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 PCTL (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 PBTL. 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, 18] 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* [26, 47] among the events sometimes offers a more direct and intuitive view of executions that can be distributed among different processes. There, events executed independently by different processes, which can also overlap in time with each other, are considered 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 [44]. Specification over this model is often given using Linear Temporal Logic (LTL) [23]. Practice shows that for most purposes, the interleaving

model is sufficient for modelling 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 [36]. 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* [8] of a distributed system, i.e. a consistent collection of local states of the system. Such a snapshot corresponds as global state 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 [7, 11, 19, 32] 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, to simplify 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 PCTL, 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 [49]. We provide a related hardness results.

We then present a second algorithm, for a subset of this logic, which we call PBTL, confined to the past operators EP (and its dual AH) together with the Boolean operators. We present a corresponding tool called `Kairos` [50]. 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.

[1] If one groups together all the interleaving sequences that are consistent with the partial order execution as in *Mazurkiewicz traces* [26], there is at least one interleaving in which this global state appears

Related work. Several logics are interpreted over partial order executions, see, e.g., the survey [33]. The branching time temporal logic POTL [37] includes both future and past branching operators, in the style of the logic CTL [10]; the interpretation is over local states and events, *combined* into a single structure the possible partial order executions of a system, as in *event structures* [47]. Other temporal logics that are interpreted over partial order executions that are defined, as in this paper, 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 [21, 35] uses the operators of CTL, applied to each such branching structure. The logic LTrL [42] 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 [35] is studied in [1, 3, 34, 43] and of LTrL in [43].

An RV verification algorithm, where the 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*), was described in [4]. Thus, the same formula has a completely different interpretation in that work than in our work. In [9, 40, 41], procedures are presented for deciding whether a partial order execution satisfies properties that can be written as a *restricted* future version of the logic ISTL [1]. These works identify also cases of specifications with lower complexity. Ogale and Garg [31] have proposed a logic called Basis Temporal Logic (BTL), which is also a subset of ISTL and a decision procedure for checking whether a partial order execution satisfies a BTL property 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, are aimed at proving that a property holds for a *representative* interleaving of a partial order execution, which does not necessarily follows the order of intercepted events [13].

2 Preliminaries

The Partial Order Model (Local View) In the *partial order execution model* [24, 38, 47], events of disjoint processes may be unordered (independent) 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, asymmetric 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* $\mathcal{E} = \langle E, \prec, \mathcal{P}, Pr, \iota, A, L \rangle$, has the following components:

- E is a (finite or infinite) set of *events*.
- $\prec \subset E \times E$ is a *partial order* (i.e., transitive, irreflexive and asymmetric) relation over E. In addition, $\prec$ is well-founded, i.e., E does not have an infinite decreasing chain of events $e_1 \succ e_2 \succ \ldots$, where $\succ$ is the symmetric

(converse) version of $\prec$. If both $e \not\prec f$ *and* $e \not\succ f$, then we say that e and f are said to be *concurrent* or *independent*.
- $\iota \in E$ is the *initial event* of E. It is *minimal* w.r.t. $\prec$.
- $\mathcal{P}$ is a finite set of *processes*.
- $Pr : E \mapsto 2^{\mathcal{P}}$ maps each event to a set of processes that are involved in its execution. Typically, most events involve only a single process, representing a *local* event, or a pair of processes, representing *synchronous* or *handshake* communication. For the initial event, $Pr(\iota) = \mathcal{P}$. For each $p \in \mathcal{P}$, the events that involve a process p, i.e., $\{e \in E | Pr(e) = p\}$ are totally ordered.
- $A = \biguplus_{p \in P} A_p$ is a finite set of *propositions*. The set A is partitioned according to processes[2] to sets A_p for each process $p \in \mathcal{P}$.
- $L(e, p) \in 2^{A_p}$ 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 $A_p \setminus L(e, p)$ appear negated. In a short form, conjunctions are removed and negated propositions are marked with an overbar, hence $t_1 \wedge \neg t_2$ is denoted $t_1\overline{t_2}$. The described model can represent synchronous (handshake) communication as in the programming language CSP [17]. 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 set $A_{p_1} = \{t_1, t_2\}$, $A_{p_2} = \{r_1, r_2\}$ and $A_{p_3} = \{q_1, q_2\}$.

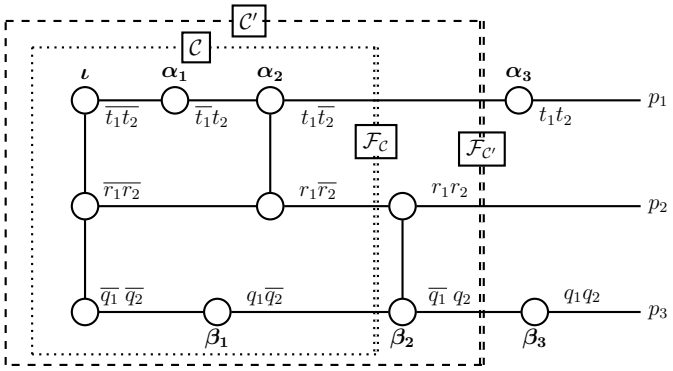

Fig. 1: A partial order execution

The Global View The described partial order execution model gives a *local view* of the computation, where a propositional assignment describes the *local state* of the processes participating in the event immediately *after* its occurrence. Based on the local view, we further define a *global view*, which contains *cuts*

[2] Thus, each proposition can represent some property (predicate) *local* to some process.

(and *frontiers*) that correspond to *global states* [24]. 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.

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 $\mathcal{C}$[3] is the set of *maximal* events from $\mathcal{C}$ for the different processes in $\mathcal{P}$ i.e., $\bigcup_{p \in \mathcal{P}} max(\mathcal{C}, p)$. Note that a single event can play the role of a maximal event for *multiple* processes if it involves these processes. In Figure 1, the marked cut $\mathcal{C}$, whose events are enclosed within an inner dotted box, contains the events $\{\alpha_1, \alpha_2, \beta_1\}$. 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 state of the cut $\mathcal{C}$ is labeled with the combination of the propositions assigned to the three processes. This can be represented by the Boolean minterm $t_1\overline{t_2}r_1\overline{r_2}q_1\overline{q_2}$. 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, $L(\mathcal{C}) = \biguplus_{p \in \mathcal{P}} L(max(\mathcal{C}, p), p)$.

Note that events that are *not independent* of each other can be maximal w.r.t. different processes, hence can belong according to the above definitions to the same frontier. This happens, for example, in the cut $\mathcal{C}'$, which appears within the outer dotted box in Figure 1; its 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, for $\mathcal{F}_{\mathcal{C}'}$, the interpretation is then $t_1\bar{t}_2r_1r_2\bar{q}_1q_2$.

We can now define, based on the (local) partial order execution, a corresponding *global* partial order between the cuts (and, accordingly, 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 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 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* [26]). 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 forntier $\mathcal{F}_\mathcal{C}$. This is a *Hasse diagram* of the global view, where the depicted edges represent the "immediate successor" relation $\twoheadrightarrow$.

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

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; this is because if e and f were dependent of one another (they share at least one process) then e and f must have been ordered with respect to each other in the generating local partial order; thus cannot both follow up immediately from the same frontier $\mathcal{F}_\mathcal{C}$. Furthermore, because e and f are independent, we also have $\mathcal{F}_\mathcal{C} \overset{e}{\twoheadrightarrow} \mathcal{F}_{\mathcal{C}_1} \overset{f}{\twoheadrightarrow} \mathcal{F}_{\mathcal{C}'}$ and $\mathcal{F}_\mathcal{C} \overset{f}{\twoheadrightarrow} \mathcal{F}_{\mathcal{C}_2} \overset{e}{\twoheadrightarrow} \mathcal{F}_{\mathcal{C}'}$. We 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 also independent.

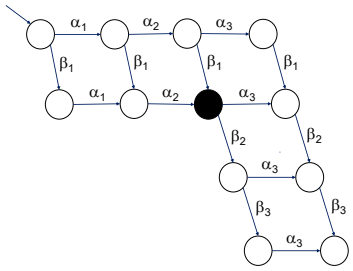

Fig. 2: A Hasse diagram of the global states view of the partial order execution

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 M. 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 M 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* [24]. As done in [20], we adopt the use of Fidge and Mattern [12, 25] *vector* clocks, where each event keeps a vector of values, one per each involved process. Comparing 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 observe 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) arrive at 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. As the description of the Fidge and Mattern vector clocks appears in the literature, it is included for convenience in the appendix A.

The Logic and the interpretation over the global view We use a branching temporal logic, in the style of CTL [10], to specify properties of the global view of partial order executions. As in CTL, each temporal operator consists of a pair of a quantifier operator: over a set of paths, either A (forall paths) or E (there exists a path), and over a path. The path operators are the *past* mirror images of the CTL operators: $\mathcal{S}$ (*since*) instead of $\mathcal{U}$ (*until*) and Y (*yesterday*) instead of X (*nexttime*). We call this logic PCTL (for *past* CTL). The syntax is as follows, where q is a proposition:

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

The semantics is as follows:

- $S \models q$ 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 $S' \models E(\varphi \mathcal{S} \psi)$.
- $S \models A(\varphi \mathcal{S} \psi)$ iff 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: $\mathit{true} = (q \vee \neg q)$ for some proposition q, $\mathit{false} = \neg \mathit{true}$, $(\varphi \vee \psi) = \neg(\neg\varphi \wedge \neg\psi)$, $EP\varphi = E(\mathit{true} \mathcal{S} \varphi)$, $AY\varphi = \neg EY \neg\varphi$, $AP\varphi = A(\mathit{true} \mathcal{S} \varphi)$, $EH\varphi = \neg AP \neg\varphi$ and $AH\varphi = \neg EP \neg\varphi$, where P reads as *previously* and H reads as *historically*.

Advanced comment. We show why some properties expressed over the global view cannot be expressed with properties over the 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 that the logic TLC can be translated into an automaton [4], hence a regular language. On the other hand, PCTL can express properties that cannot be expressed as regular languages by comparing sequences 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 distributed executions against a branching logic PCTL specification. Events performed by the monitored processes are reported to the centralized monitor, which updates a summary of the seen events and calculates a verdict. The monitor obtains

information about the process(es) participating in each reported event, and a minterm that represents the values of the propositions of the involved processes after the execution of the event. For example, the event α_2 in Figure 1 is reported to the monitor by one of the participating processes p_1 or p_2 and includes the minterm $t_1\overline{t_2}r_1\overline{r_2}$, which combines the propositional values of both participating processes.

In the case of the logic PCTL interpreted over the global view of the partial order execution, we construct a *summary* that consists of a *subgraph* of the global view; the nodes represent the frontiers and the edges correspond to the relation $\twoheadrightarrow$ between them. Each edge is labeled with an event that forms the transition between the incoming frontier and the outgoing frontier of that edge. With each newly observed event, the summary graph is updated, adding new nodes and edges, whereas some old nodes may become redundant, hence can be removed. We can consider the constructed graph, obtained after each sequence of events, as a *sliding window* into the global view graph, which slides with each newly observed event. Each node in the graph is labeled with a vector of Boolean values, corresponding to the subformulas that hold in the frontier it represents. Calculating the vector that labeled a node s in the graph is performed based on the collection of incoming edges of s and the predecessor states.

Constructing a Sliding Window of Relevant Frontiers. The monitor process updates from the reported events the sliding window of the graph containing the frontiers related to the partial order execution. It does not construct intermediately the (local) partial order execution $\mathcal{E}$, nor does it construct the corresponding cuts. As a building block of the algorithm, for each given frontier $\mathcal{F}_\mathcal{C}$ we can construct its f successor $\mathcal{F}_{\mathcal{C}'}$, i.e., $\mathcal{F}_\mathcal{C} \overset{f}{\twoheadrightarrow} \mathcal{F}_{\mathcal{C}'}$; after adding f to the events of $\mathcal{F}_\mathcal{C}$, events that are not any more maximal for *at least* one process, are removed. Formally:

1. Set $\mathcal{F}_{\mathcal{C}_{temp}} := (\mathcal{F}_\mathcal{C} \cup \{f\})$.
2. Remove from $\mathcal{F}_{\mathcal{C}_{temp}}$ any event e such that for each $p \in Pr(e)$ there exists some $g \in \mathcal{F}_{\mathcal{C}_{temp}}$, where $p \in Pr(g)$ and $e \prec g$.

For example, the frontier $\mathcal{F}_\mathcal{C} = \{\alpha_2, \beta_1\}$, which corresponds to the cut $\mathcal{C}$ in Figure 1, 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 $\mathcal{C}'$ in Figure 1, such that $\mathcal{F}_\mathcal{C} \overset{\beta_2}{\twoheadrightarrow} \mathcal{F}_{\mathcal{C}'}$ (and $\mathcal{C} \overset{\beta_2}{\twoheadrightarrow} \mathcal{C}'$), is constructed by first adding β_2 to $\mathcal{F}_\mathcal{C}$ and then removing from it β_1; the latter event is not any more maximal as its single participating processes p_3, which participates also in the event β_2. Thus, $\mathcal{F}_{\mathcal{C}'} = \{\alpha_2, \beta_2\}$. The event α_2 is *not* removed from the obtained frontier $\mathcal{F}_{\mathcal{C}'}$, 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 $\mathcal{F}_{\mathcal{C}_m}$ of G according to the order $<$. This is the frontier that corresponds to the cut containing all the events processed so far by the algorithm. Initially, the graph consists of a single node s_m, representing the frontier $\{\iota\}$, consisting of the initial event, and labeled with propositions according to $L(\iota)$. 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 appears, we first add to G an edge $s_m \overset{e}{\twoheadrightarrow} s_n$, where s_n is a new maximal node representing the frontier $\mathcal{F}_{\mathcal{C}_n}$ such that $\mathcal{F}_{\mathcal{C}_m} \overset{e}{\twoheadrightarrow} \mathcal{F}_{\mathcal{C}_n}$, i.e. $\mathcal{F}_{\mathcal{C}_n}$ is the e successor of the previous maximal frontier $\mathcal{F}_{\mathcal{C}_m}$. Further, new nodes and edges are added to G as follows: For edges $s \overset{e}{\twoheadrightarrow} s' \overset{f}{\twoheadrightarrow} s''$, with $Pr(e) \cap Pr(f) = \emptyset$, f can *propagate backwards* over the edge $s \overset{e}{\twoheadrightarrow} s'$. Consequently, edges $s \overset{f}{\twoheadrightarrow} r \overset{e}{\twoheadrightarrow} s''$ are added, with r added as a new node. This is due to the commutativity described in Section 2. The construction of the new frontier for the node r from the frontier of the node s and the event f is as described earlier this section. In the same way, the addition of the node r may induce, through commutativity, further backward propagations.

After finishing a phase of extending the graph G with new nodes due to observing a new event e, removing *redundant* nodes (and all of their emanating edges) from G starts. Accordingly, a node s becomes redundant when it can no longer affect the RV verdict. This happens when the occurrence of future events cannot generate further successors for s. A sufficient condition for this is that for each process $p \in \mathcal{P}$, there already has been some event α involving p (i.e., $p \in Pr(\alpha)$ on an edge constructed from s. For each state s on G, we can accumulate such processes in the set R_s, initialized to the empty set, and 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.

We now demonstrate the first few steps of the construction windows for the observation $\sigma = \alpha_1\alpha_2\beta_1$, which is a prefix of the linearization $\sigma = \alpha_1\alpha_2\beta_1\alpha_3\beta_2\beta_3$ 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. Note that the backward propagation in Step **E** propagates into another backward propagation that appears in Step **F**. The dotted nodes in Step **F** become redundant and are removed, resulting in Step **G**.

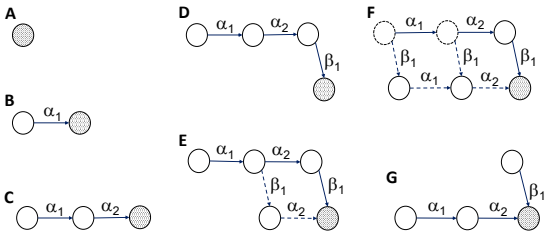

Fig. 3: Graphs constructed for the trace $\sigma = \alpha_1\alpha_2\beta_1\alpha_3$.

Calculating the Values of Subformulas on the Nodes. The following procedure calculates the truth value for each subformula for each new frontier (state) constructed by the sliding window graph-updating algorithm. This can be compared to the update single summary vector used in the case of past time LTL [18]. The update is based on the main operator of each subformula of the specification φ. For each such subformula η and each node representing a frontier

$\mathcal{F}$, we need to 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 for smaller subformulas within η, hence must be performed *later*. Further, the calculations also use the values of truth values previously calculated for the *predecessors* frontiers in the graph.

$val(\mathcal{F}, q) = true$ 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 $val(\mathcal{F}', A(\eta_1 S \eta_2)) = true$.

Complexity. The overall number of constructed global states is $\mathcal{O}((|E|/k)^k)$, where $|E|$ is the overall number of events and $k = |\mathcal{P}|$ (i.e., the number of processes); the worst case scenario occurs when all the events involving any given process are independent of all the events of any other process. Calculating the vector of Boolean values for subformulas, related to a frontier, is done in time $|\varphi| \times k$. This gives a complexity of $\mathcal{O}(|\varphi| \times k \times (|E|/k)^k)$, which is linear in the size of the property but exponential in the number of processes.

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 exponential in the number of process. In particular, after each newly observed event, the number of nodes (frontiers) added to G can be $\mathcal{O}((|E|/k)^{k-1})$, 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. (this 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 PCTL over a partial order execution. The overall complexity of the described RV algorithm is $\mathcal{O}((|E|/k)^k)$, when ignoring the linear factors involving updating the bit vectors for the subformulas. 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, which typically considers only a single parameter. Fine-grained complexity [45] 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 both 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 the complexity is not only exponential in n, 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 [46], which connects the complexity of our target problem to this hypothesis. The Boolean vectors $a_1, a_2, \ldots, a_k$ are said to be *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, given k sets $E_1, E_2, \ldots, E_k$, each contains n d-bit vectors, where $d = O(\log^2 n)$, determine 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 [46] states that the *k-OV* problem requires time at least $\Omega\left(n^{(1-\epsilon)k}\right)$ for some 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, an $1/k$ of the total number of events $|E| = \sum_{i=1}^{k} |E_i| = O(kn \log^2 n)$, where the vectors are encoded one after the other, separated by a delimiter as follows. Therefore in our case $n = \Omega(\frac{|E|}{k \log^2 \frac{E}{k}})$ and we get a lower bound of $\Omega\left(\left(\frac{|E|}{k \log^2 \frac{|E|}{k}}\right)^{(1-\epsilon)k}\right)$. As stated above, the base of the exponent is within a polylogarithmic factor of the upper bound $\mathcal{O}((|E|/k)^k)$.

- **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, and the value of v_i during such delimiter events is irrelevant.
- **Synchronization mechanism:** A binary counter proposition l_i is associated with each process[4]. This counter alternates between 0 and 1 across

[4] Adding bit patterns to compare events in different processes appears in [43].

successive non-delimiter events within the same process. When $\delta_i = 1$ (during delimiter events), the value of l_i is not significant.

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 using a *since* ($\mathcal{S}$) formula of the form $\psi \wedge EY\, E(\varphi\, \mathcal{S}\, \psi)$ where:

1. ψ identifies global states where all processes are at delimiter events, expressed as a conjunction of all δ_i propositions. The following conditions are enforced as conjuncts within φ.
2. The counter bits l_i must follow a specific *zigzaging* pattern: either all processes have the same value, or processes 1 through i (for some $1 \leq i \leq k$) have one value (0 or 1), while processes $i+1$ through k have the complement value. This is efficiently encoded in size $O(k)$ by a disjunction of two cases:
   (a) A conjunction of $(\neg l_i \vee l_{i+1})$ formulas (equivalent to $l_i \rightarrow l_{i+1}$) for all adjacent indices $1 \leq i < k$. This captures situations where a block of 0 values is followed by a block of 1 values.
   (b) A conjunction of $(l_i \vee \neg l_{i+1})$ formulas for all adjacent indices, capturing the opposite arrangement where a block of 1 values is followed by a block of 0 values.
3. When all the bits of the form l_i have identical values (either all 0 or all 1), 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. To complete the specification, this condition is applied within φ to all global states where the l_i bits are equal, effectively encoding a disjunction over all relevant bit positions.

4 An Efficient Algorithm for a Subset of the Logic

There have been several works that suggested an efficient runtime verification algorithm for temporal properties over a partial order execution [9, 40, 41]. These works deal with properties that can be expressed using a restricted future versions of the logic; so, instead of the EP operator in PCTL, 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"). In [31] this logic is called BTL, and a decision procedure for runtime verification of such properties is obtained using a procedure called *slicing*. A model checking algorithm for this logic is described in [1], based on translating conditions on linearizations of the partial order into an automaton. The logic PBTL is obtained by restricting the PCTL syntax to the operators EP and AH ($= \neg EP \neg$) and the standard Boolean operators ($\wedge$, $\vee$, $\neg$). We will present an algorithm for PBTL.

The first step of applying the algorithm is to transform the specification formula into a Boolean combination over subformulas that are in *disjunctionless normal form*, to be called here DLNF. The Boolean combination is in disjunctive normal form (DNF) formula. Each DLNF formula is of the following form:

- 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 $\neg\varphi$.

Translation of a formula into a DNF Boolean combination of DLNF subformulas for BTL is proposed in [1]. This is done by repeatedly distributing disjunctions with the EP modality, based on the equivalence $EP(\varphi \vee \psi) = (EP\varphi \vee EP\psi)$. This can result in an exponential explosion in the size of the specification, see the proof in Appendix C.

Our algorithm for RV of PBTL properties employs a procedure for *detecting* a minimal frontier M that satisfies a conjunction of literals $\eta = \bigwedge_{1 \leq i \leq n} \gamma_i$, with each γ_i consists of variables local to a single process p_i; further, $P \leq M$ for some given frontier P. This procedure, described in the next paragraph, follows a procedure presented in [28], on top of which we added the condition $P \leq M$.

We use 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 elements of M may point to the same event. We initiate M to the events in P. When a new event α is observed and added to the partial order execution, we check for each $p_i \in Pr(\alpha)$ whether $M[i]$ satisfies γ_i. If this holds for each $p_i \in Pr(\alpha)$, there is no need to change M. Otherwise, we set $M[i]$ to α. This may cause the following chain updates of the vector M in order to form again a frontier. Each time an element $M[i]$ is updated, (starting with the event α) we may need to progress the event in $M[j]$, for some $j \neq i$: this happens if the already observed partial order includes an immediate successor β to $M[j]$ (then $p_j \in Pr(\beta)$) such that $\beta \preceq M[i]$. Then we set $M[j]$ to β. The order $\preceq$ can be checked by comparing the global clock vectors added to the events by the Fidge and Mattern algorithm. At some point, this chain of updates must stabilize (there are only finitely many events) and the events in M form again a frontier of the observed partial partial order execution. Now, if for each i, $M[i]$ satisfies γ_i, we have finalized the calculation of M. Otherwise we need to wait for the next observed event, as described below, to continue the search for a frontier satisfying η.

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

The following algorithm is applied to each subformula of the form $EP\varphi$ in the DLNF form, including separately for subformulas embedded within another such subformula. 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 to the partial order. The updates for $EP\varphi$ must follow the updates for all the subformulas in φ. For each 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 as follows:

- $\mathbb{P}$ For $1 \leq j < k$, $\eta_j = EP\varphi_j$. These are the *positive* conjuncts. We denote the minimal 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 minimal 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 vector M that is used to calculate the minimal frontier that satisfies $EP\varphi$. We also keep an indication whether $EP\varphi$ was found to already hold. 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 in 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 occurrences of further events. Let P_j refers to the minimal frontier satisfying the subformula $EP\varphi_j$, for $1 \leq j < l$, that appears positively (i.e., of the form $\mathbb{P}$ *within* $EP\varphi$). For the initializations, we need calculate the minimal 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 process p_i among the 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 $C = \bigcup_{1 \leq j < k} C_j$). Initialization (including fixing) of M to the frontier $P = \bigsqcup_{1 \leq j < k} P_j$ takes place when all the components of the form P_j required for calculating P were detected.

The fourth columns 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 may be updated upon adding a new event 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 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 minimal frontier P_i satisfying it. Further 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) only refers to frontiers N_l that were detected when M that satisfies η_k was also 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 specification, we apply the Boolean operators as appearing in the DNF combination 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 $\mathcal{O}(|E| \times k^2 \times 2^{|\varphi|})$. 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 satisfying a Boolean formula φ was shown in [31] to be in 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 PBTL by setting the verified property to $EP\varphi$.

5 Implementations and Experiments

We developed two runtime verification tools implementing the monitoring algorithm presented in this paper. PoET [49] implements the complete PCTL algorithm described in Section 3, supporting the full branching temporal logic with past operators including complex nesting and arbitrary formula structures. `Kairos` [50] (from the ancient Greek concept of opportune time) implements the PBTL algorithm from Section 4, supporting the restricted subset of PCTL 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 PBTL properties (Figure 4), covering different cases, with results shown in Table 1. 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 [49, 50]. Further experiments, with properties that can be expressed in PCTL but not in PBTL, appear in Appendix D.

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. 4: Properties in the PBTL formalism.

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)

6 Conclusions

We studied RV for partial order executions. Specifically, we used the specification formalism past time branching temporal logic (PCTL), interpreted on the partial order structure between frontiers/cuts obtained from a partial order execution.

We presented a runtime verification algorithm with complexity exponential in the number of processes (base: number of events) and linear in specification size. We developed a prototype tool PoET implementing this algorithm. We also presented an algorithm for PBTL, a subset restricted to temporal operator EP and Boolean operators (including $AH = \neg EP\neg$). This algorithm has linear complexity in events, quadratic in processes, and exponential in specification size. We implemented this in the Kairos tool.

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

References

1. R. Alur, K. McMillan, D. Peled, Deciding Global Partial-Order Properties. International Colloquium on Automata, Languages and Programming, *ICALP 1998*, Lecture Notes in Computer Science 1443, Springer-Verlag, 41–52.
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, An 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. S. Chakraborty, Th. A. Henzinger, A. Sezgin, V. Vafeiadis, Aspect-oriented linearizability proofs. Logical Methods in Computer Science 11(1) (2015).
8. K. M. Chandy, L. Lamport, Distributed Snapshots: Determining the Global State of Distributed Systems, ACM Transactions on Computer Systems 3 (1985), 63–75.
9. C. M. Chase, V. K. Garg, Detection of Global Predicates: Techniques and Their Limitations, Distributed Computing, 11(1998), 191–201.
10. 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.
11. J. Dominguez, A. Nanevski, Visibility and Separability for a Declarative Linearizability Proof of the Timestamped Stack. CONCUR 2023, 1-16.
12. 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.
13. R. Ganguly, Y. Xue, A. Jonckheere, P. Ljung, B. Schornstein, B. Bonakdarpour, M. Herlihy, Distributed runtime verification of metric temporal properties. Journal of Parallel Distributed Computing 185, 104801 (2024).
14. V. K. Garg, Elements of distributed computing. Wiley 2002.
15. 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).
16. B. Genest, D. Kuske, A. Muscholl, D. Peled, Snapshot Verification, TACAS 2005, Lecture Notes in Computer Science 3440, Springer-Verlag, 510-525.
17. C. A. R. Hoare, Communicating Sequential Processes. Prentice-Hall 1985.
18. 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.
19. M. Herlihy, J. M. Wing, Linearizability: A Correctness Condition for Concurrent Objects. ACM Transactions on Programming Languages and Systems 12(3), 463-492 (1990).
20. 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.

21. S. Katz, D. Peled, Interleaving Set Temporal Logic. Theoretical Computer Science 75(3): 263-287 (1990).
22. O. Kupferman, M. Y. Vardi. Model Checking of Safety Properties. Formal Methods System Design, 19(3), 291-314, 2001.
23. Z. Manna, A. Pnueli, The temporal logic of reactive and concurrent systems - specification. Springer 1992, ISBN 978-3-540-97664-6.
24. L. Lamport, Time, clocks, and the ordering of events in a distributed system. Concurrency: the Works of Leslie Lamport 2019, 179-196.
25. 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.
26. A. Mazurkiewicz, Trace Semantics, Proceedings of Advances in Petri Nets 1986, Bad Honnef, Lecture Notes in Computer Science 255, Springer-Verlag, 1987, 279–324.
27. 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.
28. N. Mittal, V. K. Garg. Techniques and applications of computation slicing. Distributed Computing 17(3), 251-277 (2005).
29. P. Niebert, D. Peled, Efficient model checking for LTL with partial order snapshots. Theor. Comput. Sci. 410(42): 4180-4189 (2009).
30. D. Peled, Specification and Verification of Message Sequence Charts, Formal Techniques for Distributed System Development (*FORTE/PSTV*) 2000, pp.139-154. 1993: 409-423.
31. 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, pp. 420–434.
32. C. H. Papadimitriou, The Theory of Database Concurrency Control. Computer Science Press 1986.
33. W. Penczek, R. Kuiper, Traces and Logic. The Book of Traces 1995, 307-390.
34. W. Penczek, On Undecidability of Propositional Temporal Logics on Trace Systems. Information Processing Letters 43(3), 147-153 (1992).
35. D. Peled, A. Pnueli, Proving Partial Order Properties, Theoretical Computer Science, 126:143–182, 1994.
36. 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).
37. S. S. Pinter, P. Wolper, A Temporal Logic for Reasoning about Partially Ordered Computations (Extended Abstract), PODC 1984, 28-37.
38. 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.
39. A. P. Sistla, E. M. Clarke, The Complexity of Propositional Linear Temporal Logics, Journal of the ACM 32(3), 733-749 (1985)
40. A. Sen, V. K. Garg, Detecting Temporal Logic Predicates in Distributed Programs Using Computation Slicing, OPODIS 2003, 171-183
41. 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.

42. P. S. Thiagarajan, I. Walukiewicz, An Expressively Complete Linear Time Temporal Logic for Mazurkiewicz Traces. Information and Computation, 179(2), 230-249 (2002).
43. 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.
44. M.Y. Vardi, P. Wolper. Reasoning About Infinite Computations, Information and Computation, 115(1994), 1–37.
45. V. V. Willams, On Some Fine-Grained Questions in Algorithms and Complexity, ICM 2018, 3447-3487.
46. R. Williams, A new algorithm for optimal 2-constraint satisfaction and its implications. Theoretical Computer Science 348(2-3), 357-365, 2005.
47. G. Winskel, Event Structures. Advances in Petri Nets 1986, 325-392.
48. P. Wolper, M. Y. Vardi, A. P. Sistla, Reasoning about Infinite Computation Paths (Extended Abstract), FOCS 1983, 185-194.
49. `PoET` tool source code https://github.com/moraneus/PoET.
50. `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 [12, 25]. 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, $VC[1] \dots VC[k]$, 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, let $VC[m] := max(VC_i[m], VC_j[m])$ for each $m \in [1..k]$, 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 PBTL

Figure 5 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.

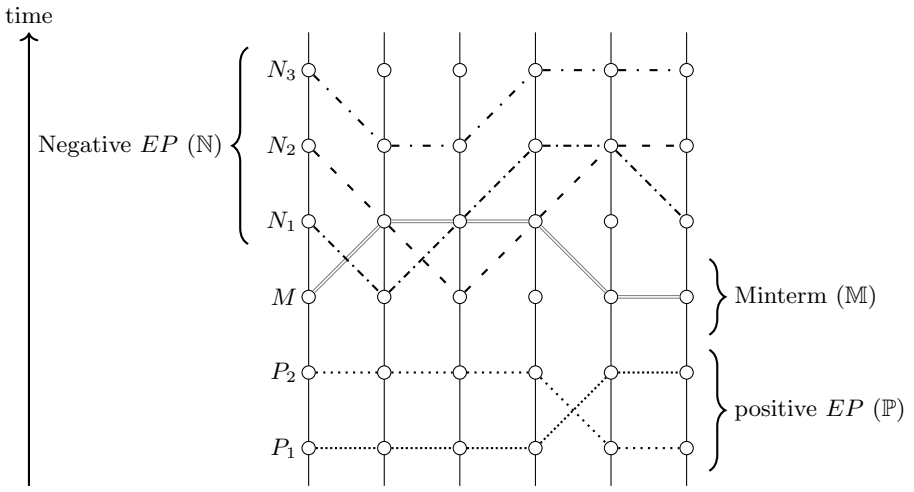

Fig. 5: Evaluation of $EP(EP(\varphi_1) \wedge EP(\varphi_2) \wedge \text{minterm} \wedge \neg EP(\psi_1) \wedge \neg EP(\psi_2) \wedge \neg EP(\psi_3))$

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

The proof is by induction on the depth of EP subformulas. Take a subformula $EP\varphi$ where φ has immediate (i.e., not subfomrulas of subofmrulas) subformulas $\varphi_1, \ldots, \varphi_n$. For the moment, replace each φ_i by an atom (a new proposition) $\hat{\varphi}_i$. We can rewrite $EP\varphi$ in the form $EP(\psi_1 \vee \ldots \psi_m)$, where $m \leq 2^n$ and each ψ_j is a minterm over the new atoms of the form $\hat{\varphi}_i$, with size no more than m each. For the base of the induction, we have an EP formula without nested EP subformulas. Then, $\hat{\varphi}_i = \varphi_i$, and we can now distribute the disjuncts over the EP.

By induction, we can translate each φ_i into a DNF of subformulas in DLNF; for each φ_i of length $k_i = |\varphi_i|$, we obtain a formula φ_i' with up to 2^{k_i} disjuncts, each of up to k_i conjuncts, each one of them in DLNF normal form. Now we replace each postive occurrence in the subformulas ψ_j by $\hat{\varphi}_i$ with φ_i'. Replacing each negative occurrence $\neg\hat{\varphi}_i$ in ψ_j, we obtain a negation of a DNF formula for which can be converted into a CNF formula by using De Morgan dulatity between conjunctions and disjunctions. Now, this can be translated back into DNF, with no more than 2^{k_i} dusjuncts, each of at most k_i conjuncts[5].

After this rewriting, each ψ_j becomes a conjunction of up to n subfomrulas, each of which is a disjunction of up to 2^{k_i} disjunctions with up to k_i conjuncts. This can be by distibuting conjunctions and disjunctions into a formula of size $2^{k_1} \times \ldots \times 2^{k_n}$. Recall that we have $m \leq 2^n$ disjuncts ψ_j. This gives us $2^n \times 2^{k_1} \times \ldots 2^{k_n} = 2^{n+k_1+\ldots+k_n}$ disjuncts. But the size of $EP\varphi$, when fully parenthesized is not more than $n + k_1 + \ldots + k_n$. Finally, we can now commute the disjuncts with the EP operator that includes them. The subformulas in this translation is in DLNF is their size stays smaller than $EP\varphi$ by the rewritings. The top level of the translation is in DNF, with subformulas in DLNF.

D Detailed Experimental Analysis

This appendix presents additional PCTL experiments that demonstrate the full expressiveness of the PoET tool beyond the PBTL-compatible properties shown in the main paper.

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: PCTL Properties demonstrating full temporal logic expressiveness

PCTL Properties: Full Expressiveness Evaluation

The following four properties in Figure 6 require operators beyond the PBTL subset (e.g., AY, $A(\phi S \psi)$) and showcase scenarios where complete PCTL expressiveness is useful. In particular, **Property 5** specifies transaction atomicity in a two-process system, ensuring mutual exclusion using AY and $A(\phi S \psi)$ operators. **Property 6** asserts the correspondence between two independent event sequences using modulo four binary counters. **Property 7** enforces synchronization ordering where process p_3 can only enter its critical section after p_1 or p_2.

[5] This is done as in building a truth table for a formula with the variables of the form $\hat{\varphi}_i$, checking which minterm satisfies all the disjuncts of the CNF.

PCTL **Experimental Results.** 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	**Metric**	**Trace 50**	**Trace 100**	**Trace 500**	**Trace 1000**
5	Time	0.44	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 forPCTL Properties

These results demonstrate that while PCTL enables expressing complex temporal relationships impossible in PBTL, the exponential complexity limits practical deployment to smaller systems, contrasting with PBTL's better scalability for online monitoring scenarios.