Monitoring Distributed Systems based on Partial Order Executions with Global States*

Moran Omer[1], Doron Peled[1], Ely Porat[1], and Vijay K. Garg[2]

[1] Department of Computer Science Bar Ilan University
[2] Department of Electrical and Computer Engineering, UT Austin

Abstract. Runtime Verification (RV) allows monitoring the behaviors of a system while checking them against a formal specification. The executions of *distributed systems* are often modeled using interleaving semantics, where events of different processes are interleaved into a total order. However, certain behavioral properties are difficult to express using interleaving semantics, whereas they can be naturally expressed in terms of partial order semantics. We study the problem of runtime verification for distributed systems based on the global states structure associated with a partial order execution. We present two algorithms for RV with branching temporal specifications and study the complexity of this problem. The first algorithm is for a global temporal logic with past operators we term PaCTL (for *Past* CTL). It involves constructing the branching structure of global states. We then show a second, more efficient, algorithm, for a subset of this logic that we term PaBTL. This algorithm does not require constructing the branching structure. We present implementations for both algorithms with experimental results.

1 Introduction

Runtime verification (RV) [5, 6, 20] monitors an execution trace consisting of events emitted by the observed system and verifies it against a given formal specification. RV for distributed systems poses a non-trivial challenge, since it depends on combining information related to events that are executed on different processes. In system verification (e.g. RV and model checking), concurrent systems are typically modeled using interleaving semantics, imposing a *total order* between the executed events; occurrences of independently executed events from different processes are *interleaved* in either order in different execution sequences. In contrast, a model that assumes a *partial order* [28, 48] among the events sometimes offers a more direct and intuitive view of executions that can be distributed among different processes. There, events executed independently

* The research performed by the first two authors was partially funded by Israeli Science Foundation grant 2454/23: "Validating and controlling software and hardware systems assisted by machine learning". The research performed by the fourth author was partially funded by the Cullen Trust for Higher Education Endowed Professorship.

by different processes, which can also overlap in time with each other, are unordered; dependencies between events in different processes can result from message passing or the access of variables shared between multiple processes.

The interleaving model is rather simple and enjoys the benefit of using common mathematical tools for verification, e.g., based on finite automata over infinite words [45]. Specification over this model is often given using Linear Temporal Logic (LTL) [25]. Practice shows that for most purposes, the interleaving model is sufficient for modeling concurrent systems as a basis for temporal specification; the fact that concurrently executed events are interleaved is often not restrictive, in particular if the specification is not sensitive to the relative order of such interleavings [37]. On the other hand, there are cases where properties of a distributed system are lost when interleaving their executed events and it is beneficial to use the partial order execution model.

In this paper, we study runtime verification of distributed systems, based on the partial order semantics. The verification is with respect to a temporal logic specification that asserts about the *branching structure* over *global states* related to a partial order execution of the monitored system. The global states in the partial order model correspond to *cuts*, where a cut is a *history closed* subset of events of the partial order. The RV monitoring in our case is centralized, which is in accordance with the global state based specification.

An example of a property that calls for the use of the partial order model is related to the detection of global *snapshots* [9] of a distributed system, i.e. a consistent collection of local states of the system. Such a snapshot corresponds to a cut of the partial order execution, while in the interleaving model such a snapshot may not appear directly as a global state of the modeled interleaving sequence[1]. Another example is from distributed databases, where *transactions*, i.e., pieces of the execution that involve multiple events, are designed to behave *as if* executed one after another in some linearizations [8, 13, 21, 33], while in other linearizations of these events the transactions may (partially) overlap; this allows achieving some concurrency between the events of the transactions, and, on the other hand, simplifying the design, based on the sequential-like behavior. A similar idea can be used for describing properties of concurrent data objects or systems implemented without a centralized control (e.g., based on blockchains).

Contributions. We present a runtime verification algorithm for distributed systems, based on the global states construction over the partial order execution model. The specification formalism that we use is a past time version of the temporal logic we call PaCTL, applied to the branching time structure of global states. This logic contains past operators such as $EP\varphi$ (φ holds sometimes in the past) and $E(\varphi \mathcal{S} \psi)$ (φ holds along some linearization since ψ held). We provide an algorithm for the complete logic, whose worst case complexity is exponential in the number of processes, with the base of the exponent being the number of events, and a corresponding tool called PoET [50]. We provide a related hardness result. We then present a second algorithm, for a subset of this logic, which we

[1] If one groups together all the interleaving sequences that are consistent with the partial order execution as in *Mazurkiewicz traces* [28], there is at least one interleaving on which this global state appears

call PaBTL, confined to the past operators EP (and its dual AH) together with the Boolean operators. We present a corresponding tool called `Kairos` [51]. The complexity of this algorithm is linear in the number of events and quadratic in the number of processes, but is exponential in the size of the property. We show experimental results comparing the two tools.

Related Work. Several logics are interpreted over partial order executions, see, e.g., the survey [34]. The branching time temporal logic POTL [38] includes both future and past branching operators, in the style of the logic CTL [11]. The interpretation is over local states and events; all the possible partial order executions are combined into a *single* structure, as in *event structures* [48]. Some other temporal logics that are interpreted over partial order executions are defined over the *global states* (cuts) rather than directly over the *local states* of the partial order between events; for each partial order execution, a *branching* structure between the global states is separately constructed. In that category of logics, the temporal logic ISTL [23, 36] uses the operators of CTL, applied to each such branching structure. The logic LTrL [43] uses the syntax of linear temporal operators but applies them to branching structures of global states that are constructed from partial order executions. Model checking various subsets of ISTL [36] is studied in [1, 3, 35, 44] and of LTrL in [44]. A logic that uses the LTL constructs over Mazurkiewicz traces and a translation from specifications to automata are presented in [7].

An RV verification algorithm, where the a past time CTL specification is interpreted directly over the *local states* associated with the events of the partial order execution, rather than over the related structure of *global states* as in our work, was described in [4]; thus, the same formula has a completely different interpretation in [4] than in our work. Another past time logic defined over the partial order between events is MTTL [41]; a distributed algorithm (and implementation) was given based on the ability to pass information between processes when accessing shared variables. In [10, 40, 42], procedures are presented for deciding whether a partial order execution satisfies properties that can be written as a *restricted* version of the logic ISTL [1]. These works identify also cases of specifications with lower complexity. Ogale and Garg [32] have proposed a logic called Basic Temporal Logic (BTL), which is also a subset of ISTL; they presented a decision procedure for checking whether a partial order execution satisfies a BTL property based on *predicate slicing*, with time complexity exponential in the size of the formula but polynomial in the size of the computation. Another line of works on verification, related to the partial order model, is aimed at proving that a property holds for a *representative* interleaving of a partial order execution assuming synchronous clocks with a small bounded maximal drift. This limits the number of possible interleavings. This is done for LTL specification based on SMT solving [15] and for a stream-based specification [12].

2 Preliminaries

The Partial Order Model (Local View). In the *partial order execution model* [26, 39, 48], events of disjoint processes may be unordered with respect

to each other, while the events that involve the same process must be totally ordered. Interactions between processes, e.g., events mutual to a pair of processes, which can model synchronous message passing, can induce order between events of different processes. This results in a partial order, i.e., a transitive, antisymmetric and irreflexive relation) between the events, rather than a total order (linearization of the events) as in the more commonly used interleaving model. A *partial order execution* $\mathcal{E} = \langle \mathcal{P}, E, Pr, \prec, \iota, A, L \rangle$, has the following components:

- $\mathcal{P}$ is a finite set of *processes*.
- E is a (finite or infinite) set of *events*.
- $Pr : E \mapsto 2^{\mathcal{P}}$ maps each event to a set of processes that are *involved* in its execution.
- $\prec \subset E \times E$ is a *partial order* relation over E. In addition, $\prec$ is well-founded, i.e., E does not have an infinite decreasing chain of events $e_1 \succ e_2 \succ \ldots$. The relation $\prec$ is the *minimal* partial order such that for each $p \in \mathcal{P}$, the events that involve a process p, i.e., $\{e \in E | p \in Pr(e)\}$ are totally ordered; the transitivity of $\prec$ and the fact that events can involve multiple processes induce further order between events from different processes.
- $\iota \in E$ is the *initial event* of E. It is *minimal* w.r.t. $\prec$ and $Pr(\iota) = \mathcal{P}$.
- $A = \biguplus_{p \in P} A_p$ is a finite set of *propositions*. The set A is partitioned into subsets A_p, one for each process $p \in \mathcal{P}$. Thus, each proposition can represent some property (predicate) *local* to some process.
- $L(e, p) \in 2^{A_p}$ for $p \in Pr(e)$. L maps each event and process that participates in it to a subset of propositions from A_p. This represents the propositions that hold (i.e., are set to *true*) in process p *immediately after* e is executed.

We can denote the labeling $L(e, p)$ also as a minterm, i.e., a conjunction of literals, over the propositions A_p; the propositions in $L(e, p)$ appear non-negated, while the propositions of $A_p \backslash L(e, p)$ appear negated. In a short form for denoting minterms, conjunctions are removed and negated propositions are marked with an overbar, hence $t_1 \wedge \neg t_2$ is denoted as $t_1 \overline{t_2}$. If $Pr(e) \cap Pr(f) = \emptyset$, we say that e and f are *independent*. If both $e \not\prec f$ *and* $e \not\succ f$, we say that e and f are *concurrent*; in this case, e and f are also independent. We say that f is an *immediate successor* of e if $e \prec f$ and there is no g such that $e \prec g \prec f$.

Typically, events involve either only a single process, i.e., *local* to a single process, or a pair of processes, representing *synchronous* or *handshake* communication. The described model can represent handshake communication as in the programming language CSP [19]. The model and subsequently the RV algorithm can be adapted to deal with asynchronous message passing.

Figure 1 shows an execution that contains three processes, p_1, p_2 and p_3, with seven events: $\{\iota, \alpha_1, \alpha_2, \alpha_3, \beta_1, \beta_2, \beta_3\}$. The event α_2 involves both processes p_1 and p_2, while β_2 involves p_2 and p_3. The rest of the events involve only a single process each. We have $A_{p_1} = \{t_1, t_2\}$, $A_{p_2} = \{r_1, r_2\}$ and $A_{p_3} = \{q_1, q_2\}$. Then, $L(\alpha_1, p_1) = \{t_2\}$, $L(\alpha_2, p_1) = \{t_1\}$ and $L(\alpha_2, p_2) = \{r_1\}$, which are represented, correspondingly, by the minterms $\overline{t_1} t_2$, $t_1 \overline{t_2}$ and $r_1 \overline{r_2}$.

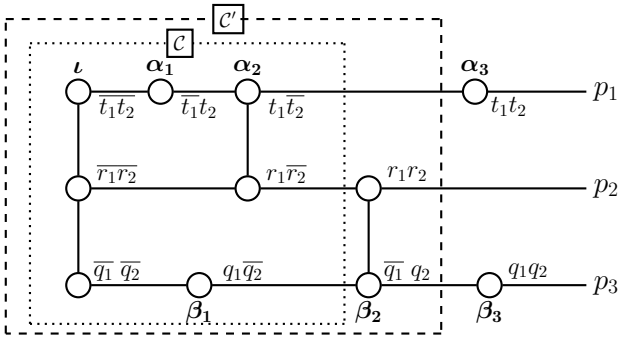

Fig. 1: A partial order execution

The Global View. Based on the local view, we further define the *global view*, which contains *cuts* (and *frontiers*) that correspond to *global states* [26]. These global states form a branching structure.

A *cut* $\mathcal{C}$ of a partial order execution $\mathcal{E}$ is a nonempty (as it always includes the initial event ι) *history-closed* finite subset of its events E. That is, if $f \in \mathcal{C}$ and $e \prec f$ then $e \in \mathcal{C}$. Intuitively, a cut represents a potential *global state* of the modeled or inspected system, where the events in the cut appeared in its past and the other events have not happened yet. The set of cuts of a partial order is closed under unions and intersections.

Denote by $max(\mathcal{C}, p)$ the maximal event in $\mathcal{C}$ of the process p w.r.t. the order $\prec$ (such a maximum exists, since $\iota \in \mathcal{C}$). A *frontier* $\mathcal{F}_\mathcal{C}$ of a cut $\mathcal{C}$ is[2] the set of *maximal* events from $\mathcal{C}$ for the different processes in $\mathcal{P}$ i.e., $\bigcup_{p\in\mathcal{P}} max(\mathcal{C}, p)$. A single event can play the role of a maximal event for *multiple* processes that it involves. We assign to a cut $\mathcal{C}$, or, equivalently to the frontier $\mathcal{F}_\mathcal{C}$, a global interpretation of the propositions A that agrees with the local maximal interpretations of each process in the cut, formally, $L(\mathcal{C}) = \biguplus_{p\in\mathcal{P}} L(max(\mathcal{C}, p), p)$ and $L(\mathcal{F}_\mathcal{C}) = L(\mathcal{C})$. In Figure 1, the marked cut $\mathcal{C}$, whose events are enclosed within an inner *dotted* box, contains the events $\{\iota, \alpha_1, \alpha_2, \beta_1\}$. The corresponding frontier $\mathcal{F}_\mathcal{C}$ is $\{\alpha_2, \beta_1\}$, where α_2 is maximal for both processes p_1 and p_2 and β_1 is maximal for p_3. The global interpretation (state) of the cut $\mathcal{C}$ (and frontier $\mathcal{F}_\mathcal{C}$) is $t_1\overline{t_2}r_1\overline{r_2}q_1\overline{q_2}$.

Note that a pair of events that are *not independent* of each other *may* be maximal w.r.t. different processes, hence can belong to the same frontier. For example, in the cut $\mathcal{C}'$, which appears within the outer *dashed* box in Figure 1, the frontier $\mathcal{F}_{\mathcal{C}'}$ includes both α_2, which is maximal for p_1, and β_2, which is maximal for p_2 and p_3, where $\alpha_2 \prec \beta_2$. The interpretation of frontiers depends for each process on the event maximal for that process, hence, $L(\mathcal{F}_{\mathcal{C}'}) = t_1\bar{t_2}r_1r_2\bar{q_1}q_2$.

We can now define, based on the (local) partial order execution, a corresponding *global* partial order between the cuts and, correspondingly, between the corresponding frontiers of $\mathcal{E}$. Let $\mathcal{C}_1 < \mathcal{C}_2$ if $\mathcal{C}_1 \subset \mathcal{C}_2$ and, correspondingly, $\mathcal{F}_{\mathcal{C}_1} < \mathcal{F}_{\mathcal{C}_2}$. We also denote $\mathcal{C}_1 \twoheadrightarrow \mathcal{C}_2$, or, more informatively, $\mathcal{C}_1 \overset{e}{\twoheadrightarrow} \mathcal{C}_2$ if $\mathcal{C}_2 = \mathcal{C}_1 \cup \{e\}$ for

[2] Denoting the corresponding cut $\mathcal{C}$ as a subscript in $\mathcal{F}_\mathcal{C}$ is optional, and we may simply write $\mathcal{F}$.

some $e \in E$. We say that $\mathcal{C}_2$ is an *immediate successor* of $\mathcal{C}_1$. Accordingly, the corresponding frontier $\mathcal{F}_{\mathcal{C}_2}$ of $\mathcal{C}_2$ is the immediate successor of the frontier $\mathcal{F}_{\mathcal{C}_1}$ of $\mathcal{C}_1$, and we also denote that $\mathcal{F}_{\mathcal{C}_1} \twoheadrightarrow \mathcal{F}_{\mathcal{C}_2}$ (or $\mathcal{F}_{\mathcal{C}_1} \overset{e}{\twoheadrightarrow} \mathcal{F}_{\mathcal{C}_2}$). Hence, the relation $<$ is the transitive closure of the relation $\twoheadrightarrow$. In the example in Figure 1, $\mathcal{F}_{\mathcal{C}} \overset{\beta_2}{\twoheadrightarrow} \mathcal{F}_{\mathcal{C}'}$.

The relation $\twoheadrightarrow$ forms a *branching structure*, over which our specification can be interpreted. The maximal paths in the constructed graph are the *equivalent* linearizations of the (local) partial order execution (see also Mazurkiewicz *trace semantics* [28]). The diagram in Figure 2 represents the global partial order execution obtained from the local partial order execution in Figure 1. Each circle represents a global state, and the filled circle corresponds to the frontier $\mathcal{F}_{\mathcal{C}}$. This is a *Hasse diagram* of the global view, where the depicted edges represent the "immediate successor" relation $\twoheadrightarrow$.

In the branching structure formed from a partial order, if $\mathcal{C} \overset{e}{\twoheadrightarrow} \mathcal{C}_1$ and $\mathcal{C} \overset{f}{\twoheadrightarrow} \mathcal{C}_2$, where $e \neq f$, then e and f are independent; furthermore, we also have $\mathcal{F}_{\mathcal{C}} \overset{e}{\twoheadrightarrow} \mathcal{F}_{\mathcal{C}_1} \overset{f}{\twoheadrightarrow} \mathcal{F}_{\mathcal{C}'}$ and $\mathcal{F}_{\mathcal{C}} \overset{f}{\twoheadrightarrow} \mathcal{F}_{\mathcal{C}_2} \overset{e}{\twoheadrightarrow} \mathcal{F}_{\mathcal{C}'}$. Vice versa, if $\mathcal{F}_{\mathcal{C}} \overset{e}{\twoheadrightarrow} \mathcal{F}_{\mathcal{C}_1} \overset{f}{\twoheadrightarrow} \mathcal{F}_{\mathcal{C}'}$ and $Pr(e) \cap Pr(f)$, then we also have $\mathcal{F}_{\mathcal{C}} \overset{f}{\twoheadrightarrow} \mathcal{F}_{\mathcal{C}_2} \overset{e}{\twoheadrightarrow} \mathcal{F}_{\mathcal{C}'}$; we then say that e and f *commute* with each other from $\mathcal{F}_{\mathcal{C}}$. Similarly, if $\mathcal{F}_{\mathcal{C}_1} \overset{e}{\twoheadrightarrow} \mathcal{F}_{\mathcal{C}}$ and $\mathcal{F}_{\mathcal{C}_2} \overset{f}{\twoheadrightarrow} \mathcal{F}_{\mathcal{C}}$, then e and f are independent.

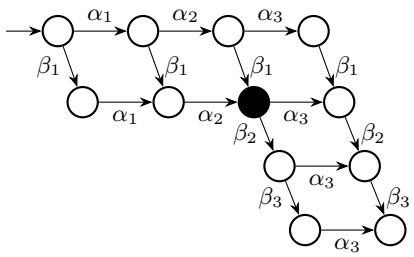

Fig. 2: A Hasse diagram of the global states view of the partial order execution

Collecting Events from the Monitored Processes. During the runtime of a distributed system by a centralized monitor, events from the different processes need to be collected and processed by the monitor. An immediate difficulty is that these events may be reported out of order. For example, in Figure 1, the event α_2 may be reported to the monitor process by process p_1, while the event β_2 may be reported to the monitor by p_3; it can happen that the information about β_2 will be received by the monitor after the information about α_2, although $\alpha_2 \prec \beta_2$. A reported event cannot be processed by the monitor until all events that happened before it according to $\prec$ were also reported; otherwise, the situation will be similar to trying to analyze a linear execution while there are holes in the sequence observed so far.

We assume the absence of a global physical clock, which synchronizes between the events of the monitored system. Instead, we use *logical clocks* [26].

As done in [22], we adopt the use of Fidge and Mattern [14, 27] *vector* clocks, where each event keeps a vector of values, one per each involved process. Com-

paring the order between vector clocks of a pair of events e and f allows to check whether $e \prec f$; furthermore, they also allow one to check if, for a reported event f, there is some $e \prec f$ that is not yet reported to the monitor process. Reported events can be kept in a queue before all their predecessors are reported. Then, they can be processed by the RV algorithm. This also guarantees that when a new event e is processed, the set of events processed so far forms a cut that is an immediate *successor* of the cut that was formed by the set of events processed before, according to the order $\overset{e}{\twoheadrightarrow}$. Consequently, these cuts (and their corresponding frontiers) are formed in an order that is a *linearization* of the monitored partial order. The RV can calculate each time a verdict for the current cut/frontier, which will be reported according to the order of this linearization. This applies to both our algorithms in Section 3 and Section 4.

The Logic and the Interpretation Over the Global View. We use a *past* time branching temporal logic, in the style of CTL [11], to specify properties of the global view of a partial order execution. The restriction to past time allows interpreting the formulas on finite structures. As in CTL, each temporal operator consists of a pair of operators: the first is a *path quantifier*, either A (forall paths) or E (there exists a path). The second operator is temporal; the temporal operators used here are the past mirror images of the corresponding CTL operators: $\mathcal{S}$ (*since*) instead of $\mathcal{U}$ (*until*) and Y (*yesterday*) instead of X (*next time*). We call this logic PaCTL (for *past* CTL). The syntax is as follows, where q is a proposition:

$$\varphi ::= q \mid (\varphi \wedge \varphi) \mid \neg\varphi \mid EY\varphi \mid A(\varphi \mathcal{S} \varphi) \mid E(\varphi \mathcal{S} \varphi)$$

The semantics is as follows, where S, S' represent cuts (or frontiers):

- $S \models q$ if $q \in L(S)$.
- $S \models (\varphi_1 \wedge \varphi_2)$ if $S \models \varphi_1$ and $S \models \varphi_2$.
- $S \models \neg\varphi$ if not $S \models \varphi$.
- $S \models EY\varphi$ if there exists $S' \twoheadrightarrow S$ such that $S' \models \varphi$.
- $S \models E(\varphi \mathcal{S} \psi)$ if either $S \models \psi$ or both $S \models \varphi$ and there exists $S' \twoheadrightarrow S$ such that $S' \models E(\varphi \mathcal{S} \psi)$.
- $S \models A(\varphi \mathcal{S} \psi)$ if either $S \models \psi$ or both $S \models \varphi$ and for each $S' \twoheadrightarrow S$ it holds that $S' \models A(\varphi \mathcal{S} \psi)$, where at least one such predecessor S' exists.

We can also define additional Boolean and temporal operators: $true = (q \vee \neg q)$ for some proposition q, $false = \neg true$, $(\varphi \vee \psi) = \neg(\neg\varphi \wedge \neg\psi)$, $EP\varphi = E(true \mathcal{S} \varphi)$, $AY\varphi = \neg EY \neg\varphi$, $AP\varphi = A(true \mathcal{S} \varphi)$, $EH\varphi = \neg AP\neg\varphi$ and $AH\varphi = \neg EP\neg\varphi$, where P reads as *previously* and H reads as *historically*.

Remark. We show that some properties expressed over the global view cannot be expressed using local properties. The branching logic TLC in [4] is defined directly over the local view. It contains both past and future operators (and in addition to the CTL modalities, an operator || for *concurrently*). It was shown in [4] that the logic TLC can be translated into an automaton, hence represents a regular language. On the other hand, PaCTL can express properties that cannot be expressed as regular languages by comparing (essentially, counting the length of) subsequences of events along independently (concurrently) executing

processes. This can be done by a zigzagging argument, as will be presented in the lower bound proof in Section 3.

3 An RV Algorithm for Global Partial Order Executions

Our RV algorithm is based on a *centralized* monitor, which checks a partial order execution against a branching logic PaCTL specification. Upon processing a new event[3], the algorithm calculates a new verdict, comparing the specification against the set of events collected so far. The sequence of issued verdicts follows a linearization of the collected events. For each processed event α, the monitor obtains the involved processes, i.e., $Pr(\alpha)$, the assignment to their propositions immediately after the execution of α, i.e., $\bigcup_{p \in Pr(\alpha)} L(\alpha, p)$ and the vector clock. For example, the event α_2 in Figure 1 involves the processes p_1 or p_2 and its associated minterm is $t_1 \overline{t_2} r_1 \overline{r_2}$.

The RV algorithm constructs a *subgraph* of the global view; the nodes represent frontiers and the edges correspond to the relation $\twoheadrightarrow$ between the corresponding adjacent frontiers (hence we use the same notation $\twoheadrightarrow$ for edges as we used between frontiers). Each edge is labeled with an event that forms the transition between the corresponding frontiers. With each newly processed event, the graph is updated, adding new nodes and edges. Some old nodes may become redundant, in which case they can be removed. We can consider the constructed graph, obtained after a sequence of processed events, as a *sliding window* into the global view graph, which slides with each new event.

Updating the Sliding Window Graph. During runtime verification, the monitor performs updates, based on the reported events, to the sliding window graph whose nodes are *frontiers* of the partial order execution.

As part of the algorithm, we use the following procedure to construct for a given frontier $\mathcal{F}$ its f successor $\mathcal{F}'$, i.e., $\mathcal{F} \overset{f}{\twoheadrightarrow} \mathcal{F}'$. This successor exists provided that all the immediate predecessors of f according to $\prec$ are in $\mathcal{F}$. This is done as follows: after adding f to the events of $\mathcal{F}$, events of $\mathcal{F}$ that are not any more maximal for *at least* one process, are removed. The order $\prec$ between events can be checked based on the vector clocks attached to the events. For example, in Figure 1, the frontier $\mathcal{F}_{\mathcal{C}} = \{\alpha_2, \beta_1\}$, which corresponds to the cut $\mathcal{C}$, is created after the sequence $\iota\alpha_1\beta_1\alpha_2$ was observed. Then if an event β_2 occurs, the successor frontier $\mathcal{F}_{\mathcal{C}'}$, which corresponds to the cut $\mathcal{C}'$, such that $\mathcal{F}_{\mathcal{C}} \overset{\beta_2}{\twoheadrightarrow} \mathcal{F}_{\mathcal{C}'}$, is constructed as follows: first we add β_2 to $\mathcal{F}_{\mathcal{C}}$ and then we remove from it β_1 since β_1 is not any more maximal for process p_3, as $\beta_1 \prec \beta_2$. Thus, $\mathcal{F}_{\mathcal{C}'} = \{\alpha_2, \beta_2\}$. The event α_2 is *not* removed from $\mathcal{F}_{\mathcal{C}'}$, since it is still maximal w.r.t. the process p_1 even after adding β_2.

We now describe how to update the sliding window graph G. At each point, a node s_m represents the maximal frontier $\mathcal{F}_m$ of G according to the global order $<$. This is the frontier that corresponds to the cut containing all the events

[3] As described in Section 2, we can process a new event only after all the events preceding according to the order $\prec$ were reported to the monitor.

processed so far by the algorithm. Initially, the graph consists of a single node, representing the frontier $\{\iota\}$, where ι is the initial event. With each new observed event, the graph G is transformed, where new nodes are added, and some nodes may also be removed.

When a new event e is added, we first add to G an edge $s_m \overset{e}{\twoheadrightarrow} s_n$, where s_n represents the new maximal frontier $\mathcal{F}_n$ and $\mathcal{F}_m \overset{e}{\twoheadrightarrow} \mathcal{F}_n$. Further, new nodes and edges are added to G as follows. For edges $s \overset{\alpha}{\twoheadrightarrow} s' \overset{\beta}{\twoheadrightarrow} s''$, where α and β are independent ($Pr(\alpha) \cap Pr(\beta) = \emptyset$), β can *propagate backwards* over the edge $s \overset{\alpha}{\twoheadrightarrow} s'$ in the following manner: new edges $s \overset{\beta}{\twoheadrightarrow} r \overset{\alpha}{\twoheadrightarrow} s''$ are added where r is a new node (unless such edges and node already exist from previous updates). Both the β successor of the frontier corresponding to s and the α successor of the frontier corresponding to r exist due to the commutativity between α and β, as described in Section 2. The addition of the node r may induce, through commutativity, further backward propagations. This chain of propagations repeats until no further edges can be added.

After finishing a phase of extending the graph G with new nodes and edges due to observing a new event e, a phase of removing *redundant* nodes (and all of their connected edges) from G starts. A node s becomes redundant when the occurrence of future events cannot generate further successors to s, hence it can no longer affect the RV verdict. A sufficient condition for this is that for each process $p \in \mathcal{P}$, the algorithm has already processed an event α involving p (i.e., $p \in Pr(\alpha)$), that generates a successor to s. To implement this, for each state s in G, we accumulate the processes involved in generating its successors in the set R_s, which is initialized to the empty set when the node s is generated. We remove s when $R_s = \mathcal{P}$. Furthermore, if there is an edge $s \overset{\alpha}{\twoheadrightarrow} s'$ and the node s' has become redundant and was removed, then s can also be removed. The reason for this is that no *new* edge $s \overset{\beta}{\twoheadrightarrow} s''$ can be further added to the graph. To see this, recall that if there is a β successor to the frontier corresponding to s, then α and β are independent. Hence there is a β successor to the frontier associated with s', thus an edge labeled with β emanates from s'. Since s' was redundant, the β successor of s' was already constructed before s' was removed, by induction on the order of removing redundant nodes and edges. Moreover, by the construction of the sliding window, the edge labeled β from s' has already been propagated backwards over $s \overset{\alpha}{\twoheadrightarrow} s'$. Thus, an edge $s \overset{\beta}{\twoheadrightarrow} s''$ has already been constructed before s' was removed.

We demonstrate the first few steps of the construction windows for the observation $\sigma = \iota\alpha_1\alpha_2\beta_1$, which is a prefix of the linearization $\sigma = \iota\alpha_1\alpha_2\beta_1\alpha_3\beta_2\beta_3$ of the partial order execution in Figure 1. The steps are denoted as **A**-**G**. In every step, the node corresponding to the maximal frontier is shaded. New edges, added due to backward propagation (in steps **E** and **F**) appear dashed. The backward propagation in Step **E** causes another backward propagation that appears in Step **F**. The two nodes in Step **F** with dotted border become redundant and are removed, with their corresponding edges, resulting in Step **G**.

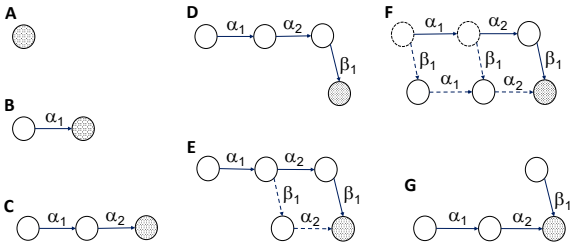

Fig. 3: Graphs constructed for the trace $\sigma = \alpha_1\alpha_2\beta_1\alpha_3$.

Calculating the Values of Subformulas on the Nodes. The following procedure calculates the truth value for each subformula for each new frontier (global state) constructed by the sliding window graph-updating algorithm. This can be compared to updating the *summary vector* used in the case of past time LTL [20]. For each subformula η of the specification φ and each node representing a frontier $\mathcal{F}$ we keep a bit $val(\mathcal{F}, \eta)$ that is *true* if η holds in $\mathcal{F}$. Calculating the truth value for a subformula η may depend on the values calculated subformulas within η; hence the calculation of the truth values progresses *bottom up* according to the syntax tree of φ. Further, the calculations also use the values of truth values previously calculated for the *predecessor* frontiers according to the sliding window graph. The calculation of the truth values is performed as follows.

- $val(\mathcal{F}, q) = true$ for a proposition q, iff $q \in L(\mathcal{F})$.
- $val(\mathcal{F}, \neg\eta) = true$ iff $val(\mathcal{F}, \eta) = false$.
- $val(\mathcal{F}, (\eta_1 \wedge \eta_2)) = true$ iff $val(\mathcal{F}, \eta_1) = true$ and $val(\mathcal{F}, \eta_2) = true$.
- $val(\mathcal{F}, EY\eta) = true$ iff there exists a predecessor frontier $\mathcal{F}'$ of $\mathcal{F}$ such that $val(\mathcal{F}', \eta) = true$.
- $val(\mathcal{F}, E(\eta_1 \mathcal{S} \eta_2)) = true$ iff either $val(\mathcal{F}, \eta_2) = true$ or both $val(\mathcal{F}, \eta_1) = true$ and there exists a predecessor frontier $\mathcal{F}'$ of $\mathcal{F}$ in the graph such that $val(\mathcal{F}', E(\eta_1 \mathcal{S} \eta_2)) = true$.
- $val(\mathcal{F}, A(\eta_1 S \eta_2)) = true$ iff either $val(\mathcal{F}, \eta_2) = true$ or both $val(\mathcal{F}, \eta_1) = true$ and for each predecessor frontier $\mathcal{F}'$ of $\mathcal{F}$, (where there must be at least one such predecessor) in the graph it holds that $val(\mathcal{F}', A(\eta_1 S \eta_2)) = true$.

The RV verdict for the specification φ over the processed events is $val(\mathcal{F}_m, \varphi)$.

Complexity. The overall number of constructed global states is $O((|E|/k)^k)$, where $|E|$ is the overall number of events and $k = |\mathcal{P}|$ (i.e., the number of processes). This worst case bound occurs when the events are distributed evenly ($|E|/k$ per process) and all the events of different processes are independent, maximizing the number of global state combinations. Constructing a new frontier from its predecessor is done in time $O(k)$. Calculating the vector of Boolean values for subformulas related to a frontier is done in time $O(k\,|\varphi|)$. This gives a complexity of $O(k\,|\varphi|\,(|E|/k)^k)$, which is exponential in the number of processes with a base proportional to the number of events, and linear in the size of the property.

Runtime verification can be performed online or offline. For online verification, an important complexity measure is the *incremental time complexity*, which measures the computation performed after each new event that is monitored. This is a critical measure if a verdict needs to be given as soon as possible, based on the prefix seen so far. Unfortunately, the incremental complexity of the RV algorithm presented here is still very hard. In particular, after each newly observed event, the number of nodes (frontiers) added to G can be $O((|E|/k)^{k-1})$, where the local state of the new event can be combined with all the local states of the other (independent) events.

It should be noted that such a centralized setting makes the speed of the monitor process a bottleneck for the RV process, as it needs to process the events from all the participating processes. The global nature of the specification formalism does not easily lend itself to an efficient distributed RV algorithm that may be implemented on the monitored processes themselves (which can be the subject of further research). This, and the complexity results described, may in fact limit the *online* application of runtime verification in some cases.

Hardness of RV Problem. We present a hardness result for the RV problem of making a verdict for the logic PaCTL over a partial order execution. The overall complexity of the described RV algorithm is $O((|E|/k)^k)$, when ignoring the linear factors involving updating each frontier. We employ a reduction from a fine-grained complexity problem that does not belong to the standard complexity hierarchy, such as P, NP, or PSPACE. Fine-grained complexity [46] is a rapidly growing area of research that seeks to establish tight computational bounds for specific problems by exploring their precise relationships. This field examines a variety of foundational problems that are considered computationally hard. This approach allows us to capture the complexity in terms of *two* parameters: the number of processes and the total number of events. We show that, under some known complexity assumption, when the length of the formula is linear in the number of processes, this is also the lower bound up to a poly-logarithmic factor in the base, i.e. in the number of events. Specifically, we establish a lower bound that shows that the complexity is not only exponential in k, but that the *base* of the exponent grows with $\Omega(\frac{|E|}{k \log^2 \frac{|E|}{k}})$, rather than remaining a fixed constant such as 2 or 3.

We will describe a fine-grained reduction from the k-OV framework [47], which connects the complexity of our target problem to this hypothesis. The Boolean vectors $a_1, a_2, \ldots, a_k$ are *orthogonal* if no bit position contains a 1 in all vectors simultaneously. Formally, this condition holds if $\bigvee_{i=1}^{d} \bigwedge_{j=1}^{k} a_j[i] = 0$, where $a_j[i]$ denotes the i-th bit of vector a_j, $\wedge$ represents bitwise conjunction (logical *and*), and $\vee$ represents disjunction (logical *or*) over the bit positions. In the *k-OV* problem, there are k sets $E_1, E_2, \ldots, E_k$, each contains n d-bit vectors, where $d = O(\log^2 n)$. One needs to decide whether there exists a selection of vectors $a_1 \in E_1, a_2 \in E_2, \ldots, a_k \in E_k$ such that the chosen vectors are orthogonal. A widely believed conjecture in complexity theory [47] states that the *k-OV* problem requires time at least $\Omega\left(n^{k-\epsilon}\right)$ for any constant $\epsilon > 0$.

We describe an encoding of the sets of vector as processes and a temporal logic formula that implements the orthogonality constraint using local propositions. Each set E_i of vectors is encoded as a sequence of $|E_i| = O(n \log^2 n)$ events, which is $1/k$ of the total number of events $|E| = \sum_{i=1}^{k} |E_i| = O(k\, n \log^2 n)$. The vectors are encoded one after the other, separated by a delimiter as will be explained below. Therefore in our case $n = \Omega(\frac{|E|}{k \log^2 \frac{|E|}{k}})$ and we get a lower bound of $\Omega\left(\left(\frac{|E|}{k \log^2 \frac{|E|}{k}}\right)^{k-\epsilon}\right)$. As stated above, the base of the exponent is within a polylogarithmic factor of the upper bound $O((|E|/k)^k)$.

- **Vector encoding:** Each bit vector is encoded as a sequence of events within its corresponding process. For example, to represent a bit vector 101 in process p_i, we generate three consecutive events where proposition v_i holds *true*, *false*, and *true*, respectively. This proposition v_i reflects the current bit value represented at each sequence position. All bit vectors are assumed to have uniform length d across all processes, though this is a global convention not explicitly enforced by our formula.
- **Vector separation:** We introduce a delimiter proposition δ_i for each process p_i to mark boundaries between consecutive vector encodings. When $\delta_i = 1$, the current event represents a transition between vectors.
- **Synchronization mechanism:** A trinary counter l_i is associated with each process.[4] This counter cycles through the values 0, 1, and 2 across successive non-delimiter events within the same process and can be implemented using two bits. To align delimiters with $l_i = 0$, we pad each vector with $v_i = 0$ until its length is a multiple of 3.

The formulas is encoded as follows. The structural constraints described above are enforced when constructing each single process p_i from a given set of vectors E_i. We express the orthogonality condition in PaCTL using a formula of the form $EP(\psi \wedge EY\, E(\varphi\, \mathcal{S}\, \psi))$ of size $O(k)$, where:

1. ψ identifies global states where all processes are at delimiter events, expressed as $\bigwedge_{1 \le i \le k} \delta_i$.
2. The formula φ constrains the path satisfying $(\varphi\, \mathcal{S}\, \psi)$ to follow a specific *zigzagging* pattern: either all processes hold the same value, or there exists an index i such that $1 \le i < k$, where processes 1 through i hold some value $v \in \{0, 1, 2\}$, and processes $i + 1$ through k hold the value $(v + 1) \bmod 3$.
   To encode this, we proceed in two steps. First, we restrict the state so that all processes use up to two of the three possible values. Second, we enforce that for each adjacent pair of processes, the value of l_{i+1} is either equal to l_i or to $(l_i + 1) \bmod 3$. Both conditions can be encoded using a formula of size $O(k)$.
3. Further, φ asserts that when all $\delta_i = 0$ and all the counters l_i have identical values, at least one process must satisfy $v_i = 0$. This enforces the core orthogonality condition by ensuring the bitwise conjunction across all processes equals zero.

[4] The use of bit patterns to compare events across different processes appears in [44].

4 An Efficient Algorithm for a Subset of the Logic

The logic PaBTL (for *Past Basic Temporal Logic*) is obtained by restricting PaCTL to the operators EP and AH (where $AH\varphi = \neg EP\neg\varphi$) and the Boolean operators ($\wedge$, $\vee$, $\neg$). This restriction enables a more efficient algorithm while still capturing many practical properties. Several works suggested RV algorithms for logics that use *future* versions of the temporal operators [10, 32, 40, 42]; so, instead of the EP operator in PaCTL one has its future mirror EF ("sometimes in the future for some path") and instead of the AH operator, one has AG ("for every state in all paths in the future"). A model checking algorithm for a logic with the future versions of the temporal operators is described in [1], based on translating conditions on linearizations of the partial order into an automaton.

Converting to normal form. A *disjunctionless normal form* or DLNF formula is defined as follows:

- Atomic propositions.
- $EP(\varphi_1 \wedge \ldots \wedge \varphi_n)$, if $\varphi_1 \ldots \varphi_n$ are in DLNF.
- If φ is in normal form, then so is $\neg\varphi$.

Our algorithm starts by translating the PaBTL specification into a DNF (Disjunctive Normal Form) combination of DLNF subformulas; while DLNF formulas have no disjunctions internally, we convert the entire PaBTL specification into a DNF of DLNF formulas, hence allowing disjunction at the top level only. The translation follows [1], where a similar translation is applied to the logic BTL. The translation is based on the equivalence $EP(\varphi \vee \psi) = (EP\varphi \vee EP\psi)$. To increase efficiency of the algorithm, the translation can rewrite the formulas into a DAG rather than in linear form (as in text) . Each node of the DAG contains an operator (Boolean or EP), open or close parenthesis or a proposition. Traversing each subgraph in depth-first-search order, left to right, corresponds to a subformula. Subformulas that are *repeated* under the translation can be represented *once* (hence the *DAG* structure, rather than an abstract syntax *tree* representation). The translation can result in an exponential explosion in the size of the specification [1].

Detecting a frontier that satisfies a conjunction of minterms. Our algorithm for RV of PaBTL properties first employs a procedure for *detecting* a frontier F that satisfies a conjunction of minterms $\eta = \bigwedge_{1\leq i\leq n} \gamma_i$, where each minterm γ_i contains only propositions from A_{p_i}. Further, F is the minimum such frontier that satisfies $P \leq F$ for some given frontier P. This procedure modifies a procedure presented in [30] for finding a frontier satisfying a minterm: on top of which we added the condition $P \leq F$ and on-the-fly processing, which performs updates based on the arrival of new events.

We now show that if such a frontier F exists, there is a minimum one with respect to the order $<$. It is sufficient to show that if two frontiers F and F' satisfy the above conditions, then the frontier that is obtained by taking the intersection of their corresponding cuts C_F and $C_{F'}$ has a frontier Q (hence $Q \leq F$ and $Q \leq F'$) satisfying η. We denote the maximal event involving p_i in F by F_i. For each process p_i, let $Q_i = \min_{\prec}(F_i, F'_i)$. Then Q_i must be in the cut $C_F \cap C_{F'}$. In addition, Q_i must be maximal with respect to $\prec$ among events

$C_F \cap C_{F'}$ that involve p_i; this is because by the selection of Q_i, it is impossible that both C_F and $C_{F'}$ have a mutual event (thus, in their intersection) involving p_i that is bigger than Q_i. Hence Q forms the frontier of $C_F \cap C_{F'}$. Since Q_i is either F_i or F'_i, it satisfies γ_i. Consequently, Q satisfies η.

Our procedure uses a vector M to store events, such that $M[i]$ is an event of p_i. Since events may involve multiple processes, it is possible that different components of M may represent the same event. We initialize M to the events in P. When a new event α is processed, the procedure is called to perform updates to M as detailed below. When no more updates are available, the value of the vector M is kept for the next call of the procedure. When for each i, $M[i] \models \gamma_i$ then M forms a frontier satisfying η and it is not updated further.

With a new event α, we check if $M[i] \models \gamma_i$ for each process $p_i \in Pr(\alpha)$. If this is the case, there is no need to update M until the next event is processed. Otherwise, we perform *component correction*: for each $p_i \in Pr(\alpha)$ we set $M[i]$ to α. As a consequence, M may not form a frontier anymore. In this case, some components of M need to be updated in order to advance M to the frontier of the minimum cut that contains the current events in M. We call this *frontier update*: each time some element $M[i]$ is updated, we may need to progress an event in other components $M[j]$, for some $j \neq i$. This happens when there is an immediate successor β of $M[j]$ such that $\beta \preceq M[i]$ (because in this case, $M[j]$ cannot be the maximal element involving p_j in a frontier that contains also $M[i]$). In this case we advance $M[j]$ to β. This update may cause a chain of similar updates to other elements of M. When no further such an update is possible, M has become a frontier again.

In the process of frontier update of M, we may have changed some component $M[l]$ such that previously $M[l] \models \gamma_l$ but not anymore. In this case, we need to check if we can perform further component corrections, progressing $M[l]$ to some successor ν. We can pick as ν either the minimal observed event involving p_l that satisfies γ_l and is bigger than $M[l]$, or, if no such event exists, the maximal observed event involving p_l. If such progress is possible, then frontier update may become necessary again. An induction on the order of the updates to M shows that whenever M forms a frontier, there is no frontier F that satisfies $P \leq F < M$ such that $F \models \eta$.

The process of alternately performing component corrections and frontier updates repeats until one of the following two cases occur: (1) for each i, $M[i]$ satisfies γ_i; then $M \models \eta$ and we are done, or (2) all the components of $M[l]$ such that $M[l] \not\models \gamma_l$ are maximal among the events involving p_l w.r.t. $\prec$. In the latter case, a further call to the procedure, upon processing the next event, will advance M further towards achieving a frontier satisfying η.

An RV algorithm for DLNF formulas. The following algorithm is applied to each subformula of the form $EP\varphi$ in DLNF, including those nested within other $EP\varphi$ subformulas. The algorithm for different such subformulas is not applied consecutively, but is first initiated for all such subformulas and then updates are performed each time a new event is added for processing to the partial order. The updates for $EP\varphi$ must follow the updates for all the subformulas in φ. For each

Step	Types	M Initialization / fixing	Success Condition
(1)	$\mathbb{P}$ only	Fix $M \triangleq P$ after (a) holds.	(a)
(2)	$\mathbb{P}+\mathbb{M}$	Init $M = P$ after (a) holds.	(a) and (b)
(3)	$\mathbb{P}+\mathbb{M}+\mathbb{N}$	Init $M = P$ after (a) holds.	(a) and (b) and (c)
(4)	$\mathbb{P}+\mathbb{N}$	Fix $M \triangleq P$.	(a) and (c)
(5)	$\mathbb{M}+\mathbb{N}$	Init M as initial event ι.	(b) and (c)
(6)	$\mathbb{N}$ only	Fix M as the initial event ι.	(d)
(7)	$\mathbb{M}$ only	Init M as initial event ι.	(b)

(a) All $EP\varphi_i$ $(1 \leq i < k)$ already hold.
(b) The minterm η_k is satisfied for detected frontier M.
(c) For each $EP\psi_l$ $(k \leq l \leq n)$ that already holds, not $N_l \leq M$ for detected frontier M.
(d) ι is not yet satisfying $EP\psi_l$ for each $k < l \leq n$.

Table 1: Monitoring Algorithm Steps and Conditions

such subformula $EP\varphi$, where, according to the DLNF form, $\varphi = \eta_1 \wedge \ldots \wedge \eta_n$, we rearrange the conjuncts η_i in the formula according to the following order:

- $\mathbb{P}$ For $1 \leq j < k$, $\eta_j = EP\varphi_j$. These are the *positive* conjuncts. We denote the minimum frontier satisfying η_j by P_j.
- $\mathbb{M}$ Let η_k be a conjunction of literals, i.e., a *minterm*, collected together (i.e., we do not consider each literal in the conjunction separately).
- $\mathbb{N}$ For $k < l \leq n$, $\eta_l = \neg EP\varphi_l$. These are the *negative* conjuncts. We denote the minimum frontier satisfying $EP\varphi_l$ by N_l.

Not all the above three types of components have to exist in $EP\varphi$. For each such $EP\varphi$ subformula, we keep a *separate* vector M that is used to calculate the minimum frontier that satisfies $EP\varphi$. (To simplify the presentation, we denoted M and P rather than M_φ and P_φ per each DLNF subformula.) The vector M is updated by calls to the procedure described above, synchronized the calls with the processing of new events. We also keep an indication of whether $EP\varphi$ was found to already hold ("success condition"). Otherwise, the truth value of $EP\varphi$ is *false*. Note that $EP\varphi$ is *stable*, i.e., when it holds, it will continue to hold when new events are added to the partial order. The evaluation of $EP\varphi$ depends on the components $\mathbb{P}$, $\mathbb{M}$ and $\mathbb{N}$ that are included in φ. At least one such component must exist; hence there are seven cases, detailed in Table 1, where the included types of components are listed in the second column.

The third column for each row of the table specifies the initialization of the corresponding vector M. We distinguish in the table between the case where M is *fixed* upon initialization, or may be achieved later than the time it is initialized, after the occurrence of further events. Let P_j refer to the minimum frontier that satisfies the subformula $EP\varphi_j$, for $1 \leq j < l$, that appears positively (i.e., of the form $\mathbb{P}$ *within* $EP\varphi$). For initializations, we need to calculate the minimum

frontier P such that $P_j \leq P$. In lattice theory, P is the *least upper bound* with respect to $\leq$ among the set of frontiers P_i, denoted $P = \bigsqcup_{1 \leq j < k} P_j$. The frontier P can be calculated as follows: for each process p_i, $P[i]$ is the maximal event involving the process p_i among the different frontiers P_j. (This follows from the fact that taking the set of cuts C_j for which P_j is a frontier, we have that P is the frontier of $C = \bigcup_{1 \leq j < k} C_j$). The initialization (including fixing) of M to the frontier $P = \bigsqcup_{1 \leq j < k} P_j$ takes place when all components of the form P_j required to calculate P were detected. The fourth column specifies conditions under which the subformula $EP\varphi$ (stably) holds, based on conditions (a), (b), (c) and (d), which are defined at the bottom of the table.

After a vector of the form M for a subformula of type $EP\varphi$ was initialized (but *not* fixed), if it does not already satisfy the subformula η_k of φ, then M is updated upon processing a new event added to the partial order. This is done according to the procedure described at the beginning of this section for detecting a frontier satisfying η_k; each γ_i in that procedure corresponds to the part of the minterm η_k that consists of variables of the process p_i. Updating M can affect conditions (a)–(d). One can optimize the algorithm by removing events that cannot contribute further to the verdict: an event α can be removed if *for each* frontier M calculated according to Table 1 for some subformula of the form $EF\varphi$, either M is already detected, or it contains an event β such that $\alpha \prec \beta$.

We now explain in some detail the case (3) in the table, which is the most involved. The subformula $EP\varphi$ requires that (a) for each of its immediate subformulas of the form $EP\varphi_i$ of φ (i.e., of type $\mathbb{P}$), we have already found a minimum frontier P_i satisfying it. In addition, it requires in (b) that we have found a frontier M that satisfies η_k (type $\mathbb{M}$) satisfying $P_i \leq M$. Condition (b) is enforced by initializing M to P, calculated as explained above. Finally, it requires in (c) that if for some $EP\psi_l$ subformula of φ (i.e., of type $\mathbb{N}$) we have already found a satisfying frontier N_l, then this frontier must not satisfy $N_l \leq M$. Condition (c) refers only to the frontiers N_l that were detected when M that satisfies η_k was already detected; a frontier N_l that will be detected after M is detected will not satisfy $N_l \leq M$.

To complete the verdict of the verified specification, recall that it was translated into a DNF combination of DLNF subformulas. We apply the Boolean operators as appearing in the DNF to the external level subformulas of the form $EP\varphi$ (i.e., those that are not proper subformulas of $EP\varphi$ subformulas).

The overall complexity of the algorithm is $O(|E|\, k^2\, 2^{|\varphi|})$. where $|E|$ is the number of events, k is the number of processes, and $|\varphi|$ is the size of the specification. The problem of detecting a frontier that satisfies a Boolean formula φ was shown in [32] to be NP-Complete using a reduction from SAT. The reduction constructs a set of processes, one per each variable of φ. Each process consists of two events, independent of the events of all other processes. The truth value assigned to the propositional variable associated with a process is set to *true* for one of these events and to *false* for the other. This reduction can be trivially adapted to PaBTL by setting the verified property to $EP\varphi$.

Extending The Algorithm with EY is hard

We saw an efficient algorithm for RV of properties that include only the EP operators (and by using Boolean operators, its dual AH). It is natural to try and extend the RV procedure to include the operator EY. Unfortunately, there is no efficient way to do this unless P=NP. We show that by a reduction from the NP-complete problem 3SAT to RV of a DLNF formula. The DLNF formula used is linear in the original formula, and does not contain disjuncts, hence is already in normal form. Recall that the complexity of our procedure was *polynomial* in all parameters for a DLNF formula, although a general PaBTL formula can be equivalent to an exponentially longer CNF of DLNFs (hence, the complexity is in general exponential in the size of the property for BTL properties).

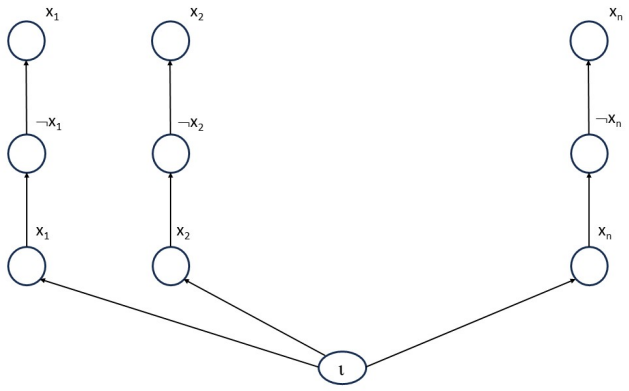

Fig. 4: An execution constructed for the reduction

Given a 3SAT formula φ, e.g., $(x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_3)$, we construct a formula $\beta(\varphi)$ as follows. Each conjunct of the form $(\gamma_j x_j \vee \gamma_k x_k \vee \gamma_l x_l)$ where the γ's are either empty or $\neg$ and x's are variables, i.e., $\gamma_j x_k$ is a *literal*, is translated into conjunct of the form $EYEY(\gamma_j x_j \wedge \gamma_k x_k \wedge \gamma_l x_l)$. Then, all of the formed conjuncts are surrounded by the EP operator. The example formula becomes $EP(EYEY(x_1 \wedge \neg x_2 \wedge x_3) \wedge EYEY(\neg x_1 \wedge \neg x_2 \wedge x_3))$. We also construct a partial order execution, with a process p_i for each variable x_i. Each process has three local events, labeled consecutively x_i, $\neg x_i$ and x_i. See Figure 4.

We show that the original formula φ is satisfiable iff $\beta(\varphi)$ is satisfied at the end of the partial order execution. If φ is satisfied, then select the maximum cut in the execution that corresponds to this assignment. Each conjunct in φ has at least one disjunct literal satisfied by the assignment. In $\beta(\varphi)$ the disjunction was converted into a conjunction. So, we have one literal guaranteed to be satisfied in the corresponding conjunction and we need to make sure that the two other literals are satisfied. In our example, suppose that $\neg x_3$ is satisfied from the first conjunct of φ, which is $(x_1 \vee \neg x_2 \vee x_3)$ and was converted into $EYEY(x_1 \wedge \neg x_2 \wedge x_3)$, Then we have to take care that x_1 and $\neg x_2$ are also satisfied. We can

guarantee this by backing up 0-2 events, depending how many literals are not satisfied, using $EYEY$. Since we do not necessarily need to use the two backward steps, we can perform them on a variable not participating in the corresponding conjunct of φ; if $n \not> 3$, we can add for this purpose a process p_0.

Conversely, suppose $\beta(\varphi)$ is satisfied at the end of the execution. Let C be the cut that satisfies the conjunction within the EP operator (over the $EYEY$ subformulas). Since $\beta(\varphi)$ is satisfied, each subformula η of the form $EYEY\eta$ is satisfied. But becase with $EYEY$ we deviated at most two events back from C, at least one of the variables in η shares the assignment with the cut C. This guarantees that each conjunct in φ is satisfied by the assignment to C.

5 Implementations and Experiments

We developed two runtime verification tools implementing the monitoring algorithm presented in this paper. PoET [50] implements the complete PaCTL algorithm described in Section 3, supporting the full branching temporal logic with past operators including complex nesting and arbitrary formula structures. `Kairos` [51] (from the ancient Greek concept of opportune time) implements the PaBTL algorithm from Section 4, supporting the restricted subset of PaCTL limited to EP operators and Boolean connectives. This tool achieves complexity that is linear in the size of the partial order execution and quadratic in the number of processes, but exponential in the size of the formula due to DLNF transformation.

We conducted comparative performance evaluation to assess the efficiency and scalability of both monitoring approaches across diverse temporal logic patterns, using an Apple MacBook Pro (M1, 16 GB RAM, macOS Sequoia). We evaluated four representative PaBTL properties (Figure 5), covering different cases, with results shown in Table 2. For each property evaluation, we generated four distinct trace files (1K–500K events per trace) with 3–6 concurrent processes. The generated traces used in our experiments are available as part of the PoET and `Kairos` GitHub repositories [50, 51].

1. $EP(status\_ok \wedge load\_lt\_100 \wedge \neg critical\_alarm)$
2. $EP(EP(a) \wedge EP(b) \wedge EP(c) \wedge \neg EP(d))$
3. $EP((aX \wedge EP(pX)) \vee (aY \wedge EP(pY)))$
4. $EP((EP(s1) \wedge \neg EP(j1)) \vee (EP(j2) \wedge ms \wedge \neg EP(s2)))$

Fig. 5: Properties in the PaBTL formalism.

We now present further experiments that demonstrate the full expressiveness of the PoET tool *beyond* the PaBTL-compatible.

The following four properties in Figure 6 require operators beyond the PaBTL subset (e.g., AY, AS, ES) and showcase scenarios where complete PaCTL expressiveness is useful. Table 3 presents results for properties 5-8 across trace sizes from 50 to 1000 events. Due to the exponential complexity of the algorithm, traces were limited to 1K events maximum.

Property	Tool	Parameters	Trace 1K	Trace 10K	Trace 100K	Trace 500K
1	Kairos	Time	0.12s	0.29s	2.64s	13.23s
		Memory	18MB	27MB	126MB	557MB
	PoET	Time	0.59s	5.55s	1032.77s	*
		Memory	40MB	85MB	652MB	
2	Kairos	Time	0.12s	0.37s	3.55s	19.39s
		Memory	19MB	33MB	177MB	803MB
	PoET	Time	5.90s	*	*	*
		Memory	190MB			
3	Kairos	Time	0.09s	0.33s	3.22s	13.93s
		Memory	19MB	29MB	143MB	650MB
	PoET	Time	20.03s	1941.84s	*	*
		Memory	237MB	1.52GB		
4	Kairos	Time	0.10s	0.47s	4.62s	25.45s
		Memory	19MB	33MB	190MB	882MB
	PoET	Time	*	*	*	*
		Memory				

Table 2: Experimental Results: Performance Comparison ($*$ means $>$ 1 hour)

5. $EH((s_{p_1} \rightarrow AY(A(\neg s_{p_2}\ S\ e_{p_2}))) \vee (s_{p_2} \rightarrow AY(A(\neg s_{p_1} S e_{p_1}))))$
6. $E((EH((((a \leftrightarrow a^{'}) \wedge (b \leftrightarrow b^{'})) \wedge ((t_1 \leftrightarrow t_1^{'}) \wedge (t_2 \leftrightarrow t_2^{'}))) \vee$
   $((t_2 \leftrightarrow \neg t_2^{'}) \wedge ((t_1 \leftrightarrow \neg t_2) \leftrightarrow \neg t_1^{'})))\ S\ init)$
7. $EH(COM \rightarrow (AH(c_{p_3} \rightarrow (EP((c_{p_1} \vee c_{p_2}) \wedge EY(COM))))))$
8. $EH(s_{p_1} \rightarrow AH(s_{p_1} \rightarrow EP(e_{p_1} \wedge EYsr_{p_1}))) \vee$
   $EH(s_{p_2} \rightarrow AH(s_{p_2} \rightarrow EP(e_{p_2} \wedge EYsr_{p_2})))$

Fig. 6: PaCTL Properties demonstrating full temporal logic expressiveness

Property	Metric	Trace 50	Trace 100	Trace 500	Trace 1000
5	Time	0.44s	0.41s	18.91s	204.02s
	Memory	34MB	37MB	126MB	328MB
6	Time	0.38s	0.57s	23.64s	230.27s
	Memory	34MB	38MB	134MB	430MB
7	Time	0.35s	0.51s	21.78s	233.33s
	Memory	34MB	39MB	138MB	474MB
8	Time	0.34s	0.23s	0.48s	0.80s
	Memory	34MB	35MB	38MB	49MB

Table 3: PoET Experimental Results for PaCTL Properties

6 Conclusions

We studied RV for partial order executions. Specifically, we used as a specification formalism a past time branching time temporal logic PaCTL, interpreted over the partial order structure between frontiers/cuts obtained from a partial order execution. We presented a runtime verification algorithm with complexity

that grows exponentially in the number of processes with a base proportional to the number of events. We implemented this algorithm in a tool named PoET.

We also presented an algorithm for PaBTL, a subset of PaCTL restricted to temporal operator EP and Boolean operators (including $AH\varphi = \neg EP\neg\varphi$). This algorithm has linear complexity in events, quadratic in processes, and exponential in specification size. We implemented this in a tool named `Kairos`.

Experimental comparison shows that `Kairos` significantly outperforms PoET in both time and memory. While PoET becomes infeasible for larger traces, `Kairos` maintains reasonable performance up to 500K events. This conforms well with the complexity results of the two algorithms. On the other hand, the PoET allows more expressive specifications, including the ES, AS and EY operators.

References

1. R. Alur, K. McMillan, D. Peled, Deciding Global Partial-Order Properties, Formal Methods in System Design 26(1), 2005, 7-25.
2. R. Alur, D. A. Peled, W. Penczek, Model-Checking of Causality Properties, LICS 1995, San Diego, CA, 90-100.
3. R. Alur, D. Peled, Undecidability of Partial Order Logics. Information Processing Letters 69(3): 137-143 (1999).
4. G. Audrito, F. Damiani, V. Stolz, G. Torta, M. Viroli, Distributed runtime verification by past-CTL and the field calculus. Journal of Systems and Software 187, 111251 (2022).
5. E. Bartocci, Y. Falcone, A. Francalanza, M. Leucker, G. Reger, Introduction to runtime verification, lectures on runtime verification - introductory and advanced topics, Lecture Notes in Computer Science 10457, Springer-Verlag, 1-23, 2018.
6. A. Bauer, M. Leucker, C. Schallhart, The good, the bad, and the ugly, but how ugly is ugly?, RV 2007, Lecture Notes in Computer Science 4839, Springer-Verlag, 2007, 126-138.
7. B. Bollig, M. Leucker. Deciding LTL over Mazurkiewicz traces, Data & Knowledge Engineering 44(2), 219-238 (2003).
8. S. Chakraborty, T. A. Henzinger, A. Sezgin, V. Vafeiadis, Aspect-oriented linearizability proofs. Logical Methods in Computer Science 11(1) (2015).
9. K. M. Chandy, L. Lamport, Distributed Snapshots: Determining the Global State of Distributed Systems, ACM Transactions on Computer Systems 3 (1985), 63–75.
10. C. M. Chase, V. K. Garg, Detection of Global Predicates: Techniques and Their Limitations, Distributed Computing, 11(1998), 191–201.
11. E. M. Clarke, E. A. Emerson, Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. Logic of Programs, Lecture Notes in Computer Science 131, Springer-Verlag, 1981, 52-71.
12. L. M. Danielsson, C. Sańchez, Decentralized Stream Runtime Verification for Timed Asynchronous Networks. IEEE Access 11, 84091-84112 (2023).
13. J. Dominguez, A. Nanevski, Visibility and Separability for a Declarative Linearizability Proof of the Timestamped Stack. CONCUR 2023, 1-16.
14. C. Fidge, Timestamps in message-passing systems that preserve the partial ordering, in K. Raymond (ed.), Proc. of the 11th Australian Computer Science Conference (ACSC'88), Volume 10, 56–66.
15. R. Ganguly, A. Momtaz, B. Bonakdarpour, Distributed Runtime Verification Under Partial Synchrony, OPODIS 2020, 20:1-17.

16. V. K. Garg, Elements of distributed computing. Wiley 2002.
17. V. K. Garg, Ch. Skawratananond, N. Mittal, Timestamping messages and events in a distributed system using synchronous communication. Distributed Comput. 19(5-6): 387-402 (2007).
18. B. Genest, D. Kuske, A. Muscholl, D. Peled, Snapshot Verification, TACAS 2005, Lecture Notes in Computer Science 3440, Springer-Verlag, 510-525.
19. C. A. R. Hoare, Communicating Sequential Processes. Prentice-Hall 1985.
20. K. Havelund, G. Rosu, Synthesizing monitors for safety properties, Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02), Lecture Notes in Computer Science 2280, Springer-Verlag, 2002, 342-356.
21. M. Herlihy, J. M. Wing, Linearizability: A Correctness Condition for Concurrent Objects. ACM Transactions on Programming Languages and Systems 12(3), 463-492 (1990).
22. C. Jard, Th. Jeron, G.-V. Jourdan, J.-X. Rampon, A General Approach to Trace-Checking in Distributed Computing Systems, 14th International Conference on Distributed Computing Systems, Pozman, Poland, 1994, pp. 396-403.
23. S. Katz, D. Peled, Interleaving Set Temporal Logic. Theoretical Computer Science 75(3): 263-287 (1990).
24. O. Kupferman, M. Y. Vardi. Model Checking of Safety Properties. Formal Methods System Design, 19(3), 291-314, 2001.
25. Z. Manna, A. Pnueli, The temporal logic of reactive and concurrent systems - specification. Springer 1992.
26. L. Lamport, Time, clocks, and the ordering of events in a distributed system. Concurrency: the Works of Leslie Lamport 2019, 179-196.
27. F. Mattern, Virtual Time and Global States of Distributed systems, Proceedings of Workshop on Parallel and Distributed Algorithms, Chateau de Bonas, France, Elsevier, 215-226.
28. A. Mazurkiewicz, Trace Semantics, Proceedings of Advances in Petri Nets 1986, Bad Honnef, Lecture Notes in Computer Science 255, Springer-Verlag, 1987, 279–324.
29. N. Mittal and V. K. Garg, Computation Slicing: Techniques and Theory, in *Distributed Computing, 15th International Conference, DISC 2001, Lisbon, Portugal, October 3-5, 2001, Proceedings*, J. L. Welch, Ed., Lecture Notes in Computer Science, vol. 2180, Springer, 2001, pp. 78–92.
30. N. Mittal, V. K. Garg. Techniques and applications of computation slicing. Distributed Computing 17(3), 251-277 (2005).
31. P. Niebert, D. Peled, Efficient model checking for LTL with partial order snapshots. Theor. Comput. Sci. 410(42): 4180-4189 (2009).
32. V. A. Ogale and V. K. Garg, "Detecting Temporal Logic Predicates on Distributed Computations," in *Distributed Computing, 21st International Symposium, DISC 2007, Lemesos, Cyprus, September 24-26, 2007, Proceedings*, A. Pelc, Ed., Lecture Notes in Computer Science, vol. 4731, Springer, 2007, 420–434.
33. C. H. Papadimitriou, The Theory of Database Concurrency Control. Computer Science Press 1986.
34. W. Penczek, R. Kuiper, Traces and Logic. The Book of Traces 1995, 307-390.
35. W. Penczek, On Undecidability of Propositional Temporal Logics on Trace Systems. Information Processing Letters 43(3), 147-153 (1992).
36. D. Peled, A. Pnueli, Proving Partial Order Properties, Theoretical Computer Science, 126:143–182, 1994.
37. D. Peled, Th. Wilke, P. Wolper, An Algorithmic Approach for Checking Closure Properties of Temporal Logic Specifications and Omega-Regular Languages, Theoretical Computer Science 195(2), 183-203 (1998).

38. S. S. Pinter, P. Wolper, A Temporal Logic for Reasoning about Partially Ordered Computations (Extended Abstract), PODC 1984, 28-37.
39. W. Reisig, Partial Order Semantics versus Interleaving Semantics for CSP-like Languages and its Impact on Fairness, ICALP 1984, Lecture Notes in Computer Science 172, Springer-Verlag, 403-413.
40. A. Sen, V. K. Garg, Detecting Temporal Logic Predicates in Distributed Programs Using Computation Slicing, OPODIS 2003, 171-183.
41. K. Sen, A. Vardhan, G. Agha, G. Rosu, Decentralized runtime analysis of multi-threaded applications. IPDPS 2006, 25-29 April 2006, Rhodes Island, Greece.
42. S. Stoller, Y.A. Liu, Efficient Symbolic Detection of Global Properties in Distributed Systems, CAV 1998, Lecture Notes in Computer Science 1427, Springer-Verlag, 357–368.
43. P. S. Thiagarajan, I. Walukiewicz, An Expressively Complete Linear Time Temporal Logic for Mazurkiewicz Traces. Information and Computation, 179(2), 230-249 (2002).
44. I. Walukiewicz, Difficult Configurations – On the Complexity of LTrL, International Colloquium on Automata, Languages and Programming, ICALP 1998, Lecture Notes in Computer Science 1443, Springer-Verlag, 140–151.
45. M.Y. Vardi, P. Wolper. Reasoning About Infinite Computations, Information and Computation, 115(1994), 1–37.
46. V. V. Willams, On Some Fine-Grained Questions in Algorithms and Complexity, ICM 2018, 3447-3487.
47. R. Williams, A new algorithm for optimal 2-constraint satisfaction and its implications. Theoretical Computer Science 348(2-3), 357-365, 2005.
48. G. Winskel, Event Structures. Advances in Petri Nets 1986, 325-392.
49. P. Wolper, M. Y. Vardi, A. P. Sistla, Reasoning about Infinite Computation Paths (Extended Abstract), FOCS 1983, 185-194.
50. PoET tool source code https://github.com/moraneus/PoET.
51. `Kairos` tool source code https://github.com/moraneus/kairos.

Appendix

A The Fidge-Mattern Vector Clocks Construction

The new algorithms presented in this paper use the Fidge and Mattern vector clock algorithm [14, 27]. This algorithm was originally presented in terms of asynchronous message passing, and we describe a simple translation into synchronized message passing. For simplicity and without loss of generality, we will restrict such synchronization to involve pairs of processes (which is by far the prevailing case).

Each vector clock VC of process p_i consists of k values, $VC[1] \ldots VC[k]$, for the k processes. In the current vector clock VC of process i, $VC[j]$ represents the number of events of process p_j that are known by process P_i to have already happened. Due to the distributed nature of the system, this knowledge does not include the actual number of events executed up to the current time, but only the information gathered through interactions between processes. However, the value of $VC[i]$, i.e., the number of events of p_i itself known to p_i is always accurate. For implementing vector clocks, we need to be able to include (piggyback) the vector clocks of the participating processes with each inter-process interaction, sharing this information between the involved processes.

The update of the vector clocks is performed as follows: For a local event e, belonging to process p_i, we perform $VC[i] := VC[i] + 1$. For an interaction e between processes p_i and p_j, the processes obtain the vector clocks VC_i and VC_j of the processes p_i and p_j respectively, piggybacked as part of the interaction. Then, a new vector clock VC is calculated as follows. First, for each $m \in [1..k]$ let $VC[m] := max(VC_i[m], VC_j[m])$, i.e., VC maximizes the value pairwise between the components of VC_i and VC_j. Further, set $VC[i] := VC[i] + 1$ and $VC[j] := VC[j] + 1$, corresponding to the fact that both processes have performed an additional (joint) event. Then VC becomes the new vector clock of both p_i and p_j.

Now, each event e reported to the monitor also includes the most recent vector clock associated with the participating process(s), denoted $VC(e)$. The order $e \prec f$, can be recovered from the vector clocks as follows:

$$e \prec f \leftrightarrow \forall i \in [1..k]\ VC(e)[i] \leq VC(f)[i] \wedge \exists i \in [1..k]\ VC(e)[i] < VC(f)[i]$$

The RV monitor can use the vector clock order to process events in an order that is a linearization of the partial order $\prec$ by processing an event f only *after* it already processed all the events e satisfying $e \prec f$. This is enforced as follows. The monitor process keeps a counter E_i that counts the number of events of process p_i that it has already processed. Processing a new event f, the monitor increments E_i for each $p_i \in Pr(f)$. Now, all the events that appear before a reported event f were already reported if the following two conditions hold:

- for each $p_i \in Pr(f)$, $E_i = VC(f)[i] - 1$, and
- for each $p_i \notin Pr(f)$, $E_i = VC(f)[i]$.

Otherwise, there is at least one event e in the execution that was not reported yet such that $e \prec f$. In this case, f is kept in a queue of unprocessed events, and waits to be processed by the RV algorithm when the above conditions will be satisfies.

B Ilustration of the computation RV for PaBTL

Figure 7 illustrates the evaluation of a compound formula $\psi = EP(EP(\varphi_1) \wedge EP(\varphi_2) \wedge \text{minterm} \wedge \neg EP(\psi_1) \wedge \neg EP(\psi_2) \wedge \neg EP(\psi_3))$. The diagram shows a partial-order execution where the frontiers P_1 and P_2 mark the minimal points where the positive subformulas hold. The minterm is satisfied at the frontier labeled M, initialized based on $P_1 \cup P_2$. The dashed lines corresponding to the frontiers N_1, N_2, and N_3 represent the points where each negative $EP(\psi_j)$ becomes true. For ψ to hold, requires also that all N_j do not occur strictly before M, which is the case here.

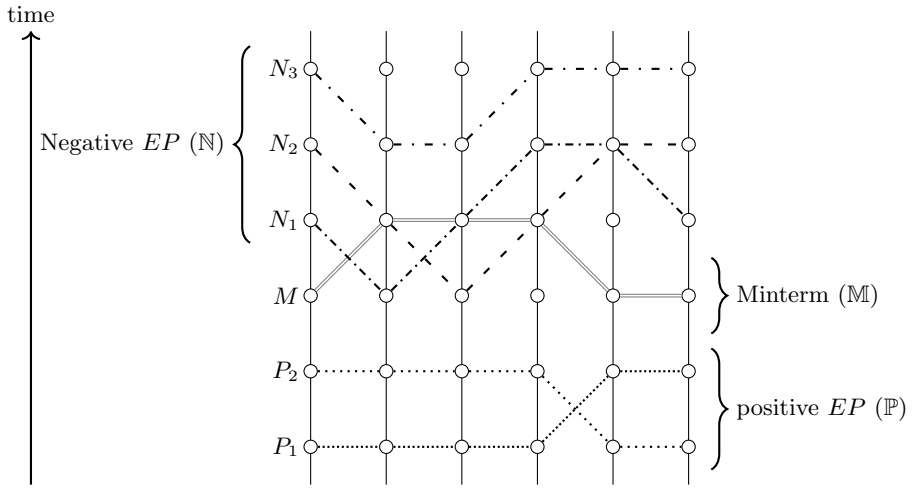

Fig. 7: Evaluation of $EP(EP(\varphi_1) \wedge EP(\varphi_2) \wedge \text{minterm} \wedge \neg EP(\psi_1) \wedge \neg EP(\psi_2) \wedge \neg EP(\psi_3))$

C Proof of Size Complexity of the Translation to DNF of DLNFs

We prove that a PaBTL formula can be translated into a DNF of DLNFs, which can be up to exponentially longer. We consider a formula of the form $EP\varphi$, where φ is itself in PaBTL. If the outermost formula is not in this form, we can prefix it by EP and later, after the transformation, deleting the outermost EP occurrences from the obtained disjuncts.

The proof is by induction on the depth of EP subformulas. Take a subformula $EP\varphi$ where φ has immediate subformulas $\varphi_1, \ldots, \varphi_k$. Rewrite φ in disjunctive

normal form with disjuncts $\psi_1 \vee \ldots \vee \psi_n$, where each ψ_j is a conjunction, with $k < m$ conjuncts, and $n \leq 2^k$. Then, the formula $EP\varphi$ can be written as $EP(\psi_1 \vee \ldots \psi_n)$ and we can use the distribution of the disjunction over the EP operator and rewrite it as $EP\psi_1 \vee \ldots EP\psi_n$. For the base of the induction, where each φ_i does not include further EP subformulas, this is just a conversion of φ to DNF followed by distribution of EP with disjunction, and we are done with the size of the obtained formula being $O(2^{|\varphi|})$.

In case that (some of) the subformulas φ_i contain further nested EP subformulas, we need to get rid of nested disjunctions. Each ψ_j, obtained in the transformation described above by converting φ to a DNF is a conjunction of components that can be either a literal (negated or non-negated proposition), a formula of the form $EP\varphi_i$ or $\neg EP\varphi_i$.

By induction, we can translate each component φ_i into DNF formulas over DLNF formulas, in the same way that was described above with at most $2^{|\varphi_i|}$ disjuncts, each one with up to $|\varphi|$ disjuncts of components of DLNF formulas. In each $EP\varphi_i$ subformula, we can, as before, commute the disjunction over the EP, obtaining up to $2^{|\varphi_i|}$ disjunctions of DLNF components, each with up to $|\varphi_i|$ top level components.

Returning to the fomrula ψ_j, literals are already in normal form, and remain as they are. Non-negated subformulas of the form $EP\varphi_i$ are replaced, by induction, with at most $2^{|\varphi_i|}$ EP disjuncts, each of at most $|\varphi_i|$ DLNF formulas. For the negated elements $\neg EP\varphi_i$ we first distribute the disjuncts obtained by translated φ_i to the normal form as for the non-negated case. Then we use De Morgan law to push negation over the disjunction. We obtain at most $2^{|\varphi|}$ conjunctions negated EP (each one with up to $|\varphi|$ top level conjuncts. Now we can distribute the disjunctions from the rewritten ψ_j over the EP.

To do that, we rewrite the formula inside the EP. For each rewritten subformula $EP\varphi_i$ we have a disjunction of at most $2^{|\varphi_j|}$ components, taking the conjunction on these components, we need to take all the combination of of disjuncts, one per each φ_i. This makes $2^{|\varphi_i|} \times \ldots \times 2^{|\varphi_k|}$ elements, which is $2^{|\varphi_1|+\ldots+|\varphi_k|}$. The rest of the conjuncts in $EP\psi_j$ are the literals and the negated EP; the latter contribute the sum of $2^{|\varphi_i|}$ per each $\neg EP\varphi_i$.