$ and $f' = d_7$, respectively. Since d_7 is a cutoff, this event is appended to cutoff in line 6. Note that variable f remains unchanged in line 7 (i.e. $\mathcal{I}_{d_2}^{-1}[d_2] = d_2$). Then, the block in lines 9-11 is executed because op is a "match" operation. Let us consider the expression in line 10, i.e. $\neg(e' < e \Leftrightarrow f' < f)$, which in this case is false, because it holds that $c_1 < c_2$ and $d_3 < d_7$. This condition ensures the consistency of the causal relations of input events (e.g. a pair of events that form candidate event match to be added to the PSP) with all the event matches recorded in the PSP that precede the state that activated the input events. In addition to validating the requirement of order preservation, as explained before, this function also returns the pair of events for which causal consistency does not hold if any pair is found. The first iteration concludes by updating σ with the value s_1 in line 12. In the second iteration, the algorithm processes arc $(s_0, \text{"match"}, s_1)$ in a similar way as in the first iteration. Since $c_0 < c_2$ and $d_0 < d_7$ hold true, order preservation is also decided true. This is the last iteration of the loop and the function returns $\perp$, indicating the absence of causal inconsistencies.

Let us now assume that the PSP has been computed up to state s_5 and function FINDCAUSALINC is called with events $e = c_5$ and $f = d_5$. In this case, this function processes the arcs associated with operations "match (G)" and "match (F)" in a similar way as described before. We further analyze the iteration where FINDCAUSALINC processes the arc associated with operation "match (E)". In this iteration, GETDELTAEVENTS returns and set $e' = c_2$ and $f' = d_7$, respectively. In line 6, event d_7 is appended to cutoff that, at this point, is set to $[d_9, d_7]$. Next, variable f is updated from d_5 to d_6 in line 7 (i.e. $\mathcal{I}_{d_2}^{-1}[d_5] = d_6$). It is because of this update that, in the following iteration of the while loop, the function finds that $c_1 \parallel c_5$ and $d_3 < d_6$, and hence concludes that the matching of c_5 and d_5 is not order-preserving.

We now turn our attention to defining the cost function φ, used in Algorithm 1. As for any conventional A*-based algorithm, the cost function is used as a criterion for selecting the next state to expand while constructing the PSP. We define function φ as follows.

Definition 5. Let $\sigma = (C^l, \xi, C^r, \widehat{C^r})$ be a state in PSP and $C_\mathrm{m} \in \mathcal{F}_\mathrm{m}(\mathcal{E}^l)$ be a maximal configuration of $\mathcal{E}^l$ (the PES of the event log), such that $C^l \subseteq C_\mathrm{m}$. The function φ is defined as

$$\varphi(\sigma, C_\mathrm{m}) = g(\sigma) + h(\sigma, C_\mathrm{m})$$

where

$$g(\sigma) = |C^l| + |C^r|_{\lambda^r}| - |\xi| * 2$$

and

$$h(\sigma, C_\mathrm{m}) = |\lambda(C_\mathrm{m} \setminus C^l) \setminus \lambda(C^r \Uparrow)|$$

Intuitively, the cost function g corresponds to the number of hide operations incurred in the path starting from the root state in the PSP and leading to a given state. Clearly, we would like to find a sequence with only match operations, if such a sequence exists, or the path that includes the minimum number of hide operations. Note that in the case of C^r, we restrict our attention to the set of observable events, denoted $C^r|_\lambda$. In fact, we are interested in characterizing differences in terms of visible events, but we have to keep some of the invisible events in the PES prefix to maintain the information about cc-pairs in the complete prefix unfolding. Thus, in our definition of g we search to not penalizing operations that involve invisible events recorded in C^r.

As per the conventional A*-based algorithm, function h corresponds to an optimistic approximation to the "future cost": the cost of a path starting from a given state up to a goal state. In our context, the goal state corresponds to the optimal state in which a maximal configuration C_m (coming from the PES of the event log) is matched. To this end, we consider the possible futures for both configurations. In the case of C^l, we consider only the set of events in $C_\mathrm{m} \setminus C^l$, because Algorithm 1 looks a finding an optimal matching for C_m. In the case of C^r, we consider the set of events $C^r \Uparrow$. Intuitively, function h provides a costs that corresponds to the estimated number of events that need to be hidden from C^l to match both configurations. Note that such estimation is optimistic because it does not consider the events to be hidden from the other configuration.

Since h is an optimistic cost function, it follows that Algorithm 1 is admissible [24] and henceforth returns an optimal solution.

When the input PESs have concurrent behavior, the PSP may contain paths with redundant information. This redundancy stems from the interleaved enablement of concurrent events. Fig. 13 gives an example of this situation.

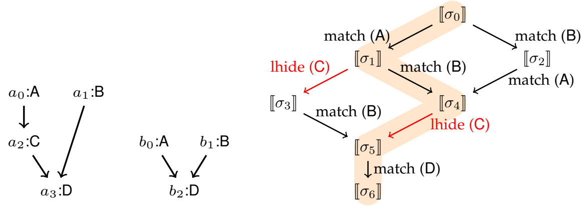


Fig. 13: PSP with redundant information due to concurrency

From the figure above, it can be easily checked that the PSP contains three different paths, all capturing the same information. This is a well known problem in areas such as Model Checking and others, where techniques have developed to discard some paths when exploring the underlying state space [25]. Therefore, we can leverage results from that field to reduce redundant information at the time of the construction of the PSP. In that context, it is well known that the exploration of the state space can be reduced by analyzing the commutativity of the transitions in the state space, which translates to our setting to the notion of commutativity of operations. Intuitively, we say that a pair of operations can be commuted if their underlying events are enabled concurrently. For instance, the event a_0 and a_1 are concurrently enabled by the empty configuration and so are the events b_0 and b_1 in the example on Fig. 13. As shown in the PSP, the operations "match (A)" and "match (B)" appear in two distinct orders, in paths that start in state σ_0 and finishing in σ_4. We will say that operations "match (A)" and "match (B)" are commutative. A similar situation happens at state σ_1 with operations "lhide (C)" and "match (B)", because the events a_1 and a_2 are concurrents. The intuition above is captured in the following function:

Algorithm 3 Commutativity of operations

```
function AreComm($e, e', f, f'$)
    assume AreConcOrUndef($a, b$) $\triangleq (a \parallel b \vee a = \perp \vee b = \perp)$
    return AreConcOrUndef($e, e'$) $\wedge$ AreConcOrUndef($e, e'$)
end function
```

Commutativity of operations is closely inspired in the notion of commutativity of transitions in model checking. This property is at the heart of a large number of partial order reduction techniques used in model checking [25]. What is more, we can recover their theoretical results to claim that one path encodes the same information as the other paths, reducing significantly the size of PSP. In this respect, Algorithm 4 presents the modifications to apply on Algorithm 1 to achieve the partial order reductions.

Algorithm 4 Partially synchronized product with partial order reductions

▷ Replace lines 8-12 with the following block

```
1: $\mathbb{E} \leftarrow \emptyset$
   ▷ Consider configuration extensions according to the lexicographical order of the event labels
2: for each ( $C^l \xrightarrow{e} C^{l\prime}, \widehat{C^r} \xrightarrow{f} \widehat{C^{r\prime}}$, s.t.
              $e \in C^l_m \wedge \lambda^l(e) = \lambda^r(f) \wedge$
              FindCausalInc(PSP, $\sigma, e, f$) $= \perp$ ) do
3:     if $\exists (e', f') \in \mathbb{E}$, s.t. $e \parallel e' \vee f \parallel f'$ then
4:         continue
5:     else
6:         $\mathbb{E} \leftarrow \mathbb{E} \cup \{(e, f)\}$
7:     end if
8:         ▷ Keep lines 9-11 as they are
9: end for
```

▷ Replace line 13 with the following one

```
for each $C^l \xrightarrow{e} C^{l\prime}$, s.t. $e \in C^l_m \wedge \nexists (e', f') \in \mathbb{E} : e \parallel e'$ do
```

▷ Replace line 18 with the following one

```
for each $\widehat{C^r} \xrightarrow{f} \widehat{C^{r\prime}}$, s.t. $\nexists (e', f') \in \mathbb{E} : f \parallel f'$ do
```

The modified Algorithm above would build a reduced PSP for the example in Fig. 13 as follows. When the algorithm analyzes the root state in the PSP, that is σ_0 in Fig. 13, it would have to process the operations "match (A)" and "match (B)". Note that we assume that configuration extensions are ordered according to the event labels. Therefore, "match (A)" will be appended first to the PSP and the variable $\mathbb{E}$, standing for *enablements*, will be set to $\{(a_0, b_0)\}$. In the next iteration of the for loop, the algorithm will not longer consider the operation "match (B)" as a successor of σ_0, as if the events a_1 and b_1 were not enabled in such a state. For the same reason, the hide operations associated with events a_1 and b_1 will not be appended to σ_0 either. In the next iteration, the algorithm will consider the operations "match (B)", which was warrantied because of the commutability of the operations "match (A)" and "match (B)". Once the operation"match (B)" is appended, the algorithm proceeds as usual (following the path highlighted in Fig. 13), because all the remaining operations are not commutable.

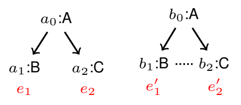


Fig. 14: Example of PESs

Note in Algorithm 4, the testing of commutativity has been slightly modified. It is by means of the expression in line 3, namely $e_1 \parallel e'_1 \vee e_2 \parallel e'_2$, that the algorithm checks commutativity of operations. In fact, line 3 will discard any match operation that is found commutable with one that has been previously appended during a previous iteration of the for loop. Let us further analyze the condition in line 3. First, if both terms in $e_1 \parallel e'_1 \vee e_2 \parallel e'_2$ hold true, then it is because we have two potential match operations that are clearly commutative, in line with Algorithm 3. Fig. 14 presents an example that illustrates the second case. Assume that events a_1 and b_1 have already been processed (i.e. $(a_1, b_1) \in \mathbb{E}$). When we want to process the match operation associated with the events a_2 and b_2, we will have that the first term in the expression $e_1 \parallel e'_1 \vee e_2 \parallel e'_2$ hold true whereas the second term does not, because $b_1 \# b_2$. Fig. 14 shows e_1, e_2, etc. in red font to ease the mapping of the example to the expression. Due to the presence of conflict, the operation "match (C)" will not be appended to the PSP in the path that follows the operation "match (B)" and, as a result, we will see a "lhide (a_1)" and this operation is commutable with "match (B)" in the sense of Algorithm 3. The remaining case, that is when the first term in $e_1 \parallel e'_1 \vee e_2 \parallel e'_2$ is false and the second term is true is the symmetric of the second case.

To analyze the complexity of the PSP computation we analyze the size of the state space explored by the A* search. This state space is in $O(3^{|\mathcal{F}(E_1)| \cdot |\mathcal{F}(E_2)|})$ where $\mathcal{F}(E)$ is the set of configurations of E. Indeed, each configuration in E_1 is associated with a configuration from E_2 via three possible operations (i.e. $match$, $lhide$ and $rhide$). This worst case complexity may be reached when the event structures are completely different. Conversely, when the event structures are identical, the heuristic search converges in linear time. When checking the conformance of a process model with a corresponding event log, we expect a high overlap in behavior and hence a complexity far below the worst case.

7 Difference Extraction and Verbalization

In the previous section, we showed that a PSP contains a minimal set of hide operations required to capture all behavioral discrepancies between the PES of an event log and the PES of a process model. In this section, we show how to traverse the PSP in order to extract a set of difference statements that characterize the behavior observed in the log and not captured in the model and vice-versa.

In order to extract such difference statements, we rely on a collection of nine *mismatch patterns* classified into the following disjoint categories:

- *Unfitting behavior patterns*, capturing behavior observed in the log but not allowed by the model. Unfitting behavior patterns are further classified into two subcategories:
  - *Relation mismatch patterns*, capturing cases where two events in the PES of the log are related via a given behavioral relation (immediate causality, direct conflict or concurrency) but they are related via a different relation in the PES prefix of the model – e.g. they are related via immediate causality in the log while they are related via direct conflict in the PES of the model.
  - *Event mismatch patterns*, capturing all other cases of unfitting behavior. These patterns capture events in the PES of the log that cannot be directly matched to an event in the PES prefix of the model.
- *Additional behavior patterns*, covering all cases where behavior is allowed in the process model but not observed in the log.

In the following, we describe each pattern in turn.

7.1 Relation mistmatch patterns

The first category of mismatch patterns corresponds to situations where a pair of events – one from the log PES and one from the model PES prefix – have the same label but they are not matched in the PSP because they have different behavioral relations with at least one other event. In other words, there is a pair of events in the PES of the log linked via a given relation, which could be matched to a corresponding pair of events in the PES prefix of the model, if it was not for the fact that these pairs are related via different behavioral relations. Since there are only three behavior relations, there are also three possible (symmetric) relation mismatches: *Immediate Causality vs. Concurrency*, *Immediate Causality vs. Direct Conflict*, and *Concurrency vs. Direct Conflict*.[3] As we will see later, the last two types of mismatches (those involving conflict) have very similar manifestations in the PSP and hence we will treat them as one single pattern. Hence below we introduce two patterns, namely *Causality-Concurrency* and *Conflict*.

Like all other patterns introduced later, relation mismatch patterns occurs in a given *context*, characterized by a pair of configurations (one configuration from each PES). For instance, consider the example shown in Fig. 15. We note that $a_1 \parallel a_2$ whereas $b_1 < b_2$ and these two pairs of events have matching labels. Thus, there is a Causality-Concurrency mismatch. This mismatch is observed in the state of the PSP associated with the pair of matching configurations $\{a_0, a_1\}$ and $\{b_0, b_1\}$ (the mismatch context). We also note that one of the events that is hidden in the PSP (μb_2) is the target of an immediate causality relation stemming from an event in the configuration $\{b_0, b_1\}$ (specifically note that $b_1 <_\mu b_2$)

We also observe that a relation mismatch pattern manifests itself in the PSP in the form of two "hide" operations. In the example shown in Fig. 15, the pair of "hide" operations are contiguous in one path of the PSP. However, the hide operations do not necessarily happen always contiguously in

3. We only report mismatches involving immediate causality (not transitive causality) and direct conflict (not transitive conflict), because we are only interested in reporting each mismatch once.

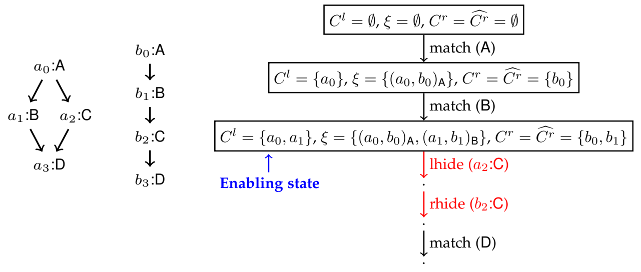

a_0:A
a_1:B
a_2:C
a_3:D
b_0:A
b_1:B
b_2:C
b_3:D
$C^l = \emptyset, \xi = \emptyset, C^r = \widehat{C^r} = \emptyset$
match (A)
$C^l = \{a_0\}, \xi = \{(a_0, b_0)_A\}, C^r = \widehat{C^r} = \{b_0\}$
match (B)
$C^l = \{a_0, a_1\}, \xi = \{(a_0, b_0)_A, (a_1, b_1)_B\}, C^r = \widehat{C^r} = \{b_0, b_1\}$
Enabling state
lhide (a_2:C)
rhide (b_2:C)
match (D)

Fig. 15: Causality-Concurrency mismatch

the PSP as shown in the example of Fig. 16. The reason why the "hide" operations are not contiguous in the PSP of this second example stems from the fact that "match (C)" and "lhide (a_1:B)" are commutative. Hence, the identification of this type of mismatches requires one to take into account the commutativity of operations.

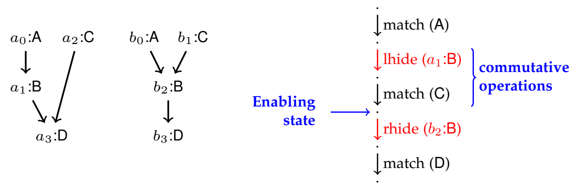


Fig. 16: Another Causality-Concurrency mismatch

The proposed approach to identify Causality-Concurrency relation mismatches (later referred to as CAUSCONC mismatches) is formalized in Algorithm 5. This algorithm analyses one path from the PSP at a time (line 2). In line 3, it selects three arcs each corresponding to an "lhide", an "rhide" and a "match" operation respectively. Then, in lines 4-6 it determines the set of events that are involved in the three operations. In line 7, it maps cutoff with its corresponding event, if required. Note that in lines 8-10, the algorithm discard the operations being analysed (i.e. the loop is forced to proceed with the next iteration in line 9), if the same pattern can be built with another hide operation that is causally predecessor of one of the hide operations being processed. This situation is checked by function CHECKPREDS. Finally, in line 11 the algorithm checks the conditions defining an elementary causality-concurrency mismatch: the pair of hidden events carry the same label and one pair of events are in immediate causal relation whereas the other pair is concurrent.

Let us now turn our attention to the two other cases, namely *Causality-Conflict* and *Concurrency-Conflict*. Again, we observe that these mismatches show up in the PSP in the form of two "hide" operations, but these two "hide" operations appear in different branches. The latter holds because because configurations are conflict-free (i.e. two conflicting events cannot occur in the same computation).

The example shown in Fig. 17 corresponds to a case of *Concurrency-Conflict* mismatch. There, it can be seen that the hide operations occur in different branches. Due to the presence of concurrency, the hide operation and the conflicting match operation can appear in either order. Fig. 18 presents a more complex situation. In this example, a *Causality-*

Algorithm 5 Finding Causality-Concurrency mismatches

```
1: procedure FINDCAUSALCONCMISMATCHES(PSP)
2:     for each PATH in PSP do
3:         for each $(\sigma_1, \text{lhide}, \sigma'_1), (\sigma_2, \text{rhide}, \sigma'_2), (\sigma_3, \text{match}, \sigma'_3) \in$ PATH do
4:             $(\perp, f) \leftarrow$ GETDELTAEVENTS$(\sigma'_1, \sigma_1)$
5:             $(e, \perp) \leftarrow$ GETDELTAEVENTS$(\sigma'_2, \sigma_2)$
6:             $(e', \_f') \leftarrow$ GETDELTAEVENTS$(\sigma'_3, \sigma_3)$
7:             $f' \leftarrow$ ISCUTOFF$(\_f')$ ? corr$(\_f')$ : $\_f'$
8:             if ( CHECKPREDS(PSP, $\sigma'_3, \sigma_1$, "lhide", $e, \lambda_l(e)$) ∨
                    CHECKPREDS(PSP, $\sigma'_2, \sigma_1$, "rhide", $f, \lambda_r(f)$) ) then
9:                 continue
10:            end if
11:            if $\lambda_l(e) = \lambda_r(f) \wedge (e' <_\mu e \vee f' <_\mu f) \wedge (e' \parallel e \vee f' \parallel f)$ then
12:                assert(CAUSCONC$(\sigma_3, e, f, e', \_f', f')$)
13:            end if
14:        end for
15:    end for
16: end procedure
17: function CHECKPREDS(PSP, $\sigma_0, \sigma_2$, op, $e$, $label$)
18:     ▷ This function returns $true$ if there does not exist an arc $(\sigma, (op), \sigma') \in A(\text{PSP})$ that involves an event that carries the label $label$ and that causally precedes event $e$, and $false$ otherwise
19: end function
```

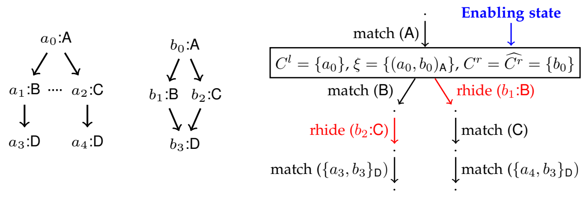


Fig. 17: Concurrency/conflict mismatch

Conflict mismatch is intertwined with a pair of concurrent events. This leads to the hide operation and the conflicting match operation being separated. Therefore, an approach to identify these mismatches must take into account the commutativity of operations, in the same way as for the *Causality-Concurrency* mismatch pattern.

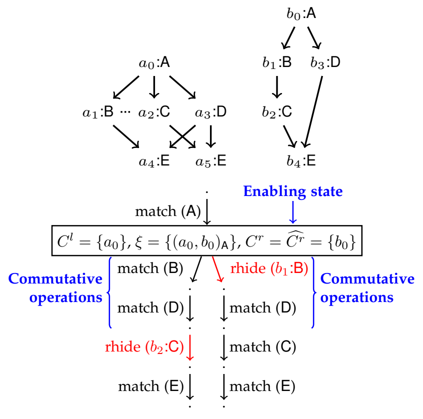


Fig. 18: Causality/Conflict mismatch

The above observations on the characteristics of mismatch patterns involving conflict are formalized in Algorithm 6, which can identify both *Causality-Conflict* and *Concurrency-Conflict* mismatches (herein referred to as CONFLICT mismatches). Algorithm 6 is similar way to Algorithm 5, except for two key points. First, Algorithm 6 processes three arcs at a time, but the arcs are not required to come from the same path. Recall that the hide operations associated to a Causality-Conflict or a Concurrency-Conflict mismatch pattern will be located in two different branches. Second, the conditions in line 10 are the ones that define a conflict-related mismatch: the hidden events carry the same label and one pair of events are in immediate conflict relation whereas the other pair are in either causal or in concurrency relation. Note that the symmetric condition holds, i.e. $\neg(e' \# e) \wedge f' \# f$).

Algorithm 6 Finding Conflict mismatches

```
1: procedure FINDCAUSALCONCMISMATCHES(PSP)
2:     for each $(\sigma_1, \text{lhide}, \sigma'_1), (\sigma_2, \text{rhide}, \sigma'_2), (\sigma_3, \text{match}, \sigma'_3) \in A(\text{PSP})$ do
3:         $(\perp, f) \leftarrow$ GETDELTAEVENTS$(\sigma'_1, \sigma_1)$
4:         $(e, \perp) \leftarrow$ GETDELTAEVENTS$(\sigma'_2, \sigma_2)$
5:         $(e', \_f') \leftarrow$ GETDELTAEVENTS$(\sigma'_3, \sigma_3)$
6:         $f' \leftarrow$ ISCUTOFF$(\_f')$ ? corr$(\_f')$ : $\_f'$
7:         if ( CHECKPREDS$(\sigma'_3, \sigma_1$, "lhide", $\lambda_l(e)$) ∨
                CHECKPREDS$(\sigma'_2, \sigma_1$, "rhide", $\lambda_r(f)$) ) then
8:             continue
9:         end if
10:        if $\lambda_l(e) = \lambda_r(f) \wedge (e' \#_\mu e \vee f' \#_\mu f) \wedge \neg(e' \# e \wedge f' \# f)$ then
11:            assert(CONFLICT$(\sigma_3, e, f, e', \_f', f')$)
12:        end if
13:    end for
14: end procedure
```

In the way they are formulated, Algorithms 5 and 6 are O(n^3), where n is the number of arcs in the PSP. However, with some technical optimizations, the identification of all relation mismatch patterns can be implemented using a single depth-first search traversal of the PSP – hence with an O(n) complexity. Details of how the two algorithms can be combined into a single depth-first search traversal are omitted as they are purely technical optimizations.

7.2 Event mismatch patterns

In this category of patterns, we group together all cases of unfitting behavior that cannot be characterized via a relation mismatch. A naive way of characterizing such cases would be to simply state that there are some events in the PES of the log that are not matched to any event in the PES prefix of the model. However, such an approach to diagnose differences is too low level and would lead to a high number of difference statements. Instead, we introduce four mismatch patterns that capture possible reasons for the presence of an unmatched event at a higher level of abstraction, namely *task skipping*, *unmatched repetition*, *task substitution* and *task relocation*. When a given unfitting behavior cannot be characterized using any of these four patterns, we use a fifth "catch all" pattern (namely *Task absence*), which essentially states that there is an event can occur in the PES of the log in a given configuration but not in the corresponding configuration in the PES prefix of the model. Below we present these five patterns in turn.

Task skipping (TASKSKIP). This pattern is illustrated in our running example and, for discussion purposes, in the PSP fragment presented in Fig. 19. The way this pattern shows up in the PSP bears some similarity with how the *Causality-Conflict* mismatch pattern shows up, in the sense that it requires us to combine information coming from two branches of the PSP stemming at a given state. What makes this pattern different from the *Causality-Conflict* mismatch is that the operation "match $((a_2, b_2)_C)$" (which is interfering with the event a_1 (i.e. $a_1 \#_\mu a_2$) has a counterpart match operation in the other brach, namely "match $(\{a_3, b_2\}_{\mathsf{C}})$" and both match operations involve the event b_2.

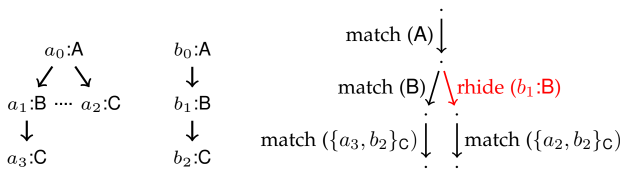


Fig. 19: Task skipping

The identification of this pattern requires a second traversal of the PSP. In the first traversal, a *Causality-Conflict* mismatch is identified and the information about the state where this mismatch is enabled is also gathered. In the second traversal, we analyze the sibling branches to look for the counterpart match operation. If the latter is found, we assert an occurrence of a *Task skipping* pattern instead of asserting an occurrence of a *Causality-Conflict* pattern. We do not provide a separate algorithm to detect this pattern as it would be largely redundant with Algorithm 6.

Unmatched repetition (UNMREPETITION). A second scenario where one hide operation cannot be matched is when the event log is capturing repetitive behavior that is not specified in the process model. In that context, every occurrence in the log of the same label will be mapped to a different event. Every time an event is repeated in the log that cannot be matched a hide operation will be appended to the PSP. Fig. 20 presents a simple example of this pattern.

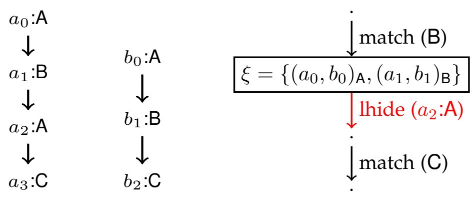


Fig. 20: Unmatched repetition

This pattern can be straightforwardly detected in the PSP by analyzing the set of matchings ξ, which is stored along with every state in the PSP. In our example, we observe that ξ contains the match $(a_0, b_0)_{\mathsf{A}}$ that carries the same label as event a_2. We can therefore conclude that there is an activity with label A that occurs twice in the same trace, but cannot be matched to the behavior specified in the model. This test can be piggybacked in Algorithm 5 in line 12 and not adding the hide operation to n_chs if it has been found to be an unmatched repetition.

Note that the symmetric case (where repetitive behavior specified in the process model cannot be matched with behavior observed in the event log) cannot be processed in the same way. This is because PESs corresponding to the process model explicitly represents repetitive behavior by means of cc-pairs and shift operations. The problem of identifying repetitive behavior captured in the model but not observed in the log is discussed later (cf. additional behavior patterns).

Task substitution (TASKSUB). In some cases, an event cannot be matched because its counterpart has been substituted by a task with a different label. Fig. 21 presents an example of this pattern.

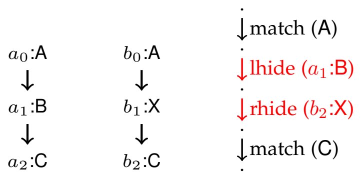


Fig. 21: Task substitution

Our assumption is that a task substitution must happen in the same execution context. Concretely, we require that the candidate events are enabled immediately after the events involved in a "match" operation. Given this requirement, the identification of this mismatch matching can be done with a variant of Algorithm 5. The changes to that Algorithm are basically to modify the condition of line 11 to eliminate the requirement about the equality of the event labels, checking that the events are immediately activated after a match operation ($e' <_m ue \wedge f' <_m uf$) and, additionally, that the hide operations are causally consistent with all the operations that precede the event match. To make this analysis deterministic, we order the hide operations in alphabetic order of the event labels.

Task relocation (TASKRELOC). Let us now consider the case where a pair of events carrying the same labels cannot be matched because they appear in different places in the same path of a PSP. The main difference with respect to the relation mismatch patterns is that the events are not enabled after the same event. One simple case is the one where the order of a pair of events in two PESs is inverted. For instance, let us assume that in one PES it holds a_1:A $<$ a_2:B whereas in the other PES it holds b_1:B $<$ b_2:A, and there exists one state in the PSP where a_1 and b_1 are both enabled. Evidently, the PSP would have two different branches, each one with two hide and one match operation, respectively. This situation can be generalized to the case the events are not contiguous, as illustrated in Fig. 22.

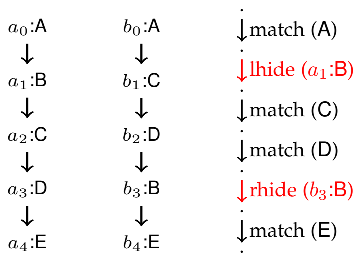


Fig. 22: Task relocation

Occurrences of the relocation pattern can be identified by keeping track of the events that were not found to be part of a relation mismatch pattern, and then checking equality of the labels associated to the hidden events.

Task absence/insertion (TASKABS). Any hide operation in the log of the PES that is not involved in any occurrence of one of the previous patterns is treated as an occurrence of a *Task absence* pattern, meaning that a task is observed in the log but missing in the model.[4] In other words, *Task absence* is a "catch all" pattern for all remaining cases of unfitting log behavior, thus ensuring that the set of patterns is complete. In the simplest case, an occurrence of the *Task absence* pattern corresponds to the situation where a task label is observed in the event log despite the fact no task with such label is specified in the process model. However, this pattern also captures the case where there exists at least a pair of events (one from each PES) with the same label, which are enabled in different states. The example shown in Fig. 23 illustrates the latter situation.

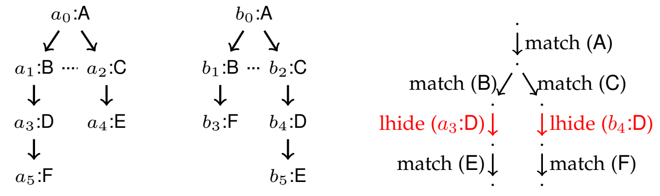


Fig. 23: Task absence/insertion

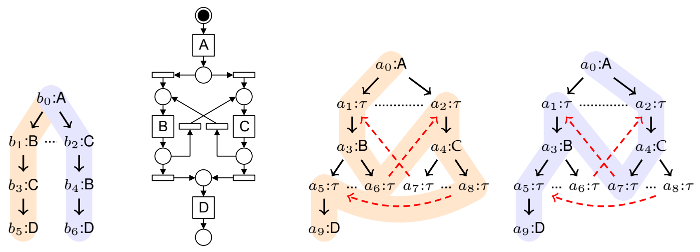
7.3 Patterns of additional model behavior

The 8 patterns presented above characterize behavior observed in the event log but not allowed in the process model. We now seek to characterize behavior allowed in the model but not observed in the log. Such additional behavior is captured by two patterns:

- *Unobserved acyclic interval* (UNOBSACYCLICINTER) – an acyclic fragment of a process model not observed in the log. Each such fragment is characterized by an initial task and a final task and is thus called an *interval*.
- *Unobserved cyclic interval* (UNOBSCYCLICINTER) – a cyclic fragment (*interval*) of a process model not observed in the log.

An example of additional model behavior is depicted in Fig. 24. Fig. 24(a) denotes a PES constructed from a log. Next, Fig. 24(b) shows a process model (in the form of a Petri net), while Fig. 24(c) shows the PES prefix derived from the model. The PES prefix appears in two copies, in order to show how each of the two paths in the PES of the log is also found in the PES of the model. In other words, there is no unfitting behavior in this example. On the other hand, there is additional behavior: PES of the log does not contain any repetitive behavior, while the process model has a loop with two entry points and two exit points. Yet, if we constructed the PSP, we would find that it contains no hide operations and it covers all events and causal relations in both PESs. This is because the PSP is constructed with the goal of finding optimal matchings for every maximal configuration of the log's PES, and does not try to achieve full coverage of the model's PES prefix. In other words, a PSP with no hide operations only means that all behavior observed in the event log fully is captured in the process model, but not vice-versa.

4. Symmetrically, we can state that a task has been inserted in the log.



(a) Log's PES (b) Model (c) PES prefix (duplicated to show paths)

Fig. 24: Additional (cyclic) model behavior

Hence, in order to characterize additional model behavior (both acyclic and cyclic), we need to define a notion of *coverage* of the PES prefix of a process model. In other words, we need to answer the question: What does it mean that all the behavior captured in a process model is "covered by" (i.e. observed in) an event log? To answer this question, we use the notions of elementary paths and elementary cycles from the field of graph theory [26], [27]. Intuitively, we will say that an event log "covers" the behavior of a process model, if every elementary path and elementary cycle in the PES prefix of the model is represented by a path in the PSP – and thus represented by a maximal configuration in the PES of the log after hide operations have been applied to account for unfitting log behavior.

We recall some basic definitions from graph theory. A directed graph is a set of vertices and a set of directed edges. A path is a sequence of vertices connected by edges. A path is said elementary if it contains no vertex twice. A cycle is a path where the initial and the final vertex are the same. A cycle is said to be elementary if, after removing the last vertex in the sequence, the resulting path is elementary.

The above concepts provide a straightforward approach to define a notion of coverage of a graph by a set of traces. However, we cannot directly apply the above concepts to characterize the possible executions a PES prefix. Indeed, a path (along the direct causality relation) in a PES prefix does not characterize a possible execution, because an execution may contain concurrent events, and a single path in the PES prefix necessarily misses some of these events. Instead, we characterize the executions of a PES prefix by means of the set of elementary paths on a graph where the vertices are the configurations explicitly represented in the PES prefix, and the edges are the possible configuration extensions (i.e. direct transitions from one configuration to the next), including possible extensions induced by a cc-pair in the PES prefix.

In order to reason over this graph of configurations and configuration extensions, we rely on the notion of *pomset* (standing for *partially ordered multisets*) from the literature of concurrency theory [28]. A pomset is a Directed Acyclic Graph (DAG) where the nodes are configurations, and the edges represent direct causality relations between configurations. An edge is labeled by an event. Unlike an event structure, a pomset does not have any conflict relation, since a pomset represents one possible execution. The behavior of a PES can be characterized by the set of pomsets it induces.[5]

In the case of a PES prefix, the set of induced pomsets is infinite when the PES prefix captures cyclic behavior via cc-pairs. Hence in general we cannot enumerate all pomsets of a PES prefix in order to check if each of them is observed in the PES of the log. However, we can extract a set of elementary pomsets (inspired by the notion of elementary paths), which collectively cover all the possible pomsets induced by a PES prefix without unfolding the cyclic behavior infinitely. Intuitively, this corresponds to unfolding every cycle so that it is traversed once only.

Algorithm 7 Identification of elementary pomsets

```
 1: procedure FINDEPOMSETS(conf, sconf, visited, var cycles, var runs)
 2:     APPEND(visited, (conf, sconf))
 3:     for (sconf $\xrightarrow{e}$ n_sconf) do
 4:         n_conf ← conf ∪ {e}
 5:         if $e$ is cutoff event then
 6:             n_sconf ← $\mathcal{S}$(n_sconf)
 7:         end if
 8:         if ∃(entryConf, n_sconf) ∈ visited ∧ n_sconf ∩ ||[e] = ∅ then
 9:             cycles ← cycles ∪ {(n_conf \ entryConf, entryConf)}
10:             ADDBRANCHTOEXPPREFIX(visited)
11:         else
12:             FINDEPOMSETS(n_conf, n_sconf, visited, cycles, runs)
13:         end if
14:     end for
15:     if sconf is a maximal configuration then
16:         runs ← runs ∪ {conf}
17:         ADDBRANCHTOEXPPREFIX(visited)
18:     end if
19:     REMOVELAST(visited)
20: end procedure
```

This intuition is formalized by Algorithm 7, which computes the set of *elementary pomsets* of a PES prefix. Fig. 25 illustrates the execution of this algorithm taking as input the PES shown in Fig. 24. Function FINDEPOMSETS builds an expanded prefix by successively applying configuration extensions and shift operations (cf. Section 5.1) on the PES prefix. Specifically, function FINDEPOMSETS adds one path (or branch) to the expanded prefix every time an elementary pomset is found (in lines 10 and 17). The result is a directed acyclic graph reflecting the configuration extension relation. For illustration purposes, Fig. 24 presents the expanded prefix as a PES prefix, with the corresponding label in the right-hand side of each "event" in the expanded prefix.

A key observation is that the value of conf can be used as a unique identifier for each event in the expanded prefix. The uniqueness of conf stems from the following facts: 1) for elementary acyclic pomsets, conf is finitely extended by a different event until a complete configuration is found, 2) for elementary cyclic pomsets, conf would be finitely extended up to the point where a duplicate event. It is by means of the shifted version of conf, i.e. the variable sconf, that cycles can be identified.

The proofs of completeness and correctness of Algorithm 7 follow directly from the proofs for algorithms for identifying elementary cycles [26], [27]. With the aim of explaining how the Algorithm 7 works and to sketch the proofs, we describe three cases.

Case 1: *Identification of elementary acyclic pomsets that include no cutoff event.* This case is illustrated with the sequence of "events" that are shown in blue font in the expanded prefix shown in Fig. 25. The function FINDEPOMSETS is first called with conf and sconf set to empty set. We note that in this case, conf and sconf are updated in such a way that they both hold the same value. The function FINDEPOMSETS extends the configuration conf by one event in line 4 and recursively calls itself in line 12. The recursive call will eventually stop, when sconf contains a maximal configuration, because a maximal configuration has no further possible extension. Moreover, a maximal configuration is warrantied to be found because we consider only PESs coming from sound systems and consisting of a finite number of events. The elementary acyclic pomset that has been found is added to the set runs in line 16, just before the recursive call returns.

Case 2: *Identification of elementary acyclic pomsets that contain at least one cutoff event.* This case is illustrated with the sequence of events in the blue dashed box in the expanded prefix in Fig. 25. As for the previous case, conf and sconf contain the same value at every recursive call of FINDEPOMSETS, as long as no cutoff event is found. When a (forward) cutoff event is found, such an event is added to conf and, afterwards, n_sconf is assigned with the shifted configuration $\mathcal{S}(\text{sconf} \oplus \text{e})$, in line 6. In line 12, FINDEPOMSETS is recursively called with the extended configuration n_conf and also with the shifted configuration n_sconf. Note that sconf is used for testing if the function has found a maximal configuration in line 15 and also to compute the set of possible extensions in line 3. Conceptually, conf keeps track of the events that are "executed" by the underlying run, dynamically unrolling towards a larger prefix, whereas sconf maps the execution back to a configuration in the original PES prefix. Note that the recursive call to the function can only find a finite number of forward cutoff events. Thus, if no cycle is found as the recursive call to FINDEPOMSETS proceeds, the function will extend the configuration until a maximal configuration is found.

Case 3: *Identification of elementary cyclic pomsets.* We observe that, when the input PES has repetitive behavior, the function FINDEPOMSETS can only be recursively called a finite number of times before it finds an elementary cyclic pomset, because the PES has only a finite number of events. One case corresponds with finding a cutoff event that induces a backward shift. For simplicity, let us assume that the input PES has only one cutoff. Let e be a cutoff event. By definition, we know that e induces a backward shift if and only if $\lceil \text{corr}(e) \rceil \subset \lceil e \rceil$. Therefore, there exists a sequence of calls that finds first $\lceil \text{corr}(e) \rceil$, storing the pair $(\lceil \text{corr}(e) \rceil, \lceil \text{corr}(e) \rceil)$ in visited. Since the input PES has a finite number of events, FINDEPOMSETS will eventually process $\lceil e \rceil$, which will induce a backward shift operation in line 6. Since visited contains a pair associated with such a configuration, FINDEPOMSETS will record a new elementary cyclic pomset in line 9. Note that the cycle is stored with a pair of configurations, $(\lceil e \rceil \setminus \lceil \text{corr}(e) \rceil, \lceil \text{corr}(e) \rceil)$ in our example, where the first set corresponds with the set of events in the body of the cycle, herein called the *cyclic interval*, and the configuration characterizing the entry point

5. This is exactly the definition of a notion of equivalence known as visible pomset equivalence: two PES are equivalent iff they induce the same set of pomsets.

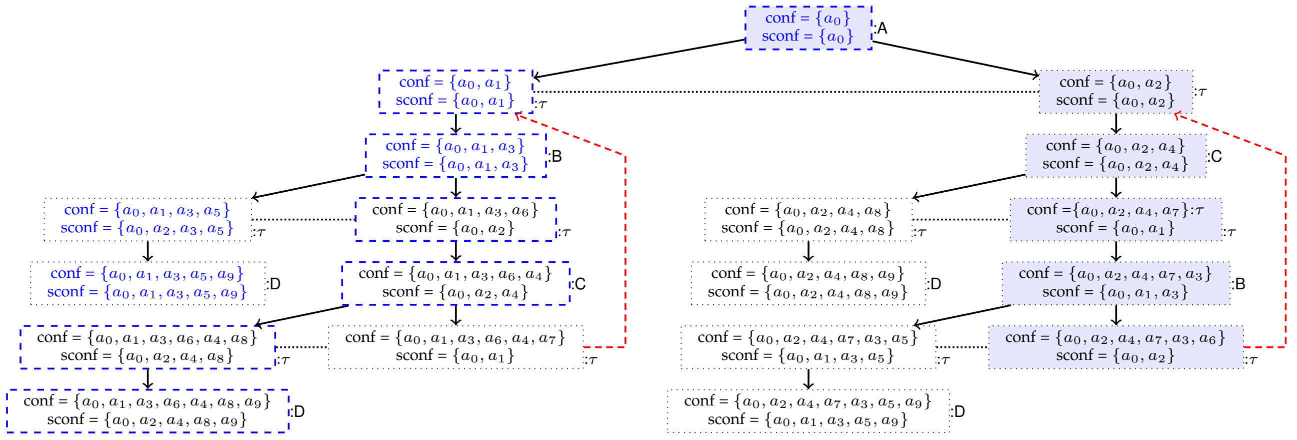


Fig. 25: Expanded prefix with elementary pomsets of PES in Fig. 24(c)

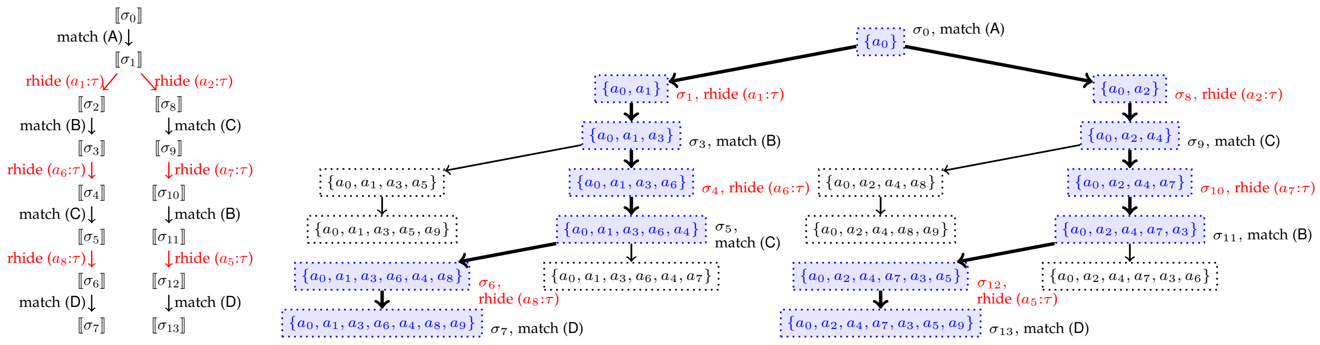


Fig. 26: Using PSP and expanded prefix for identifying additional model behavior in the example shown in Fig. 24

to the cyclic behavior.

However, an elementary cyclic pomset does not always involve a backward shift as it is the case for the PES prefix shown in Fig. 24. In fact, the PES prefix has two elementary cycles but no backward cutoff. The elementary cycles can still be detected, because we store the shifted configuration (along with the unshift configuration) at every recursive call of the function FINDEPOMSETS. This information can then be used to check if the current shifted configuration has been previously visited, in which case an elementary cyclic pomset is reported. One example of this case is illustrated by the sequence of events shown in the filled blue boxed in the expanded prefix in Fig. 25.

When an elementary cyclic pomset is embedded in a block of concurrency, FINDEPOMSETS will find the cycle multiple times. For instance, for the PES prefix shown in Fig. 27, FINDEPOMSETS will find an elementary cycle comprising the set of events $\{a_1, a_4\}$, when the function processes the configuration $\{a_0, a_1\}$ and the cutoff event a_4. The same cycle will be found when the function processes the configuration $\{a_0, a_1, a_2\}$ and later when it processes the configuration $\{a_0, a_1, a_2, a_3\}$. In order to prevent the recording of multiple copies of the same elementary cyclic pomset, line 8 checks if n_sconf contains the cycle and no other concurrent event. Only when that condition holds, the elementary cyclic pomset is retained.

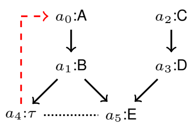


Fig. 27: Cycle within a block of concurrency

The set of elementary pomsets along with the expanded prefix are then used for identifying all the additional model behavior as follows. Traverse the PSP in depth-first search order. At every step, when operation associated with an arc involves an event coming from the model, mark the "event" in the expanded prefix associated with the configuration C^r, with a reference to the operation and the state in the PSP that is reached as an outcome to the operation at hand. Fig. 26 illustrates the result of the previous stage on the example PES presented in Fig. 24. The "events" in the expanded prefix that have been marked are shown with a blue background. Additionally, the information about the state in the PSP and the corresponding operation is shown to the right-hand side of each "event" in the expanded prefix.

In a second stage, we iterate over the set of elementary pomsets to identify those that were not marked. When an elementary pomset is not marked, we find it in the expanded prefix and traverse bottom-up the prefix to find the closest "event" marked with a match operation in the PSP. The state in the PSP associated with this "event" serves to give context to an occurrence of one of the two additional behavior patterns (UNOBSACYCLICINTER and UNOBSCYCLICINTER). Let us consider again the example shown in Fig. 26. If we proceed from left to right, the first elementary pomset is associated with the "event" labeled as $\{a_0, a_1, a_3, a_5, a_9\}$ and its closest match operation occurs at state σ_3 in the PSP. This case corresponds to an elementary acyclic pomset. Part of the pomset has been observed, i.e. the sequence of tasks

A and B, and only task D was not observed. The interval of tasks that are not observed can be computed with a set difference. Thus, in this example we will assert a mismatch with the constructor $\text{UNOBSACYCLICINTER}(\sigma_3, \{a_5, a_9\})$. Given that a_5 corresponds to an invisible task, one can remove that event from the difference diagnosis. In this case, we can report that in the event log the interval of tasks between a_5 and a_9 is not observed.

The second elementary pomset in the example is associated with the "event" labeled as $\{a_0, a_1, a_3, a_6, a_4, a_7\}$ and its closest match operation occurs at state σ_5 in the PSP. This case corresponds to an unobserved elementary cyclic pomset. The diagnostic will be asserted with the constructor $\text{UNOBSCYCLICINTER}(\sigma_5, \{a_3, a_6, a_4, a_7\})$, meaning that the loop formed by tasks B and C is not observed in the log. Note that in this example, there is a loop with two entries and two exits comprising tasks B and C. When expanded, this loop leads to two elementary cyclic pomsets not covered by the PES of the event log and thus two occurrences of the UNOBSCYCLICINTER mismatch pattern

In summary, for the example in Fig. 24 we will assert four mismatches:

Two unobserved elementary acyclic pomsets
- $\text{UNOBSACYCLICINTER}(\sigma_3, \{a_5, a_9\})$
- $\text{UNOBSACYCLICINTER}(\sigma_9, \{a_8, a_9\})$

Two onbserved elementary cyclic pomsets
- $\text{UNOBSCYCLICINTER}(\sigma_5, \{a_3, a_6, a_4, a_7\})$
- $\text{UNOBSCYCLICINTER}(\sigma_{11}, \{a_4, a_7, a_3, a_6\})$

7.4 Verbalization

The last step in the method is to turn occurrences of mismatch patterns identified using the PSP, into plain natural language statements that can be interpreted by users. Table 1 shows the statements corresponding to each of the nine mismatch patterns defined in Section 7. For some of the mismatch patterns, we identify multiple sub-cases based on conditions on the events of the log and/or model, leading to more than one statement type per mismatch pattern. This depends on whether the events in question are causal or concurrent, or if the event in the log is defined. As a result, the nine constructors give rise to 16 different types of difference statements.

In each statement, the states of the PSP (σ, and when required, also σ') are used to precisely localize where the difference occurs in the log and/or in the model. The text "after σ" means that a difference is observed *immediately* after the occurrence of that state.

8 EVALUATION

We implemented the proposed method in a tool called ProConformance[6]. This tool takes as input a process model in BPMN format and a log in MXML or XES format. Its output is a set of difference statements. The tool allows users to customize the output by switching on/off PSP states, and selecting which elements of a state to show, e.g. only the last matched event.

Using this tool, we conducted a two-pronged (qualitative and quantitative) evaluation of the proposed method.

6. Available at http://apromore.org/platform/tools

First, we performed a qualitative evaluation of the output produced by the method on a real-life event log and a corresponding process model. Next, we performed a quantitative evaluation of time performance and number of produced difference statements, based on large collections of real-life process models. In both evaluations, we compared our method against the *trace alignment* method, which, as discussed in Section 2, is a state of the art method in business process conformance checking.

8.1 Qualitative evaluation

For the qualitative evaluation, we used a publicly available log extracted from an information system for managing road traffic fines in Italy [29], and a normative process model, which we derived from the description of this business process in [9]. The normative model in Petri nets is shown in Fig. 28. This traffic fines management process starts when a fine is created. The fine can be paid by the offender right away, after a notification is sent to the offender by the police, or when the offender receives the notification. The payment itself can be done in one or more instalments, depending on the amount of the fine. The case is closed as soon as the payment for the full amount has been done. If the fine is not paid within 180 days, a penalty is charged on top of the fine and if after further 180 days the fine is still due, a credit collection organization will take over the handling of the case. At any time after receiving the notification, the offender can appeal against the fine through a judge or a prefecture. In case of a successful appeal, the case is dismissed and the process ends. If the appeal is unsuccessful, the fine is still to be paid. An appeal can be made more than once, depending on the circumstances (e.g. when escalating the appeal to a higher court).

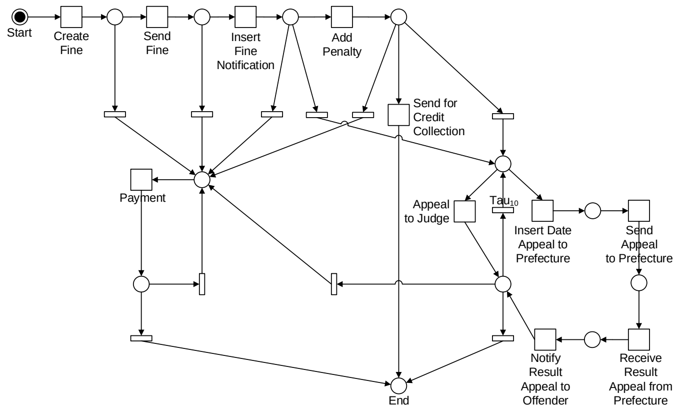

Start
Create Fine
Send Fine
Insert Fine Notification
Add Penalty
Send for Credit Collection
Payment
Appeal to Judge
Tau10
Insert Date Appeal to Prefecture
Send Appeal to Prefecture
Notify Result Appeal to Offender
Receive Result Appeal from Prefecture
End

Fig. 28: Traffic fines management process model.

The log covers fines recorded in the period 2000–2013. It contains 150,370 traces comprising 231 distinct traces and a total of 561,470 events.

We assessed the conformance of this log with the Petri net of Fig. 28. Our method produced 15 distinct statements capturing all the differences between the log and the normative model. As an example, the following statements were retrieved (states are indicated through the last matched event):

1) *In the log, "Send for credit collection" occurs after "Payment" and before the end state*

Constructor	Condition	Statement type
CAUSCONC(σ, e, f, e', f',coff)	if $e' < e$	*In the log, after σ, $\lambda(e')$ occurs before $\lambda(e)$, while in the model they are concurrent*
	else	*In the model, after σ, $\lambda(f')$ occurs before $\lambda(f)$, while in the log they are concurrent*
CONFLICT(σ, e, f, e', f',coff)	if $e' \parallel e$	*In the log, after σ, $\lambda(e')$ and $\lambda(e)$ are concurrent, while in the model they are mutually exclusive*
	else if $f' \parallel f$	*In the model, after σ, $\lambda(f')$ and $\lambda(f)$ are concurrent, while in the log they are mutually exclusive*
	else if $e' < e$	*In the log, after σ, $\lambda(e')$ occurs before task $\lambda(e)$, while in the model they are mutually exclusive after σ*
	else	*In the model, after σ, $\lambda(f')$ occurs before $\lambda(f)$, while in the log they are mutually exclusive*
TASKSKIP(σ, e, f, e', f',coff)	if $e \neq \perp$	*In the log, after σ, $\lambda(e)$ is optional*
	else	*In the model, after σ, $\lambda(f)$ is optional*
TASKSUB(σ, e, f, e', f',coff)		*In the log, after σ, $\lambda(f)$ is substituted by $\lambda(e)$*
UNMREPETITION(σ, e, f, e', f',coff)		*In the log, $\lambda(e)$ is repeated after σ*
TASKRELOC($\sigma, e, f, \sigma', e', f'$)	if $e \neq \perp$	*In the log, $\lambda(e)$ occurs after σ instead of σ'*
	else	*In the model, $\lambda(f)$ occurs after σ instead of σ'*
TASKABS(σ, σ', e, f)	if $e \neq \perp$	*In the log, $\lambda(e)$ occurs after σ and before σ'*
	else	*In the model, $\lambda(f)$ occurs after σ and before σ'*
UNOBSACYCLICINTER(σ, inter)		*In the log,* inter *do(es) not occur after σ*
UNOBSCYCLICINTER(σ, inter)		*In the log, the cycle involving* inter *does not occur after σ*

TABLE 1: Verbalization of mismatch patterns

2) *In the model, after "Insert fine notification", "Add penalty" occurs before "Appeal to judge", while in the log they are concurrent*
3) *In the log, after "Add penalty", "Receive results appeal from prefecture" is substituted by "Appeal to judge"*
4) *In the log, the cycle involving "Insert date appeal to prefecture, Send appeal to prefecture, Receive result appeal from prefecture, Notify result appeal to offender" does not occur after "Insert fine notification".*

Statement 1 (an example of task insertion) denotes a potential compliance issue: credit collection should never occur if the payment has been done, though there are cases in the log where this happens. Similarly, Statement 2 (an example of causality/concurrency mismatch) indicates that there are cases in the log where the penalty is charged even after the appeal, while this should be done only if the appeal is unsuccessful. Given that these two events have been observed in any order in the log, they are identified as concurrent. These compliance issues may be related to recording errors in the system (e.g. a payment not being recorded or being recorded for a lower amount).

Statement 3 (an example of task substitution) pinpoints that in the log there are traces where after "Add penalty", event "Receive results appeal from prefecture" is observed. In the PSP, this event in the log is substituted by "Appeal to judge" in the model, after which we know the process can complete. This means that tasks "Insert date appeal to prefecture", "Send appeal to prefecture" and "Notify result appeal to offender", which are in the same path as "Receive results appeal from prefecture" in the model, are not observed in the log. The method substitutes "Receive results appeal from prefecture" with "Appeal to judge" because this minimizes the number of mismatches, as opposed to skipping the three tasks above.[7] This statement suggests that in some cases, the results of an appeal to the prefecture are received by the police, without the appeal having actually been lodged by the offender. This might be due to a mistake at the prefecture (e.g. fines being swapped), which explains why the police does not notify the offender (event "Notify result appeal to offender" is not observed after "Receive results appeal from prefecture").

Finally, Statement 4 (an example of unobserved elementary cycle) indicates that while in principle an offender can appeal to the prefecture multiple times, this has not being observed in the log. Given that the log covers over 10 years of behavior, this may suggest that our model perhaps generalizes the behavior in the log, or that subsequent appeals are never recorded in the system.

For trace alignment, we used the plugins "Replay a Log on Petri Net for All Optimal Alignments"[8] and "Replay a Log on Petri Net for Conformance Analysis"[9] for the ProM 6.5.1 environment. Both plugins report on conformance issues related to fitness, by computing several visual diagnostics as well as a fitness metric (a value from 0 to 1). The former plugin finds *all optimal alignments* for each distinct trace of the log, while the latter provides a good approximation of this result by computing only *one optimal alignment* per distinct trace.

The main diagnostic consists in projecting the results of alignment onto the log, which results in a list of individual trace alignments (a small except of this view for our example is shown in Figure 29). Besides statistics on fitness, this diagnostic shows a great deal of information for each distinct trace, including the exact order in which synchronous moves, (silent) moves on model and moves on log occur, and for each move, the label of the involved event.

7. The same holds with task "Send for credit collection" though the substitution considers the lexicographical order of task labels.

8. Parameters used: graph-based state space replay to obtain all optimal alignments with maximum explored states equal to 10,001,000.

9. Parameters used: A^* cost-based fitness express with ILP with maximum explored states equal to 10,001,000.

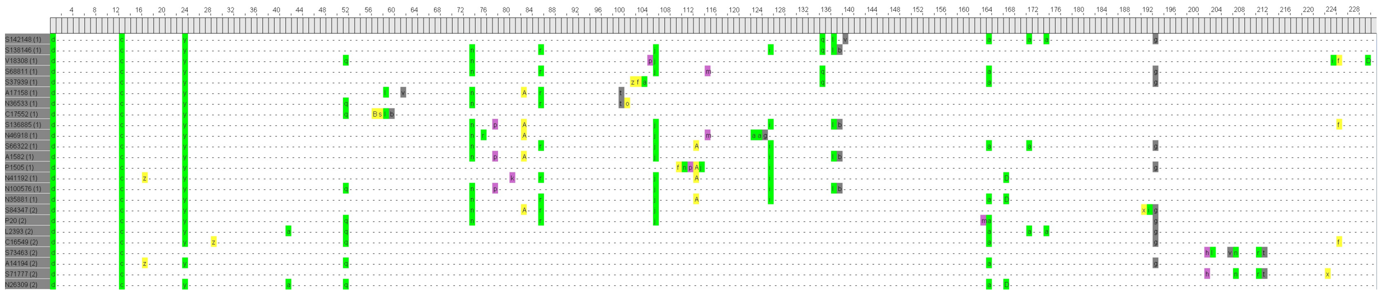
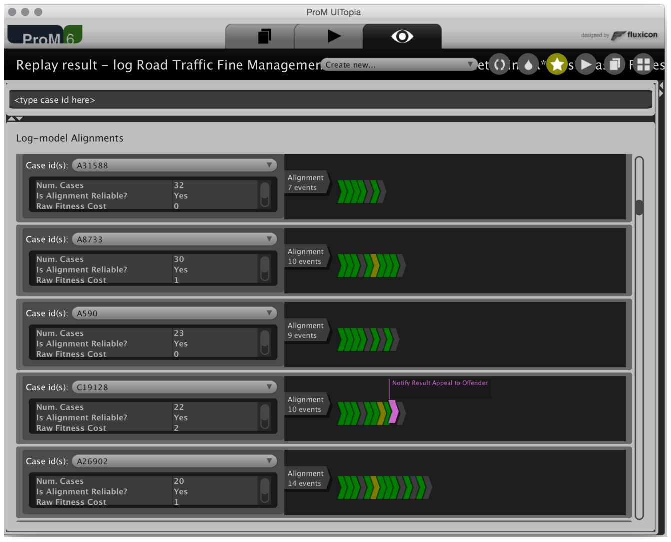


Fig. 30: Excerpt of tabular view of trace alignments for the traffic fine management process.



Case id(s): A31588
Num. Cases 32
Is Alignment Reliable? Yes
Raw Fitness Cost 0
Alignment 7 events

Case id(s): A8733
Num. Cases 30
Is Alignment Reliable? Yes
Raw Fitness Cost 1
Alignment 10 events

Case id(s): A590
Num. Cases 23
Is Alignment Reliable? Yes
Raw Fitness Cost 0
Alignment 9 events

Case id(s): C19128
Num. Cases 22
Is Alignment Reliable? Yes
Raw Fitness Cost 2
Alignment 10 events
Notify Result Appeal to Offender

Case id(s): A26902
Num. Cases 20
Is Alignment Reliable? Yes
Raw Fitness Cost 1
Alignment 14 events

Fig. 29: Excerpt of alignments projected on log for the traffic fine management process (green = synchronous move, purple = move on model, grey = silent move on model, yellow = move on log).

The "Replay a Log on Petri Net for Conformance Analysis" plugin, while providing a sub-optimal solution, offers a range of additional diagnostics. For example, one can visualize all trace alignments in a single tabular view and apply various filters on top of it (an excerpt is shown in Figure 30). More interestingly, one can also project the results of alignment onto the normative Petri net (see Fig. 31). This diagnostic can be used to show which model tasks are often skipped (those with a red border), and when tasks that should not be performed according to the model are actually performed according to the log (the darker the color of a path, the more frequent the path is executed in the log). Further, a colored bar at the bottom of a task box shows the ratio between the number of times the task is executed synchronously in the log and in the model (called synchronous move) and the number of times the task is only executed in the model (called move on model).

Although the model view pinpoints, to a certain extent, differences in executions, the exact differences have to be obtained by inspecting the individual misalignments, i.e. those trace alignments that have at least one move on model or on

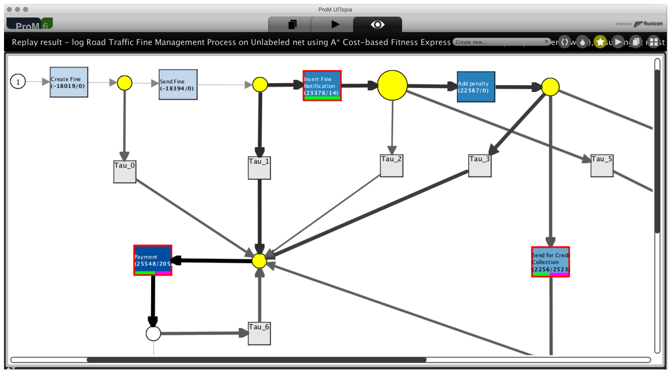

ProM UITopia
ProM 6
designed by fluxicon
Replay result – log Road Traffic Fine Management Process on Unlabeled net using A* Cost-based Fitness Express
Create new...
1
Create Fine (-18019/0)
Send Fine (-18394/0)
Insert Fine Notification (23378/14)
Add penalty (22587/0)
Tau_0
Tau_1
Tau_2
Tau_3
Tau_5
Payment (25548/205
Send for Credi Collection (2256/2523)
Tau_6

Fig. 31: Alignments projected on model for the traffic fine management process.

log.[10] This requires additional analysis. In our example, we need to examine 205 misalignments out of 231 alignments when using one-optimal alignment, and 406 misalignments out of 412 alignments when using all-optimal alignments. Still, differences related to additional model behavior, such as that captured by Statement 4 with our method, cannot be distilled from the misalignments as these only focus on fitness. For this, the underlying technique of the plugin "Check Precision based on Align-ETConformance" could be used, which relies on the prefix automaton built from trace alignments to identify the escaping edges from which the additional model behavior starts, as discussed in Section 2.[11] In our example, however, the escaping edge being reported would be the invisible task Tau_{10}, because this is the last event before the tasks in the interval referred to by Statement 4 can be repeated. From this, by looking at the model, one may infer that there are tasks in the model that can be repeated after Tau_{10}, which are not observed in the log. Similarly, Statement 2 refers to two events being concurrent in the log and causal in the model. This difference cannot be detected by examining the misalignments, because in trace alignment diagnostics are provided at the level of *individual* traces, while concurrency is a behavioral relation that can only be observed *across* traces.

10. Silent moves on model are excluded as they do not capture observable differences.

11. This plugin only provides statistics on precision such as a precision metric. However, one could extract the escaping edges from the code.

8.2 Quantitative evaluation

In order to test the scalability of our approach to increasing model and log complexity, we used two collections of process models: the IBM Business Integration Technology (BIT) library, a publicly-available collection of process models in financial services, telecommunication and other domains, gathered from IBM's consultancy practice [30],[12], and the SAP R/3 collection, the reference model used by SAP to customize their R/3 ERP product, documented in [31].

The BIT collection contains 735 models, while the R/3 collection contains 604 models. We extracted 348, respectively, 494 models from these collections, by removing models that were not single-entry single-exit (i.e. models that were not Workflow nets) and that were behaviorally incorrect (i.e. unsound).

For each model, we generated an event log using the ProM plugin "Generate Event Log from Petri Net" documented in [32]. This plugin generates a distinct log trace for each possible execution sequence in the model.[13] The tool was only able to parse 274 models from the BIT collection, and 438 models from the R/3 collection, running into out-of-memory exceptions for the remaining models. As such, our quantitative evaluation is based on the logs generated from 712 sound Workflow nets. The statistics on these models are provided in Table 2. The models range from simple ones, with a minimum of 7 nodes and a small number of XOR and AND splits, to very large and complex models with up to 177 nodes and a large number of XOR and AND splits with many outgoing arcs.

Collection		Size	#XOR splits	Outdegree XOR	#AND splits	Outdegree AND
BIT	Min	7	0	2	0	2
	Max	177	5	10	33	7
	Mean	38.10	0.61	2.42	6.49	2.08
	StDev	30.08	1	1.18	5.72	0.42
R/3	Min	7	0	2	0	2
	Max	85	4	8	4	5
	Mean	27.62	0.59	2.48	0.83	2.29
	StDev	17.78	0.82	0.94	0.86	0.61

TABLE 2: Statistics on model complexity.

Next, in order to create random differences between each log and its corresponding model, we injected noise in each original log. We achieved this by repeatedly adding or removing a random event that already existed in the original log in a random position of a randomly selected trace, until the number of added and removed events equals a percentage of the total number of events in the original log. We applied four noise levels, corresponding to 5%, 10%, 15% and 20% of the total number of events, thus obtaining four "noisy" variants for each original log. The noise injection procedure is inspired by the technique documented in [33]. Before performing this operation, we duplicated each distinct trace in every original log, so that each distinct execution sequence in the corresponding model is represented twice in the log. We did so in order not to increase the total number of traces in the log when injecting noise.

12. The BIT collection is available at http://apromore.qut.edu.au
13. Parameters used: simulation method: complete generation; min./max. traces to add for each generated sequence: 1; max. times marking seen: 2; only include traces that reach end state; only include traces without remaining tokens.

Table 3 provides statistics on the complexity of the logs for both collections, divided by noise level. The logs range from 6 to 1,433 total events, with a maximum of 38 traces (having 29 distinct traces on average) in the case of the BIT collection, and from 6 to 9,462 total events, with a maximum of 840 traces (38 distinct traces on average) for the R/3 collection. In the remainder, with *total log size* we refer to the total number of events, which corresponds to the sum of the lengths of the traces.

Collection		No noise	5% noise	10% noise	15% noise	20% noise
BIT	Min	6	6	7	7	7
	Max	1,432	1,427	1,433	1,428	1,428
	Mean	58	58	58	57	57
	StDev	120	120	120	120	120
R/3	Min	6	6	7	7	7
	Max	9,408	9,410	9,432	9,449	9,462
	Mean	515	514	515	515	515
	StDev	1,377	1,377	1,379	1,382	1,384

TABLE 3: Total log size in terms of number of events.

Using these two logsets, we measured the execution time and counted the number of statements provided by our method for each model-log pair along all noise levels. We performed the tests on a computer with a dual core Intel Core i7-4710HQ 2.5GHz (4 cores), 16GB RAM, running Windows 8.1 x64 and Java 1.8.0_31 with 14GB of allocated memory. To eliminate load time from the measures, we executed each test five times and recorded average times of three executions, removing the fastest and the slowest executions.

The execution times against the total log size for each noise level are plotted in Figures 32 (BIT) and 33 (R/3), where the measuring points are color-coded depending on the range of execution times. Summary statistics are shown in Table 4.

Without noise, the execution time is linear on the log size ($R^2 = 0.83$ for BIT and 0.95 for R/3), reaching a peak of 72ms for a log of 1,432 events (BIT) and 107ms for a log of 8,352 events (R/3). The plots show that the discrepancies between the model and the log, due to the injected noise, result in higher execution times than the case without noise. This is due to the complexity of building the PSP. Still, execution times are always under 10sec for BIT (6.6sec max at 20% noise on a log of 1,428 events and model of 177 nodes) and under 2min for R/3 (max 103sec at 20% noise on a log of 8,352 events and model of 68 nodes).

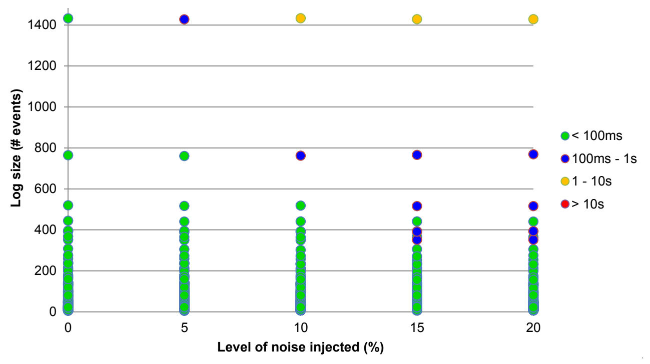


Fig. 32: Effect of log size and noise on the time performance of our method (BIT).

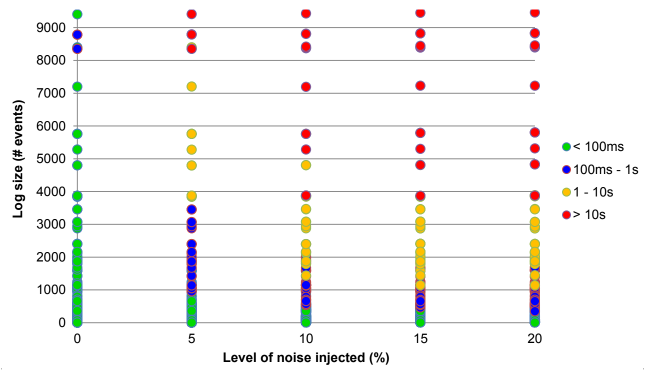


Fig. 33: Effect of log size and noise on the time performance of our method (R/3).

Collection		No noise	5% noise	10% noise	15% noise	20% noise
BIT	Min	3	3	3	3	3
	Max	72	224	1,037	2,704	6,625
	Mean	7	9	13	20	36
	95%	18	21	24	26	30
	StDev	6	15	63	163	400
R/3	Min	3	3	3	3	3
	Max	107	56,251	63,210	99,724	103,257
	Mean	11	522	951	1,426	1,432
	95%	38	746	2,258	4,333	4,587
	StDev	16	3,783	5,501	8,395	7,948

TABLE 4: Execution time (ms) for each logset using our method.

Table 5 reports the number of statements produced by our method for each logset. The noiseless logs all produce zero statements (as they are an exact representation of the model and, hence, do not differ from its behavior). In the extreme case of a log with 20% noise, 104 statements were required to describe all the differences for the BIT collection (with a log of 1,428 events and model of 177 nodes – this is the pair that took 6.6sec to be compared), and 593 statements for the R/3 collection (with a log of 8,352 events and model of 68 nodes).

Comparing execution times with number of statements, we can observe a relatively sharp increase in average execution time between 0% and 5% noise in the R/3 data set (11ms vs 522ms). This, however, coincides with a similarly sharp increase in the amount of produced statements (0 vs 18 on average and 7,469 in total). The additional statements required for e.g. the 10% noise level compared to the 5% noise level is smaller, resulting in a similarly smaller increase in required execution time.

Collection		No noise	5% noise	10% noise	15% noise	20% noise
BIT	Min	0	0	0	0	0
	Max	0	42	65	89	104
	Mean	0	2	4	5	7
	StDev	0	4	7	9	11
	Total	0	480	943	1,409	1,776
R/3	Min	0	0	0	0	0
	Max	0	370	408	544	593
	Mean	0	18	28	37	43
	StDev	0	43	62	79	88
	Total	0	7,469	11,720	15,532	18,048

TABLE 5: Statements produced for each logset.

Next, we carried out the same tests using trace alignment with all optimal alignments (using the plugins "Replay a Log on Petri Net for All Optimal Alignments" to obtain the individual trace alignments, and "Check Precision based on Align-ETConformance" to obtain the escaping edges[14]). The execution times against log size and noise level are reported in Figures 34 (BIT) and 35 (R/3), while Table 6 provides the summary statistics. Similar to our method, performances grow linearly on the log size in the case of no noise (R^2=0.88 for BIT and 0.97 for R/3). Comparatively, trace alignment is faster than our method, reaching a peak of just 0.5sec for BIT and 1.9sec for R/3 in the case of 20% noise, compared to 6.6sec, respectively, 103sec, with our method. In particular, the difference in performance is more evident for larger logs with many model-log discrepancies, where the complexity of the PSP is exposed. Conversely, on simpler logs our method tends to be slightly faster than trace alignment. This is the case for the model-log pairs with up to 5% noise levels for the BIT collection, and the model-log pairs with no noise for the R/3 collection.

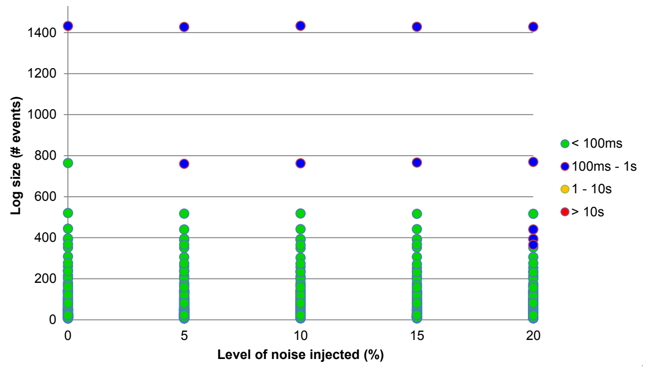


Fig. 34: Effect of log size and noise on the time performance of all optimal alignments (BIT).

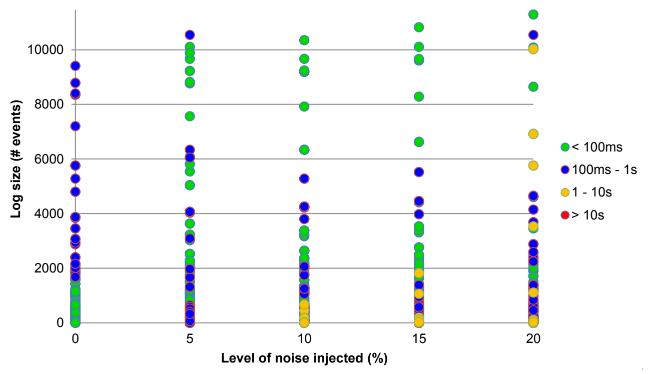


Fig. 35: Effect of log size and noise on the time performance of all optimal alignments (R/3).

Tables 7 and 8 report the number of misalignments and escaping edges. From these we can observe that the number of diagnostics provided by trace alignment is significantly higher than that reported by our method, with a total of 6,968 misalignments + escaping edges for the BIT collection and 153,698 misalignments + escaping edges for the R/3 collection (summing up across all noise levels), compared to a total of 4,608 statements, respectively, 52,769 statements, with our method.

14. Parameters used in the latter plugin: ordered representation; all optimal alignments.

Trace alignment reports escaping edges also in the case of logs with no noise (316 for BIT and 2,495 for R/3). This is due to the fact that these edges are detected whenever there is repetitive behavior (i.e. infinite behavior) in the model, since the log records finite behavior. For example, if a loop in the model is only observed twice in the log, an escaping edge will be reported on the state enabling the third iteration of this loop in the model.

Collection		No noise	5% noise	10% noise	15% noise	20% noise
BIT	Min	2	2	3	3	3
	Max	102	221	317	414	522
	Mean	7	10	11	12	13
	95%	25	31	36	38	42
	StDev	10	18	24	30	37
R/3	Min	2	2	2	2	2
	Max	475	919	1,218	1,539	1,910
	Mean	29	55	72	86	98
	95%	148	290	406	486	553
	StDev	70	143	193	239	282

TABLE 6: Execution time (ms) for each logset using all optimal alignments.

Collection		No noise	5% noise	10% noise	15% noise	20% noise
BIT	Min	0	0	1	1	1
	Max	0	36	61	69	94
	Mean	0	3	4	6	7
	StDev	0	4	6	8	10
	Total	0	759	1,194	1,567	1,864
R/3	Min	0	0	1	1	1
	Max	0	826	1,689	2,554	3,711
	Mean	0	35	65	93	121
	StDev	0	96	180	267	358
	Total	0	14,287	26,285	37,271	48,725

TABLE 7: Misalignments for each logset using all optimal alignments.

Collection		No noise	5% noise	10% noise	15% noise	20% noise
BIT	Min	0	0	0	0	0
	Max	8	8	7	7	7
	Mean	1	1	1	1	1
	StDev	1	1	1	1	1
	Total	316	315	317	316	320
R/3	Min	0	0	0	0	0
	Max	494	1,246	1,533	1,795	1,821
	Mean	6	14	16	16	16
	StDev	32	78	92	102	102
	Total	2,495	5,748	6,266	6,405	6,216

TABLE 8: Escaping edges for each logset using all optimal alignments.

9 CONCLUSION

We presented an approach for checking the conformance between an event log capturing the actual execution of a business process, and a model capturing its expected or normative execution. The method relies on a unified representation of process models and event logs. Specifically, the event log is folded into an event structure and the process model is unfolded into another event structure. The two event structures are then compared via a partially synchronized product, from which a complete set of behavioral differences between the model and the log is extracted.

We empirically compared the proposed method to existing conformance checking methods both in terms of execution times and size of the difference diagnosis. A qualitative evaluation based on a real-life event log and a corresponding process model showed that the presented approach produces more compact, yet much more understandable diagnosis than conformance checking methods based on trace alignment. The evaluation also showed that the proposed method exposes behavioral differences that are difficult or impossible to identify using trace alignment techniques.

Meanwhile, a quantitative evaluation based on two real-life collections with over 700 process models in total, showed that the proposed approach, while being generally slower than trace alignment, it has reasonable execution times (within 10 seconds). In extreme cases involving logs with over 8,000 event occurrences (considering distinct traces only) and a high number of differences between the process model and the event log, the execution time is still below 2 minutes. The quantitative evaluation also showed that the proposed approach consistently produces more compact difference diagnosis than trace alignment methods.

A limitation of the proposed method is that it treats the input log as consisting of sequences of event labels, thereby ignoring timestamps and event payloads. Possible directions for future work include designing temporal and data-aware extensions of the method, along the lines of data-aware extensions of trace alignment methods [9].

In addition to these possible extensions, there are also multiple directions to improve the proposed method. First, the proposed method relies on a concurrency oracle when transforming sets of traces into sets of partially ordered runs. In the empirical evaluation, we relied on a relatively simple concurrency oracle, namely the $\alpha+$ oracle. This oracle has the limitation that it sometimes cannot isolate concurrency in the presence of short loops (involving 1 or 2 events). Accordingly, another direction for future work is to evaluate the performance of the proposed method with a range of more sophisticated concurrency oracles. A more accurate concurrency oracle can lead to a more accurate transformation from traces to runs, which in turn would lead to an event structure that better reflects the log.

Another direction for improvement is to design techniques for summarizing the difference diagnosis, for example by grouping together related difference statements and abstracting them via higher-level statements that strike a tradeoff between accuracy and interpretability. In a similar vein, another possible improvement is to design techniques to visually overlay the difference statements on top of the input process model, in order to help the user to pinpoint the location and the nature of each difference.

Acknowledgements. We thank Jorge Muñoz-Gama for his support with the ProM plugin for conformance checking, and Boudewijn van Dongen for his insights on trace alignment. NICTA is funded by the Australian Government. This research is funded by the Australian Research Council Discovery Project DP150103356 and the Estonian Research Council.

REFERENCES

[1] W. M. P. van der Aalst, *Process Mining - Discovery, Conformance and Enhancement of Business Processes*. Springer, 2011.

[2] M. Dumas and L. García-Bañuelos, "Process mining reloaded: Event structures as a unified representation of process models and event logs," in *Proc. of PETRI NETS*. Springer, 2015, pp. 33–48.

[3] W. van der Aalst, A. ter Hofstede, B. Kiepuszewski, and A. Barros, "Workflow Patterns," *Distributed and Parallel Databases*, vol. 14, no. 1, pp. 5–51, 2003.

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

[5] A. A. de Medeiros, "Genetic process mining," Ph.D. dissertation, Eindhoven University of Technology, 2006.

[6] S. K. L. M. vanden Broucke, J. Munoz-Gama, J. Carmona, B. Baesens, and J. Vanthienen, "Event-based real-time decomposed conformance analysis," in *Proc. of OTM Conferences*. Springer, 2014, pp. 345–363.

[7] J. Munoz-Gama, J. Carmona, and W. M. P. van der Aalst, "Single-entry single-exit decomposed conformance checking," *Inf. Syst.*, vol. 46, pp. 102–122, 2014.

[8] A. Adriansyah, B. van Dongen, and W. van der Aalst, "Conformance checking using cost-based fitness analysis," in *Proc. of EDOC*. IEEE, 2011, pp. 55–64.

[9] F. Mannhardt, M. de Leoni, H. A. Reijers, and W. M. van der Aalst, "Balanced multi-perspective checking of process conformance," *Computing*, pp. 1–31, 2015.

[10] J. D. Weerdt, M. D. Backer, J. Vanthienen, and B. Baesens, "A robust f-measure for evaluating discovered process models," in *Proceedings of the CIDM 2011*, 2011, pp. 148–155.

[11] S. K. L. M. vanden Broucke, J. D. Weerdt, J. Vanthienen, and B. Baesens, "Determining process model precision and generalization with weighted artificial negative events," *IEEE Trans. Knowl. Data Eng.*, vol. 26, no. 8, pp. 1877–1889, 2014.

[12] J. Munoz-Gama and J. Carmona, "A fresh look at precision in process conformance," in *Proc. of BPM*. Springer, 2010, pp. 211–226.

[13] A. Adriansyah, J. Munoz-Gama, J. Carmona, B. F. van Dongen, and W. M. P. van der Aalst, "Measuring precision of modeled behavior," *Inf. Syst. E-Business Management*, vol. 13, no. 1, pp. 37–67, 2015.

[14] L. García-Bañuelos, N. van Beest, M. Dumas, and M. L. Rosa, "Business process conformance checking based on event structures," in *Proc. of NWPT'2015*. Reykjavik University, 2015, 3 pages.

[15] M. Nielsen, G. D. Plotkin, and G. Winskel, "Petri nets, event structures and domains, part I," *Theor. Comput. Sci.*, vol. 13, pp. 85–108, 1981.

[16] R. M. Dijkman, M. Dumas, and C. Ouyang, "Semantics and analysis of business process models in BPMN," *Information & Software Technology*, vol. 50, no. 12, pp. 1281–1294, 2008.

[17] K. L. McMillan, "A technique of state space search based on unfolding," *Formal Methods in System Design*, vol. 6, no. 1, pp. 45–65, 1995.

[18] J. Engelfriet, "Branching processes of Petri nets," *Acta Informatica*, vol. 28, pp. 575–591, 1991.

[19] A. Armas-Cervantes, P. Baldan, M. Dumas, and L. García-Bañuelos, "Diagnosing behavioral differences between business process models: An approach based on event structures," *Information Systems*, vol. 56, pp. 304–325, 2016.

[20] J. Esparza, "Model checking using net unfoldings," *Sci. Comput. Program.*, vol. 23, no. 2-3, pp. 151–195, 1994.

[21] N. van Beest, M. Dumas, L. García-Bañuelos, and M. L. Rosa, "Log delta analysis: Interpretable differencing of business process event logs," in *Proc. of BPM 2015*. Springer, 2015, pp. 386–405.

[22] A. K. Alves de Medeiros, W. M. P. van der Aalst, and A. J. M. M. Weijters, "Workflow mining: Current status and future directions," in *Proc. of On The Move to Meaningful Internet Systems (OTM) 2003*, 2003, pp. 389–406.

[23] J. E. Cook and A. L. Wolf, "Event-based detection of concurrency," in *FSE*. ACM, 1998, pp. 35–45.

[24] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Trans. Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[25] A. Valmari, "A stubborn attack on state explosion," *Formal Methods in System Design*, vol. 1, no. 4, pp. 297–322, 1992.

[26] J. C. Tiernan, "An Efficient Search Algorithm to Find the Elementary Circuits of a Graph," *Commun. ACM*, vol. 13, no. 12, pp. 722–726, 1970.

[27] J. L. Szwarcfiter and P. E. Lauer, "A search strategy for the elementary cycles of a directed graph," *BIT Numerical Mathematics*, vol. 16, no. 2, pp. 192–204, 1976.

[28] V. Pratt, "Modeling concurrency with partial orders," *International Journal of Parallel Programming*, vol. 15, no. 1, pp. 33–71, 1986.

[29] M. de Leoni and F. Mannhardt, "Road traffic fine management process," 2015. [Online]. Available: http://dx.doi.org/10.4121/uuid:270fd440-1057-4fb9-89a9-b699b47990f5

[30] D. Fahland, C. Favre, J. Koehler, N. Lohmann, H. Völzer, and K. Wolf, "Analysis on demand: Instantaneous soundness checking of industrial business process models," *Data Knowl. Eng.*, vol. 70, no. 5, pp. 448–466, 2011.

[31] T. Curran and G. Keller, *SAP R/3 Business Blueprint: Understanding the Business Process Reference Model*. Upper Saddle River, 1997.

[32] S. Vanden Broucke, J. De Weerdt, J. Vanthienen, and B. Baesens, "An improved process event log artificial negative event generator," Faculty of Economics and Business, KU Leuven (Belgium), Tech. Rep. KBI_1216, 2012.

[33] R. Conforti, M. Dumas, L. García-Bañuelos, and M. La Rosa, "BPMN Miner: Automated discovery of BPMN process models with hierarchical structure," *Information Systems*, vol. 56, pp. 284–303, 2016.

Luciano García-Bañuelos Luciano García-Bañuelos is Associate Professor of Software Engineering at University of Tartu. He obtained his PhD in 2003 from Grenoble Institute of Technology for his work on long-running transactions. His current research interests are in the fields of service-oriented computing and business process management, with a focus is on formal methods for business process modeling and analysis.

Nick van Beest Nick van Beest is researcher at the Software Systems Research Group at NICTA Queensland. He obtained his PhD in Information Systems in 2013 at the University of Groningen, The Netherlands. He is a visiting researcher at the Queensland University of Technology (Australia) and University of Tartu (Estonia). His research experience covers artificial intelligence, process mining, business process compliance and knowledge-intensive business processes. He currently works on deviance mining and conformance checking for the purpose of performance improvement and automated runtime anticipation of disruptions.

Marlon Dumas Marlon Dumas is Professor of Software Engineering and University of Tartu, Estonia and Adjunct Professor of Information Systems at Queensland University of Technology, Australia. His interests span across the fields of software engineering, information systems and business process management. His research focuses on combining data mining and formal methods for analysis and monitoring of business processes. He has published extensively in conferences and journals across the fields of software engineering and information systems and has co-authored two textbooks on business process management.

Marcello La Rosa Marcello La Rosa is Professor of Business Process Management (BPM) and the academic director for corporate programs and partnerships at the Information Systems school of the Queensland University of Technology, Brisbane, Australia. His research interests include process consolidation, mining and automation, in which he published over 80 papers. He leads the Apromore initiative (www.apromore.org) - a strategic collaboration between various universities for the development of an advanced process model repository, and has co-authored a textbook on BPM.