Runtime Verification of Partially-Synchronous Distributed System

Ritam Ganguly[1†], Anik Momtaz[1†] and Borzoo Bonakdarpour[1*]

[1]Department of Computer Science and Engineering, Michigan State University, East Lansing, 48823, sMichigan, USA.

*Corresponding author(s). E-mail(s): borzoo@msu.edu; Contributing authors: gangulyr@msu.edu; momtazan@msu.edu;
[†]These authors contributed equally to this work.

Abstract

This paper focuses on runtime verification of distributed systems in the *partial synchronous* model, where a clock synchronization algorithm ensures a bound on maximum clock skew among all processes. We introduce two centralized monitoring technique where the specification in the linear temporal logic (LTL) is either represented by a deterministic finite automaton, or, we use a progression-based formula rewitting technique to reduce the distributed runtime verification problem to an SMT problem. We report on rigorous synthetic, as well as real-world case studies involving Cassandra (a non-SQL database management system) and data available from RACE (Runtime for Airspace Concept Evaluation) by NASA. We show that both our automata-based as well as our progression-based approached are effective in evaluating all the possible verdicts given a distributed computation with the progression-based approach having less overhead in general.

Keywords: Runtime Verification, Monitoring, Distributed Computation, Partially Synchronous, Cassandra

1 Introduction

Distributed monitoring consists of evaluating the execution of a distributed application with a centralized or decentralized monitor with respect to a formal specification. A distributed application typically comprises of multiple processes that do not share a global clock and memory, while attempting to accomplish a joint task. For example, in a distributed database, data is stored across different physical locations, possibly dispersed over a network of interconnected computers, and a monitor may want to ensure that queries to the database satisfy some type of consistency criteria.

Motivation

The main challenge with distributed monitoring lies within the fact that in the absence of a global clock, it is not always possible for the monitor to establish the correct order of occurrence of events across different processes. In fact, given the non-deterministic nature of distributed applications, it is perfectly foreseeable that a runtime monitor may produce different verdicts for the same distributed computation based on different ordering of events. In the case of complete asynchrony, this in turn results in a combinatorial blow-up of possibilities that the monitor must explore at run time, which in turn makes the problem computationally expensive. However, state-of-the-art networks, such as Google Spanner are augmented with clock synchronization techniques that result in partial-synchrony [1]. These clock synchronization techniques guarantee a maximum clock-skew of ε between any pair of processes. Having such a guarantee considerably limits the combinatorial blow-up, as events outside the window of ε can be ordered.

To give an example of the blow-up experienced by the monitor, consider Figure 1, where we have two processes P_1 and P_2 hosting two discrete variables x_1 and x_2, respectively. Let us also consider the linear temporal logic (LTL) property $\varphi = \bigcirc(x_2 > x_1)$ and a maximum clock-skew, also known as clock-synchronization constant, to be $\varepsilon = 2$. Events $x_1 = 1$ and $x_2 = 0$, as well as $x_1 = 0$ and $x_2 = 2$, are not considered concurrent, as the events in these event pairs are more than ε time apart. However, events $x_1 = 1$ and $x_2 = 2$ are considered concurrent, as these events occurred within ε time from one another. Therefore, it is not possible to determine the exact ordering of these events, without a global clock. Thus, the formula gets evaluated to both true and false, as both possible ordering of events must be taken into account. The

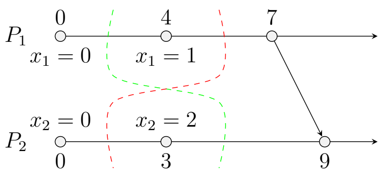


Fig. 1 Distributed computation.

number of different possible ordering of events can increase dramatically as more events and processes are introduced.

Handling concurrent events generally results in combinatorial enumeration of all possibilities and, hence, intractability of distributed RV. Existing distributed RV techniques operate in two extremes: they either assume a global clock [2], which is unrealistic for large-scale distributed settings or assume complete asynchrony [3, 4], which do not scale well. To further elaborate on our point, consider the processes P_1 and P_2 in Fig. 2, with events $\{e_0^1, e_1^1, e_2^1, e_3^1, e_4^1, e_5^1\}$ on process P_1, and events $\{e_0^2, e_1^2, e_2^2, e_3^2, e_4^2\}$ on process P_2 divided into two segments, seg_1 and seg_2, and a LTL formula,

$$\varphi = \bigcirc \left(\Diamond r \rightarrow (\neg p \, \mathcal{U} \, r) \right).$$

Observe that the predicate p (resp. r) is true at events e_0^2 and e_4^2 (resp. e_4^1), and in the rest of the events both predicates are false, denoted by $\emptyset$. The scenario where e_0^2 happens before e_0^1 and e_4^1 happens before e_4^2, the LTL property, φ, is satisfied. However, the scenario where e_0^1 happens before e_0^2 and e_4^1 happens after e_4^2, violates φ.

Thus, following the above example, the main research problem we aim to tackle in this paper is the following. Given a finite distributed computation and an LTL formula, our objective is to design efficient algorithms that determine whether or not the computation satisfies the formula. As shown above, the main obstacle is solving this problem is the explosion of interleavings at run time that need to be explored in order to monitor a computation.

Contributions

In order to address the combinatorial explosion of various interleavings introduced by the absence of a global clock, our first design choice is a practical assumption, namely, a bounded skew of ε between local clocks of each pair of processes, which is guaranteed by a clock synchronization mechanism (e.g., NTP [5]).

Our first technique is based on constructing the LTL_3 [6] monitor automaton of an LTL formula and constructing multiple SMT queries to determine which states of the monitor automaton are reachable for a given distributed computation. For example, Fig. 3 shows the monitor automaton for formula φ mentioned earlier and one has to construct 4 different SMT queries to determine the set of all possible reachable states at the end of the computation in

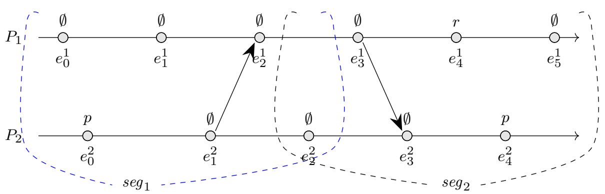


Fig. 2 Distributed computation.

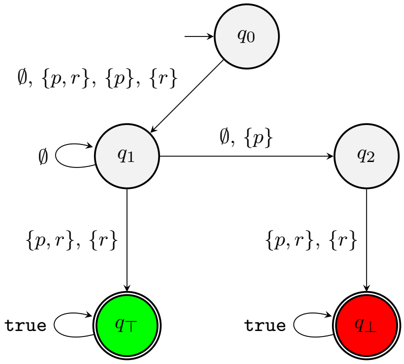
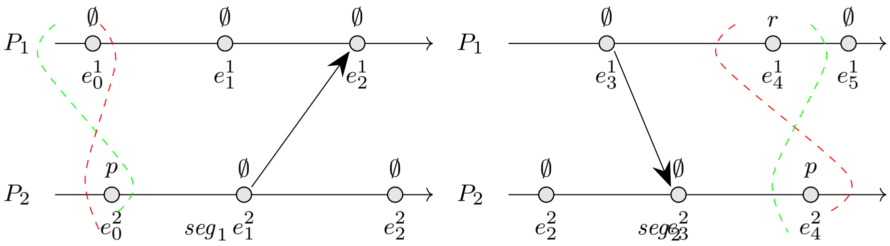


Fig. 3 Monitori automaton for formula φ.

Fig. 2. We transform our monitoring decision problem into an SMT solving problem. The SMT instance includes constraints that encode (1) our monitoring algorithm based on the 3-valued semantics of LTL [6], (2) behavior of communicating processes and their local state changes in terms of a distributed computation, and (3) the happened-before relation subject to the ϵ clock skew assumption. Then, it attempts to concretize an uninterpreted function whose evaluation provides the possible verdicts of the monitor with respect to the given computation. In order to make the verification problem tractable, we chop a computation into multiple segments and effectively reduce the search space of each SMT query (see Fig. 4). Thus, the result of monitoring each segment (the possible LTL_3 states) should be carried to the next segment. Furthermore, given the fact that distributed applications nowadays run on massive cloud services, we extend our solution to a parallel monitoring algorithm to utilize the available computing infrastructure and achieve better scalability.

The intuition behind our second monitoring technique is that since (in the first approach) running SMT queries to test whether each state of the LTL_3 monitor automaton is reachable is excessive, it should be sufficient to test whether temporal sub-formulas of an LTL formula hold in a distributed computation. Similar to the first approach, we utilize segmentation, to break down the problem size. In the second, approach to carry the result of monitroing from one segment to the next, we also develop a *formula progression* technique. Specifically, given a finite trace α, and an LTL formula φ, we define a function Pr, such that $\mathsf{Pr}(\alpha, \varphi)$ characterizes the *progression* of φ and α. Progression is defined as the rewritten formula for future extensions of α depending on what has been observed thus far, which returns either `true`, `false`, or an LTL formula. We emphasize that the main difference between our technique and the classic rewriting technique [7] is that, function Pr takes a finite trace as input while the algorithm in [7] rewrites the input LTL formula in a state-by-state manner. This means that in our setting, rewriting based on the fixed point representation of temporal operators is not possible. Our motivation is due to the fact that when a given distributed computation is chopped into a number of segments then a state-by-state rewriting approach would incur too many



$$\varphi = \bigcirc\,(\Diamond\, r \rightarrow (\neg p\, \mathcal{U}\, r))$$

Fig. 4 Progression and segmentation.

SMT queries, making it unscalable. For example, in Fig. 4 (which is the computation in Fig. 2 chopped to two segments), our progression-based approach needs the same 4 SMT queries for seg_1 (2 for each of the sub-formulas $\Diamond\, r$ and $\Box(\neg p)$) as compared to [8]. The evaluation yields formulas $\neg(\Diamond\, r)$ and $\Diamond\, r \rightarrow (\neg p\, \mathcal{U}\, r)$ as the possible formulas and as a result we only need to build 4 SMT queries in seg_2 compared to 5 for the automata-based approach in [8].

Our method is fully implemented and the datasets generated during and/or analysed during the current study are available in https://github.com/TART-MSU/dist-ltl-rv. We make a detailed comparison between the proposed approaches in this paper through not only a set of vigorous synthetic experiments, but also monitoring the same set of consistency conditions in Cassandra. We also put our approach to test using a real-time airspace monitoring dataset (RACE) from NASA [9]. Our experiments show that the progression-based approach has 35% reduced overhead (See Section 6 as compared to the automata-based approach.

In summary, the main contributions of this paper is as follows:

- We transform our monitoring decision problem into an SMT[1] problem, to make for an efficient yet correct approach to consider different interleavings. Given an LTL formula, our solution provides all possible verdicts on a given computation.
- We present two monitoring approaches to address the challenges (mentioned earlier) of distributed runtime verification with regard to LTL formulas under a partially synchronous setting. In our first approach, we keep track of the observed events and the possible future outcomes by employing an automata-based technique. In our second approach, we employ a more efficient progression-based technique, where we rewrite the given LTL specifications based on the current observations. For both of our approaches, we consider a fault-proof central monitor.
- We divide a given computation into multiple segments in order to make the verification problem tractable, and as a result, significantly reduce the search space of each SMT query. Furthermore, we parallelize our monitoring

[1]Satisfiability modulo theories (SMT) is the problem of determining whether a formula involving Boolean expressions comprising of more complex formulas involving real numbers, integers, and/or various data structures is satisfiable.

technique in order to utilize the available computational resources and gain greater scalability.
- Finally, we explore and report on extensive comparisons between our automata-based approach and our progression-based approach in terms of runtime and complexity.

Comparison to the conference submission

A preliminary version of this paper appeared in the 2020 International Conference on Principles of Distributed Systems (OPODIS) [8]. The OPODIS'20 paper only included the automata-based approach. This paper extends the OPODIS'20 paper in multiple fronts. First, the progression-based technique is completely new. This includes the notion of progression functions for LTL and its SMT formulation. This technique is also fully implemented and its performance is rigorously analzyed and compared with the original automata-based approach.

Organization

Section 2 presents the background concepts and the problem statement. We go over the theory of formula progression in Section 3 and discuss SMT-based approach both of the automata-based solution and the progression-based solution in Section 4. We introduce some optimization approaches to yield better run time in Section 5, while we go over the analysis of experimental results in Section 6. Related work is discussed in Section 7. Finally, we make concluding remarks in Section 8.

2 Preliminaries and Problem Statement

2.1 Linear Temporal Logics (LTL) for RV

Let AP be a set of *atomic propositions* and $\Sigma = 2^{\mathsf{AP}}$ be the set of all possible *states*. A *trace* is a sequence $s_0 s_1 \ldots$, where $s_i \in \Sigma$ for every $i \geq 0$. We denote by Σ^* (resp., Σ^ω) the set of all finite (resp., infinite) traces. For a finite trace $\alpha = s_0 s_1 \ldots s_k$, $|\alpha|$ denotes its *length*, $k+1$. Also, for $\alpha = s_0 s_1 \ldots s_k$, by α^i, we mean trace $s_i s_{i+1} \ldots s_k$ of α.

2.1.1 Infinite-trace Semantics of LTL

The syntax and semantics of the *linear temporal logic* (LTL) [10] are defined for infinite traces. The syntax is defined by the following grammar:

$$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid \bigcirc\varphi \mid \varphi \,\mathcal{U}\, \varphi$$

where $p \in \mathsf{AP}$, and where $\bigcirc$ and $\mathcal{U}$ are the 'next' and 'until' temporal operators respectively. We view other propositional and temporal operators as abbreviations, that is, $\mathtt{true} = p \vee \neg p$, $\mathtt{false} = \neg\mathtt{true}$, $\varphi \rightarrow \psi = \neg\varphi \vee \psi$,

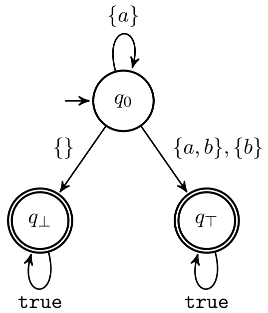


Fig. 5 LTL_3 monitor for $\varphi = a\ \mathcal{U}\ b$.

$\varphi \wedge \psi = \neg(\neg\varphi \vee \neg\psi)$, $\Diamond\varphi = \mathtt{true}\ \mathcal{U}\ \varphi$ (*eventually* φ), and $\Box\varphi = \neg\Diamond\neg\varphi$ (*always* φ). We denote the set of all LTL formulas by Φ_{LTL}.

The infinite-trace semantics of LTL is defined as follows. Let $\sigma = s_0 s_1 s_2 \cdots \in \Sigma^\omega$, $i \geq 0$, and let $\models$ denote the *satisfaction* relation:

$$\begin{array}{lll}
\sigma, i \models p & \text{iff} & p \in s_i \\
\sigma, i \models \neg\varphi & \text{iff} & \sigma, i \not\models \varphi \\
\sigma, i \models \varphi_1 \vee \varphi_2 & \text{iff} & \sigma, i \models \varphi_1 \ \text{ or } \ \sigma, i \models \varphi_2 \\
\sigma, i \models \bigcirc\varphi & \text{iff} & \sigma, i+1 \models \varphi \\
\sigma, i \models \varphi_1\ \mathcal{U}\ \varphi_2 & \text{iff} & \exists k \geq i : \sigma, k \models \varphi_2 \ \text{ and } \ \forall j \in [i, k) : \sigma, j \models \varphi_1
\end{array}$$

Also, $\sigma \models \varphi$ holds if and only if $\sigma, 0 \models \varphi$ holds.

2.1.2 Finite-trace Semantics of LTL

In the context of RV, the 3-valued LTL (LTL_3 for short) [6] evaluates LTL formulas for *finite* traces, but with an eye on possible future extensions where as finite LTL, or FLTL [11] only takes into consideration the current trace with no eye towards the future. In LTL_3, the set of truth values is $\mathbb{B}_3 = \{\top, \bot, ?\}$, where $\top$ (resp., $\bot$) denotes that the formula is *permanently* satisfied (resp., violated), no matter how the current finite trace extends, and '?' denotes an *unknown* verdict, i.e., there exists an extension that can violate the formula, and another extension that can satisfy the formula. Let $\alpha \in \Sigma^*$ be a non-empty finite trace. The truth value of an LTL_3 formula φ with respect to α, denoted by $[\alpha \models_3 \varphi]$, is defined as follows:

$$[\alpha \models_3 \varphi] = \begin{cases} \top & \text{if} \quad \forall \sigma \in \Sigma^\omega : \alpha\sigma \models \varphi \\ \bot & \text{if} \quad \forall \sigma \in \Sigma^\omega : \alpha\sigma \not\models \varphi \\ ? & \text{otherwise.} \end{cases}$$

Definition 1 The LTL_3 *monitor* for a formula φ is the unique deterministic finite state machine $\mathcal{M}_\varphi = (\Sigma, Q, q_0, \delta, \lambda)$, where Q is the set of states, q_0 is the initial state, $\delta : Q \times \Sigma \to Q$ is the transition function, and $\lambda : Q \to \mathbb{B}_3$ is a function such that $\lambda\big(\delta(q_0, \alpha)\big) = [\alpha \models_3 \varphi]$, for every finite trace $\alpha \in \Sigma^*$. ■

For example, Fig. 5, shows the monitor automaton for formula $\varphi = a\,\mathcal{U}\,b$. The syntax of FLTL is also identical to that of LTL, and the semantics is based on the truth values $\mathbb{B}_2 = \{\top, \bot\}$, where $\top$ (resp., $\bot$) denotes that the formula is satisfied (resp., violated) given the current finite trace. For atomic propositions and Boolean operators, the semantics of FLTL is identical to those of LTL. Let φ, φ_1, and φ_2 be LTL formulas, $\alpha = s_0 s_1 \ldots s_n$ be a *non-empty* finite trace, and $\models_F$ denote the satisfaction relation in FLTL. The semantics of FLTL for the temporal operators are as follows:

$$[\alpha \models_F \bigcirc\varphi] = \begin{cases} [\alpha^1 \models_F \varphi] & \text{if } \ \alpha^1 \neq \epsilon \\ \bot & \text{otherwise.} \end{cases}$$

$$[\alpha \models_F \varphi_1 \mathcal{U} \varphi_2] = \begin{cases} \top & \text{if } \ \exists k \in [0,n] : ([\alpha^k \models_F \varphi_2] = \top) \wedge \\ & \qquad \forall l \in [0,k) : ([\alpha^l \models_F \varphi_1] = \top) \\ \bot & \text{otherwise.} \end{cases}$$

Remark 1 It is to be noted that the semantics of LTL, FLTL, and LTL_3 are not interchangeable as is demonstrated below:
Consider a formula $\varphi = \Box p$, and a finite trace $\alpha = s_0 s_1 \cdots s_n$. If $p \notin s_i$ for some $i \in [0,n]$, then $[\alpha \models_3 \varphi] = \bot$, that is, the formula is permanently violated and so is the case in FLTL where, $[\alpha \models_F \varphi] = \bot$. Now, consider formula $\varphi = \Diamond p$. If $p \notin s_i$ for all $i \in [0,n]$, then $[\alpha \models_3 \varphi] = ?$. This is because there exist infinite extensions to α that can satisfy or violate φ in the infinite semantics of LTL. But, this is not the case in FLTL where $[\alpha \models_F \varphi] = \bot$ as it did not observe any p in the observed finite trace.

2.2 Distributed Computations

We assume a loosely coupled asynchronous message passing system, consisting of n reliable (that do not fail) processes, denoted by $\mathcal{P} = \{P_1, P_2, \ldots, P_n\}$, without any shared memory or global clock. Channels are assumed to be FIFO, and lossless. In our model, each local state change is considered an event, and every message activity (send or receive) is also represented by a new event. Message transmission does not change the local state of processes and the content of a message is immaterial to our purposes. We will need to refer to some global clock which acts as a '*real*' timekeeper. It is to be understood, however, that this global clock is a theoretical object used in definitions, and is *not* available to the processes.

We make a practical assumption, known as *partial synchrony*. The *local clock* (or time) of a process P_i, where $i \in [1,n]$, can be represented as an increasing function $c_i : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$, where $c_i(\chi)$ is the value of the local clock at global time χ. Then, for any two processes P_i and P_j, we have:

$$\forall \chi \in \mathbb{R}_{\geq 0}. |c_i(\chi) - c_j(\chi)| < \epsilon$$

with $\epsilon > 0$ being the maximum *clock skew*. The value ϵ is assumed to be fixed and known by the monitor in the rest of this paper. In the sequel, we make it explicit when we refer to 'local' or 'global' time. This assumption is met by using a clock synchronization algorithm, like NTP [5], to ensure bounded clock skew among all processes.

An *event* in process P_i is of the form $e^i_{\tau,\sigma}$, where σ is *logical time* (i.e., a natural number) and τ is the local time at global time χ, that is, $\tau = c_i(\chi)$. We assume that for every two events $e^i_{\tau,\sigma}$ and $e^i_{\tau',\sigma'}$, we have $(\tau < \tau') \Leftrightarrow (\sigma < \sigma')$.

Definition 2 A *distributed computation* on N processes is a tuple $(\mathcal{E}, \rightsquigarrow)$, where $\mathcal{E}$ is a set of events partially ordered by Lamport's *happened-before* ($\rightsquigarrow$) relation [12], subject to the partial synchrony assumption:

- In every process P_i, $1 \leq i \leq N$, all events are totally ordered, that is,

$$\forall \tau, \tau' \in \mathbb{R}_+ . \forall \sigma, \sigma' \in \mathbb{Z}_{\geq 0} . (\sigma < \sigma') \rightarrow (e^i_{\tau,\sigma} \rightsquigarrow e^i_{\tau',\sigma'}).$$

- If e is a message send event in a process, and f is the corresponding receive event by another process, then we have $e \rightsquigarrow f$.
- For any two processes P_i and P_j, and any two events $e^i_{\tau,\sigma}, e^j_{\tau',\sigma'} \in \mathcal{E}$, if $\tau + \epsilon < \tau'$, then $e^i_{\tau,\sigma} \rightsquigarrow e^j_{\tau',\sigma'}$, where ϵ is the maximum clock skew.
- If $e \rightsquigarrow f$ and $f \rightsquigarrow g$, then $e \rightsquigarrow g$. ■

Definition 3 Given a distributed computation $(\mathcal{E}, \rightsquigarrow)$, a subset of events $C \subseteq \mathcal{E}$ is said to form a *consistent cut* iff when C contains an event e, then it contains all events that happened-before e. Formally, $\forall e \in \mathcal{E} . (e \in C) \wedge (f \rightsquigarrow e) \rightarrow f \in C$. ■

The *frontier* of a consistent cut C, denoted $\mathsf{front}(C)$ is the set of events that happen last in the cut. $\mathsf{front}(C)$ is a set of e^i_{last} for each $i \in [1, |\mathcal{P}|]$ and $e^i_{last} \in C$. We denote e^i_{last} as the last event in P_i such that $\forall e^i_{\tau,\sigma} \in \mathcal{E} . (e^i_{\tau,\sigma} \neq e^i_{last}) \rightarrow (e^i_{\tau,\sigma} \rightsquigarrow e^i_{last})$.

2.3 Problem Statement

Given a distributed computation $(\mathcal{E}, \rightsquigarrow)$, a *valid* sequence of consistent cuts is of the form $C_0 C_1 C_2 \cdots$, where for all $i \geq 0$, (1) C_i denotes a set of events included in the consistent cut, (2) C_i is a subset of its succeeding consistent cut, C_{i+1}, that is, $C_i \subset C_{i+1}$, and (3) C_{i+1} has one additional event compared to its preceding consistent cut C_i, that is, $|C_i| + 1 = |C_{i+1}|$. Let $\mathcal{C}$ denote the set of all valid sequences of consistent cuts. We define the set of all traces of $(\mathcal{E}, \rightsquigarrow)$ as follows:

$$\mathsf{Tr}(\mathcal{E}, \rightsquigarrow) = \Big\{ \mathsf{front}(C_0)\mathsf{front}(C_1) \cdots \mid C_0 C_1 C_2 \cdots \in \mathcal{C} \Big\}.$$

Now for our automata-based approach (resp. progression-based approach), the evaluation of the LTL formula φ with respect to $(\mathcal{E}, \rightsquigarrow)$ in the 3-valued semantics (resp. finite semantics) is the following:

$$[(\mathcal{E}, \rightsquigarrow) \models_3 \varphi] = \left\{ \alpha \models_3 \varphi \mid \alpha \in \mathsf{Tr}(\mathcal{E}, \rightsquigarrow) \right\}$$

and

$$[(\mathcal{E}, \rightsquigarrow) \models_F \varphi] = \left\{ \alpha \models_F \varphi \mid \alpha \in \mathsf{Tr}(\mathcal{E}, \rightsquigarrow) \right\}$$

respectively. This means evaluating a distributed computation with respect to a formula results in a *set* of verdicts, as a computation may involve several traces.

2.4 Hybrid Logical Clocks

A *hybrid logical clock* (HLC) [13] is a tuple (τ, σ, ω) for detecting one-way causality, where τ is the local time, σ ensures the order of send and receive events between two processes, and ω indicates causality between events. Thus, in the sequel, we denote an event by $e^i_{\tau,\sigma,\omega}$. More specifically, for a set $\mathcal{E}$ of events:

- τ is the local clock value of events, where for any process P_i and two events $e^i_{\tau,\sigma,\omega}, e^i_{\tau',\sigma',\omega'} \in \mathcal{E}$, we have $\tau < \tau'$ iff $e^i_{\tau,\sigma,\omega} \rightsquigarrow e^i_{\tau',\sigma',\omega'}$.
- σ stipulates the logical time, where:
  - For any process P_i and any event $e^i_{\tau,\sigma,\omega} \in \mathcal{E}$, τ never exceeds σ, and their difference is bounded by ϵ (i.e, $\sigma - \tau \leq \epsilon$).
  - For any two processes P_i and P_j, and any two events $e^i_{\tau,\sigma,\omega}, e^j_{\tau',\sigma',\omega'} \in \mathcal{E}$, where event $e^i_{\tau,\sigma,\omega}$ receiving a message sent by event $e^j_{\tau',\sigma',\omega'}$, σ is updated to $\max\{\sigma, \sigma', \tau\}$. The maximum of the three values are chosen to ensure that σ remains updated with the largest τ observed so far. Observe that σ has similar behavior as τ, except the communication between processes has no impact on the value of τ for an event.
- $\omega : \mathcal{E} \rightarrow \mathbb{Z}_{\geq 0}$ is a function that maps each event in $\mathcal{E}$ to the causality updates, where:
  - For any process P_i and a send or local event $e^i_{\tau,\sigma,\omega} \in \mathcal{E}$, if $\tau < \sigma$, then ω is incremented. Otherwise, ω is reset to 0.
  - For any two processes P_i and P_j and any two events $e^i_{\tau,\sigma,\omega}, e^j_{\tau',\sigma',\omega'} \in \mathcal{E}$, where event $e^i_{\tau,\sigma,\omega}$ receiving a message sent by event $e^j_{\tau',\sigma',\omega'}$, $\omega(e^i_{\tau,\sigma,\omega})$ is updated based on $\max\{\sigma, \sigma', \tau\}$.
  - For any two processes P_i and P_j, and any two events $e^i_{\tau,\sigma,\omega}, e^j_{\tau',\sigma',\omega'} \in \mathcal{E}$, $(\tau = \tau') \wedge (\omega < \omega') \rightarrow e^i_{\tau,\sigma,\omega} \rightsquigarrow e^j_{\tau',\sigma',\omega'}$.

HLC is susceptible to faults that might be present in the system, such as missing clock-synchronization messages or processes changing their local clock

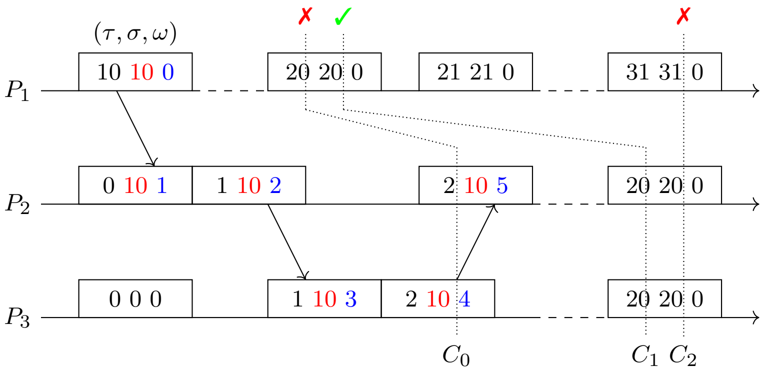


Fig. 6 HLC example.

values. We assume our system is free of such faults. Fig. 6 shows an HLC incorporated partially synchronous concurrent timelines of three processes with $\varepsilon = 10$. Observe that the local times of all events in $\mathsf{front}(C_1)$ are bounded by ε. Therefore, C_1 is a consistent cut, but C_0 and C_2 are not.

3 Formula Progression for LTL

In a synchronous system, verification on a computation can be performed in a state by state approach due to the existence of a total ordering of events [14]. However, in a *partially* synchronous system, no such total ordering of events is possible. A distributed computation $(\mathcal{E}, \rightsquigarrow)$ may have different partial ordering of events dictated by different interleaving of events. Therefore, it is possible to obtain multiple verdicts on the same distributed computation $(\mathcal{E}, \rightsquigarrow)$. In order to explore these verdicts, we propose a monitoring approach based on *formula progression* that, if possible, *partially* evaluates a formula on the current computation, and based on the verdict, provides a *rewritten* formula that is to be evaluated on the extensions of the computation. As an example, let us consider the formula to be monitored as, $\varphi = \Diamond(a \rightarrow \Diamond b)$. Now, if in some trace in a computation, the monitor observes a, then for the extensions of computations, it is enough to monitor the rewritten formula, $\varphi' = \Diamond b$, as the final verdict is no longer dependent on the occurrence of a. We call this method of rewriting formula **Progression**, which we discuss in length in the following section.

Definition 4 A *progression function* $\mathsf{Pr} : \Sigma^* \times \Phi_{\mathsf{LTL}} \rightarrow \Phi_{\mathsf{LTL}}$ is one that for all finite traces $\alpha \in \Sigma^*$, infinite traces $\sigma \in \Sigma^\omega$, and formulas $\varphi \in \Phi_{\mathsf{LTL}}$, we have: $\alpha\sigma \models \varphi$ iff and only if $\sigma \models \mathsf{Pr}(\alpha, \varphi)$. ■

We emphasize that the main difference between our technique and the classic rewriting technique [7] is that, function Pr takes a finite traces as input, while the algorithm in [7] rewrite the input LTL formula in a state-by-state manner. This means that rewriting based on the fixed point representation of temporal operators is not possible. The motivation for our approach comes from the fact the a given distributed computation is chopped into a number of *segments*, and verification of each segment is handled by an SMT query. A state by state approach would incur too many SMT queries, making it unscalable.

Remark 2 It is straightforward to see that for any $\alpha \in \Sigma^*$ and $\varphi \in \Phi$, if a progression function returns a non-trivial formula, which we denote by $\mathsf{Pr}(\alpha, \varphi) = \varphi'$ for some $\varphi' \in \Phi_{\mathsf{LTL}}$, then the verdict of monitoring is *unknown*.

Atomic propositions. Let $\varphi = p$ for some $p \in \mathsf{AP}$. The verdict is provided depending upon whether or not $p \in \alpha(0)$. This is the only case where the output of Pr cannot be a rewritten formula; the possible verdicts are either `true` or `false`:

$$\mathsf{Pr}(\alpha, \varphi) = \begin{cases} \texttt{true} & \text{if} \quad p \in \alpha(0) \\ \texttt{false} & \text{if} \quad p \notin \alpha(0) \end{cases}$$

Negation. Let $\varphi = \neg\phi$. We have $\mathsf{Pr}(\alpha, \varphi) = \neg\mathsf{Pr}(\alpha, \phi)$.

Disjunction. Let $\varphi = \varphi_1 \vee \varphi_2$. If either sub-formula φ_1 or φ_2 is evaluated to `false`, then the progression of φ becomes the other sub-formula φ_2 or φ_1 respectively, since that will be the only responsible sub-formula for the verdict of all future computations:

$$\mathsf{Pr}(\alpha, \varphi) = \begin{cases} \texttt{true} & \text{if} \quad \mathsf{Pr}(\alpha, \varphi_1) = \texttt{true} \ \vee \ \mathsf{Pr}(\alpha, \varphi_2) = \texttt{true} \\ \texttt{false} & \text{if} \quad \mathsf{Pr}(\alpha, \varphi_1) = \texttt{false} \ \wedge \ \mathsf{Pr}(\alpha, \varphi_2) = \texttt{false} \\ \varphi'_2 & \text{if} \quad \mathsf{Pr}(\alpha, \varphi_1) = \texttt{false} \ \wedge \ \mathsf{Pr}(\alpha, \varphi_2) = \varphi'_2 \\ \varphi'_1 & \text{if} \quad \mathsf{Pr}(\alpha, \varphi_2) = \texttt{false} \ \wedge \ \mathsf{Pr}(\alpha, \varphi_1) = \varphi'_1 \\ \varphi'_1 \vee \varphi'_2 & \text{if} \quad \mathsf{Pr}(\alpha, \varphi_1) = \varphi'_1 \ \wedge \ \mathsf{Pr}(\alpha, \varphi_2) = \varphi'_2 \end{cases}$$

Next operator. Let $\varphi = \bigcirc\phi$. The verdicts `true`, `false` and ϕ' can only be reached if α^1 is not an empty trace, that is, $\mid \alpha^1 \mid \neq 0$. Otherwise, if we are at the last event in the trace, then the progression of φ becomes ϕ; implying ϕ must hold at the beginning of the future extension:

$$\mathsf{Pr}(\alpha, \varphi) = \begin{cases} \texttt{true} & \text{if} \quad \mathsf{Pr}(\alpha^1, \phi) = \texttt{true} \wedge \mid \alpha^1 \mid \neq 0 \\ \texttt{false} & \text{if} \quad \mathsf{Pr}(\alpha^1, \phi) = \texttt{false} \wedge \mid \alpha^1 \mid \neq 0 \\ \phi' & \text{if} \quad \mathsf{Pr}(\alpha^1, \phi) = \phi' \wedge \mid \alpha^1 \mid \neq 0 \\ \phi & \text{if} \quad \mid \alpha^1 \mid = 0 \end{cases}$$

Always and eventually operators. Progression in the temporal operator 'always', $\Box$ (resp. 'eventually', $\Diamond$) may yield `false` (resp. `true`) or remain unchanged:

$$\mathsf{Pr}(\alpha, \varphi) = \begin{cases} \texttt{false} & \text{if} \quad [\alpha \models_F \varphi] = \bot \\ \Box \phi & \text{if} \quad \text{otherwise} \end{cases}$$

$$\mathsf{Pr}(\alpha, \varphi) = \begin{cases} \texttt{true} & \text{if} \quad [\alpha \models_F \varphi] = \top \\ \Diamond \phi & \text{if} \quad \text{otherwise} \end{cases}$$

Note that the semantics of FLTL is not frequently used, due to LTL_3 being generally more expressive, as shown in [15]. However, LTL_3 cannot be used to construct the progression rules. To be more precise, the '?' (*unknown*) verdict in LTL_3 semantics would raise additional and unnecessary complications in the progression rules, as this verdict does not provide any additional information as far as our progression-based approach is concerned. In fact, if progression results in a formula, it represents the '?' verdict in LTL_3. Therefore, we use FLTL for specifying the progression rules without any loss of generality as shown later in the proof of Lemma 1.

Until operator. Let $\varphi = \varphi_1 \,\mathcal{U}\, \varphi_2$. Recall that $\varphi_1 \,\mathcal{U}\, \varphi_2 = \varphi_2 \vee (\varphi_1 \wedge \bigcirc(\varphi_1 \,\mathcal{U}\, \varphi_2))$. We divide the $\mathcal{U}$ formula into two parts, one with globally ($\Box \varphi_1$) and the other eventuality ($\Diamond \varphi_2$). These sub-formulas are evaluated separately and the verdict of each of them is used to define the progression for the $\mathcal{U}$ operator. However, for the case when both φ_1 and φ_2 occur in the same computation, we cannot come to a verdict without considering the order of satisfaction of these sub-formulas. That is, on a given finite trace α, if φ_2 holds in $\alpha(i)$ (denoted $\Diamond_i \varphi_2$) and φ_1 holds throughout in all states from $\alpha(0)$ to $\alpha(i-1)$ (denoted $\Box_{i-1} \varphi_1$), then the progression of φ becomes $\texttt{true}$. If this is not the case, and $\Box \varphi_1$ does not hold in α, the progression of φ becomes $\texttt{false}$, since this signifies a break from the streak of φ_1 required for φ to hold. If it is neither of the above two cases, and the evaluated verdict of $\Diamond \mathsf{Pr}(\alpha, \varphi_2)$ is $\top$, then this represents a case where we do not have enough information about φ_1 to evaluate $\varphi_1 \,\mathcal{U}\, \varphi_2$. Thus, making the progression solely dependant on φ_1. The progression of φ remains unchanged if φ_1 holds throughout α, but φ_2 does not hold anywhere:

$$\mathsf{Pr}(\alpha, \varphi) = \begin{cases} \texttt{true} & \text{if } \exists i \in [0, |\alpha|-1] \,.\, [\alpha \models_F \Diamond_i \mathsf{Pr}(\alpha, \varphi_2)] = \top \\ & \quad \wedge\ [\alpha \models_F \Box_{i-1} \mathsf{Pr}(\alpha, \varphi_1)] = \top \\ \texttt{false} & \text{if } [\alpha \models_F \Box \mathsf{Pr}(\alpha, \varphi_1)] = \bot \\ & \quad \wedge\ \text{not the first case} \\ \mathsf{Pr}(\alpha, \varphi_1) & \text{if } [\alpha \models_F \Diamond \mathsf{Pr}(\alpha, \varphi_2)] = \top \\ & \quad \wedge\ \text{not the second case} \\ \mathsf{Pr}(\alpha, \varphi_1) \ \mathcal{U}\ \mathsf{Pr}(\alpha, \varphi_2) & \text{if } [\alpha \models_F \Box \mathsf{Pr}(\alpha, \varphi_1)] = \top \\ & \quad \wedge\ [\alpha \models_F \Diamond \mathsf{Pr}(\alpha, \varphi_2)] = \bot \end{cases}$$

Example. Consider the formula, $\varphi = \Diamond r \to (\neg p \,\mathcal{U}\, q)$ with sub-formulas $\varphi_s = \{\Diamond r,\ q,\ \Diamond q,\ \Box p\}$, according to our progression rules. Consider the

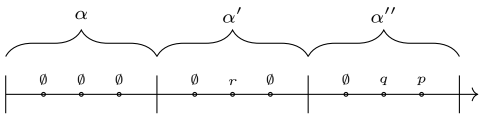


Fig. 7 Progression example.

trace in Fig. 7 divided into three segments. In the first segment α, neither p, q nor r are present, and as far as the laws of the progression function defined above, φ remains unchanged for the next segment; i.e., $\mathsf{Pr}(\alpha, \varphi) = \varphi$. In the second segment α', proposition r is observed, this satisfies sub-formula $\Diamond r$ the progressed formula becomes $\neg p \,\mathcal{U}\, q$; i.e., $\mathsf{Pr}(\alpha', \varphi) = \neg p \,\mathcal{U}\, q$. In the next segment α'', proposition q occurs before p. This falls under the first case of the until progression operator. Since q happens after a streak of $\neg p$, we arrive at the verdict $\texttt{true}$; i.e., $\mathsf{Pr}(\alpha'', \neg p \,\mathcal{U}\, q) = \texttt{true}$. Put it another way, $\mathsf{Pr}(\alpha\alpha'\alpha'', \varphi) = \texttt{true}$.

Lemma 1 Given an LTL formula φ, and finite traces $\alpha, \sigma \in \Sigma^*$, trace $\alpha\sigma$ satisfies φ if and only if σ satisfies $\mathsf{Pr}(\alpha, \varphi)$, as defined in above. Formally,

$$[\alpha\sigma \models_F \varphi] \iff [\sigma \models_F \mathsf{Pr}(\alpha, \varphi)]$$

Proof We distinguish the following cases:

Case 1: First, we consider the base case of this proof, where the formula is an atomic proposition, that is, $\varphi = p$.

($\Rightarrow$) Let us first consider that p is observed on the first state of $\alpha\sigma$. This implies, $[\alpha\sigma \models_F \varphi]$ yields $\texttt{true}$, and $\mathsf{Pr}(\alpha, \varphi)$ yields $\top$. Therefore, $[\sigma \models_F \mathsf{Pr}(\alpha, \varphi)]$ must also yield $\texttt{true}$.

Now, let us consider that p is not observed on the first state of $\alpha\sigma$. This implies, $[\alpha\sigma \models_F \varphi]$ yields $\texttt{false}$, and $\mathsf{Pr}(\alpha, \varphi)$ yields $\bot$. Therefore, $[\sigma \models_F \mathsf{Pr}(\alpha, \varphi)]$ must also yield $\texttt{false}$.

($\Leftarrow$) Let us first consider that $[\sigma \models_F \mathsf{Pr}(\alpha, \varphi)]$ yields $\texttt{true}$. This implies, $\mathsf{Pr}(\alpha, \varphi)$ yields $\top$, and $[\alpha\sigma \models_F \varphi]$ yields $\texttt{true}$. Therefore, p must have been observed on the first state of $\alpha\sigma$.

Now, let us consider that $[\sigma \models_F \mathsf{Pr}(\alpha, \varphi)]$ yields $\texttt{false}$. This implies, $\mathsf{Pr}(\alpha, \varphi)$ yields $\bot$, and $[\alpha\sigma \models_F \varphi]$ yields $\texttt{false}$. Therefore, p must not have been observed on the first state of $\alpha\sigma$.

Case 2: Assume that the proof has been established for the case when the formula is $\varphi = \phi$. Now, we consider the case where the formula is $\varphi = \neg\phi$.

We can say $[\alpha\sigma \models_F \neg\phi]$ is equivalent to $\neg[\alpha\sigma \models_F \phi]$ according to the finite-trace semantics of LTL. We can also say $[\sigma \models_F \mathsf{Pr}(\alpha, \neg\phi)]$ is equivalent to

$[\sigma \models_F \neg\mathsf{Pr}(\alpha,\phi)]$ since $\mathsf{Pr}(\alpha,\neg\phi) = \neg\mathsf{Pr}(\alpha,\phi)$ is defined as a progression rule. Furthermore, $[\sigma \models_F \neg\mathsf{Pr}(\alpha,\phi)]$ is equivalent to $\neg[\sigma \models_F \mathsf{Pr}(\alpha,\phi)]$ according to the finite-trace semantics of LTL.

Based on our assumption, the proof has already been established for $[\alpha\sigma \models_F \phi] \iff [\sigma \models_F \mathsf{Pr}(\alpha,\phi)]$. Therefore, $\neg[\alpha\sigma \models_F \phi] \iff \neg[\sigma \models_F \mathsf{Pr}(\alpha,\phi)]$, and by extension, $[\alpha\sigma \models_F \neg\phi] \iff [\sigma \models_F \mathsf{Pr}(\alpha,\neg\phi)]$

Case 3: Assume that the proof has been established for the case when the formula is $\varphi = \phi$. Now, we consider the case where the formula is $\varphi = \bigcirc\phi$.

Let us first consider the case where the length of the trace α is 1, that is, $\mid \alpha \mid= 1$ and $\mid \alpha^1 \mid= 0$. In this particular case, $[\alpha\sigma \models_F \bigcirc\phi]$ is equivalent to $[\sigma \models_F \phi]$. Furthermore, $\mathsf{Pr}(\alpha,\bigcirc\phi) = \phi$; which implies, $[\sigma \models_F \mathsf{Pr}(\alpha,\bigcirc\phi)]$ is equivalent to $[\sigma \models_F \phi]$. Therefore, $[\alpha\sigma \models_F \bigcirc\phi] \iff [\sigma \models_F \mathsf{Pr}(\alpha,\bigcirc\phi)]$.

Now, let us consider the case where the length of the trace α is longer than 1, that is, $\mid \alpha \mid\geq 1$ and $\mid \alpha^1 \mid\geq 1$. In this case, $[\alpha\sigma \models_F \bigcirc\phi]$ is equivalent to $[\alpha^1\sigma \models_F \phi]$, and $[\sigma \models_F \mathsf{Pr}(\alpha,\bigcirc\phi)]$ is equivalent to $[\sigma \models_F \mathsf{Pr}(\alpha^1,\phi)]$.

Based on our assumption, the proof has already been established for $[\alpha^1\sigma \models_F \phi] \iff [\sigma \models_F \mathsf{Pr}(\alpha^1,\phi)]$. Therefore, $[\alpha\sigma \models_F \bigcirc\phi] \iff [\sigma \models_F \mathsf{Pr}(\alpha,\bigcirc\phi)]$.

Case 4: Assume that the proof has been established for the cases when the formulas are $\varphi = \varphi_1$ and $\varphi = \varphi_2$. Now, we consider the case where the formula is $\varphi = \varphi_1 \vee \varphi_2$.

Based on our assumption, the proof has already been established for $[\alpha\sigma \models_F \phi_1] \iff [\sigma \models_F \mathsf{Pr}(\alpha,\phi_1)]$ and $[\alpha\sigma \models_F \phi_2] \iff [\sigma \models_F \mathsf{Pr}(\alpha,\phi_2)]$. Therefore, we can derive the following:

$$\begin{aligned}
[\alpha\sigma \models_F (\varphi_1 \vee \varphi_2)] &\iff [\alpha\sigma \models_F \varphi_1] \vee [\alpha\sigma \models_F \varphi_2] \\
&\iff [\sigma \models_F \mathsf{Pr}(\alpha,\varphi_1)] \vee [\sigma \models_F \mathsf{Pr}(\alpha,\varphi_2)] \\
&\iff [\sigma \models_F \mathsf{Pr}(\varphi_1 \vee \varphi_2)].
\end{aligned}$$

Case 5: Now, we consider the case where the formula is $\varphi = \varphi_1 \,\mathcal{U}\, \varphi_2$.
We prove this by induction on the length of α:

Base Case: $\mid \alpha \mid= 0$.

$$\begin{aligned}
[\alpha\sigma \models_F \varphi] &\iff [\sigma \models_F Pr(\alpha,\varphi)] \\
&\iff [\sigma \models_F \varphi]
\end{aligned}$$

Hypothesis Step: $\mid \alpha \mid= k$.

$$[\alpha\sigma \models_F \varphi_1 \,\mathcal{U}\, \varphi_2]$$

$$
\begin{aligned}
&\iff [\alpha\sigma \models_F \Big(\varphi_2 \vee (\varphi_1 \wedge \bigcirc(\varphi_1 \,\mathcal{U}\, \varphi_2))\Big)] \\
&\iff [\alpha\sigma \models_F \varphi_2] \vee [\alpha\sigma \models_F \Big(\varphi_1 \wedge \bigcirc(\varphi_1 \,\mathcal{U}\, \varphi_2)\Big)] \\
&\iff [\alpha\sigma \models_F \varphi_2] \vee \Big([\alpha\sigma \models_F \varphi_1] \wedge [\alpha^1\sigma \models_F \varphi_1 \,\mathcal{U}\, \varphi_2\Big)] \\
&\iff [\alpha\sigma \models_F \varphi_2] \vee \Bigg([\alpha\sigma \models_F \varphi_1] \wedge [\alpha^1\sigma \models_F \Big(\varphi_2 \vee (\varphi_1 \wedge \bigcirc(\varphi_1 \,\mathcal{U}\, \varphi_2))\Big)]\Bigg) \\
&\iff [\alpha\sigma \models_F \varphi_2] \vee ([\alpha\sigma \models_F \varphi_1] \wedge [\alpha^1\sigma \models_F \varphi_2]) \vee \ldots \vee \Big([\alpha\sigma \models_F \varphi_1] \wedge \\
&[\alpha^1\sigma \models_F \varphi_1] \wedge \ldots \wedge [\alpha^{k-2}\sigma \models_F \varphi_1] \wedge [(\alpha^{k-1}\sigma \models_F \varphi_2]\Big) \vee \\
&\Big([\alpha\sigma \models_F \varphi_1] \wedge \ldots \wedge [\alpha^{k-1}\sigma \models_F \varphi_1] \wedge [\alpha^k\sigma \models_F \varphi_1 \,\mathcal{U}\, \varphi_2]\Big) \\
&\iff [\alpha\sigma \models_F \varphi_2] \vee \Big([\alpha\sigma \models_F \varphi_1] \wedge [\alpha^1\sigma \models_F \varphi_2]\Big) \vee \ldots \vee \Big([\alpha\sigma \models_F \varphi_1] \wedge \\
&\ldots \wedge [\alpha^{k-1}\sigma \models_F \varphi_1] \wedge [\sigma \models_F \varphi_1 \,\mathcal{U}\, \varphi_2]\Big)
\end{aligned}
$$

Inductive Step: $\mid \alpha \mid = k+1$
Trivially expanded from the above expansion.

$$
\begin{aligned}
[\alpha\sigma \models_F \varphi_1 \,\mathcal{U}\, \varphi_2] \iff & [\alpha\sigma \models_F \varphi_2] \vee \Big([\alpha\sigma \models_F \varphi_1] \wedge [\alpha^1\sigma \models_F \varphi_2]\Big) \vee \ldots \vee \\
& \Big([\alpha\sigma \models_F \varphi_1] \wedge \ldots \wedge [\alpha^{k-1}\sigma \models_F \varphi_1] \wedge [\alpha^k\sigma \models_F \varphi_1] \wedge [\sigma \models_F \varphi_1 \,\mathcal{U}\, \varphi_2]\Big)
\end{aligned}
$$

Now, in order for $[\alpha\sigma \models_F \varphi_1 \,\mathcal{U}\, \varphi_2]$ to yield `true`, there must be a $k \geq 1$ such that $[\alpha\sigma \models_F \varphi_1] \wedge \ldots \wedge [\alpha^{k-1}\sigma \models_F \varphi_1] \wedge [\alpha^k\sigma \models_F \varphi_2]$, that is

$$
\begin{aligned}
[\alpha\sigma \models_F \varphi_1 \,\mathcal{U}\, \varphi_2] &\iff \exists k \geq 1 \;.\; [\alpha^0\sigma \models_F \varphi_1] \wedge \ldots \wedge [\alpha^{k-1}\sigma \models_F \varphi_1] \wedge \\
&\qquad [\alpha^k\sigma \models_F \varphi_2] \\
&\iff \exists k \geq 1 \;.\; [\alpha\sigma \models_F \Diamond_k \varphi_2] \wedge [\alpha\sigma \models_F \Box_{k-1} \varphi_1]
\end{aligned}
$$

Note that the above recursive definition of Until allows us to evaluate any until formula, and by extension, eventually ($\Diamond \varphi = \top \,\mathcal{U}\, \varphi$) and always formulas. Therefore, we can evaluate any sub-formula using this fixed point representation of until.

$\blacksquare$

4 SMT-based Solution

In this section, we elaborate on our solution for distributed monitoring using the two monitoring techniques mentioned before: (1) automata-based approach, and (2) progression-based approach.

4.1 Overall Idea

Automata-based approach. Recall from Section 1 (Fig. 4) that monitoring a distributed computation may result in multiple verdicts depending upon different ordering of events. In other words, given a distributed computation

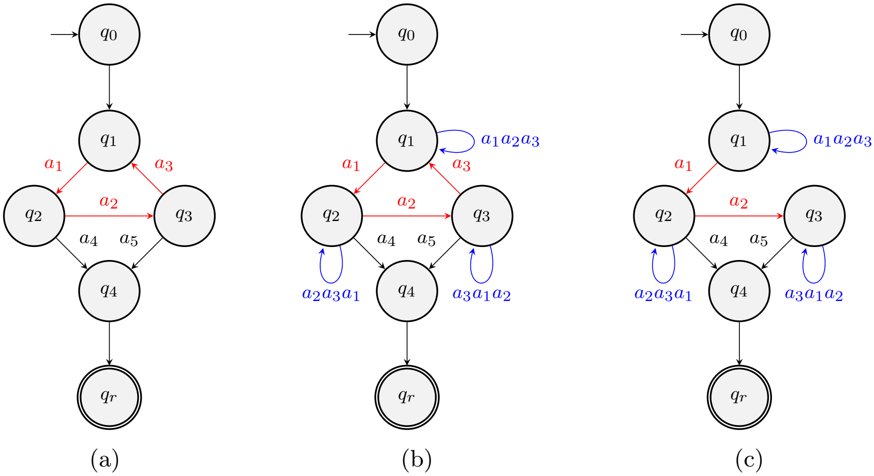


Fig. 8 Removing non-loop cycles in an LTL_3 Monitor.

$(\mathcal{E}, \rightsquigarrow)$ and an LTL formula φ, different ordering of events may reach different states in the monitor automaton $\mathcal{M}_\varphi = (\Sigma, Q, q_0, \delta, \lambda)$ (as defined in Definition 1). In order to ensure that all possible verdicts are explored, we generate an SMT instance for (1) the distributed computation $(\mathcal{E}, \rightsquigarrow)$, and (2) each possible path in the LTL_3 monitor. Thus, the corresponding decision problem is the following: given $(\mathcal{E}, \rightsquigarrow)$ and a monitor path $q_0q_1 \cdots q_m$ in an LTL_3 monitor, can $(\mathcal{E}, \rightsquigarrow)$ reach q_m? If the SMT instance is satisfiable, then $\lambda(q_m)$ is a possible verdict. For example, for the monitor in Fig. 5, we consider two paths $q_0^*q_\perp$ and $q_0^*q_\top$ (and, hence, two SMT instances). Thus, if both instances turn out to be unsatisfiable, then the resulting monitor state is q_0, where $\lambda(q_0) =?$.

We note that LTL_3 monitors may contain non-self-loop cycles. In order to simplify the SMT instance creation process (for each possible path in the LTL_3 monitor), we collapse each non-self-loop cycle into one state with a self-loop labeled by the sequence of events in the cycle using Algorithm 1. As an example, in Fig. 8, Algorithm 1 first takes an LTL_3 monitor (Fig. 8a) and adds the necessary self-loops (Fig. 8b). Then it eliminates all non-self-loop cycles by removing transitions from states with higher identifiers to states with lower identifiers in cycles (Fig. 8c). The non-deterministic nature of the final automata ensure that all the transitions and the accepting language of the automata are preserved.

Lemma 2 Let $\mathcal{M}_\varphi = (\Sigma, Q, q_0, \delta, \lambda)$ be the monitor automaton for LTL formula, φ, and $\mathcal{M}'_\varphi = (\Sigma, Q, q_0, \delta', \lambda)$ be the monitor automaton with no non-self loop cycles, obtained from applying Algorithm 1 on $\mathcal{M}_\varphi$. Given a finite trace, $\alpha = a_1a_2 \cdots a_n$ and a initial state, $q \in Q$, we prove that $\lambda(\delta(q, \alpha)) = \lambda(\delta'(q, \alpha))$.

Algorithm 1: Non-Self Loop Cycle Removal Algorithm

Data: $\mathcal{M}_\varphi = (\Sigma, Q, q_0, \delta, \lambda)$
Result: $\mathcal{M}'_\varphi = (\Sigma, Q, q_0, \delta', \lambda)$

Let CP be the set of all possible paths containing cycles
$\delta' \leftarrow \delta$
foreach $q \in Q$ **do**
  foreach $q \xrightarrow{s_m} \cdots \xrightarrow{s_n} q \in \mathit{CP}$ **do**
    $\delta'(q, s_m \cdots s_n) \leftarrow q$
foreach $q_m \xrightarrow{s} q_n \in \{q_i \xrightarrow{s_k} q_j \mid q \xrightarrow{s_m} \cdots q_i \xrightarrow{s_k} q_j \cdots \xrightarrow{s_n} q \in \mathit{CP}\}$ **do**
  if $m > n$ **then**
    $\delta'(q_m, s) \leftarrow \emptyset$
return $\mathcal{M}_\varphi$

Proof We distinguish the following cases:

Case 1 ($\Rightarrow$):

First we show, $\lambda(\delta(q, \alpha)) \rightarrow \lambda(\delta'(q, \alpha))$, that is, $\forall \alpha, \forall q \in Q \ . \ \lambda(\delta(q, \alpha)) \implies \lambda(\delta'(q, \alpha))$ Let $\alpha = a_1 a_2 \cdots a_n$, where $\forall i \in [1, n].a_i \in \Sigma$. Algorithm 1 removes non-self loop cycles by removing a transition such that the corresponding transition of $\delta(q, a_i)$, $\delta'(q, a_i)$, where $i \in [1, m]$ does not exist. This is such that $\exists k \in [1, i] \ . \ q' \xrightarrow{a_{i-k}} \cdots q \xrightarrow{a_i} q'$. This transition is same as $\delta'(q', a_{i-k} \cdots a_i) = q'$ which was one of the added self-loops. The rest of the transitions are maintained such that $\delta(q, a_i) = \delta'(q, a_i)$, where $q \in Q$ and $i \in [1, m]$.

Case 2 ($\Leftarrow$):

Now, we show, $\lambda(\delta'(q, \alpha)) \rightarrow \lambda(\delta(q, \alpha))$, that is, $\forall \alpha, \forall q \in Q \ . \ \lambda(\delta'(q, \alpha)) \implies \lambda(\delta(q, \alpha))$ Let $\alpha = a_1 a_1 \cdots a_n$, where $\forall i \in [1, n].a_i \in \Sigma$. A self-loop in $\mathcal{M}'_\varphi$ can be represented by $\exists i \in [1, n], \exists k \in [1, n-i] \ . \ \delta'(q, a_i a_{i+1} \cdots a_{i+k}) = q$. In another words, there exists a path $q \xrightarrow{a_i} q' \xrightarrow{a_{i+1}} \cdots \xrightarrow{a_{i+k}} q$ in $\mathcal{M}_\varphi$. The rest of the non-self loop transitions are the same, such that $\delta'(q, a_i) = \delta(q, a_i)$, where $q \in Q$ and $i \in [1, m]$.

Thus, $\lambda(\delta(q, \alpha)) = \lambda(\delta'(q, \alpha))$ ■

Progression-based approach. In a synchronous system, verification on a computation can be performed in a state by state approach due to the existence of a total ordering of events [14]. However, in a *partially* synchronous system, no such ordering of events is possible. A distributed computation $(\mathcal{E}, \rightsquigarrow)$ may have different ordering of events dictated by different interleavings of events. Therefore, it is possible to obtain multiple verdicts on the same distributed computation $(\mathcal{E}, \rightsquigarrow)$. In order to explore these verdicts, we propose a monitoring approach based on *formula progression* that, if possible, *partially* evaluates a formula on the current computation, and based on the verdict, provides a *rewritten* formula that is to be evaluated on the extensions of the computation. As an example, let us consider the formula to be monitored as, $\varphi = \diamondsuit(a \to \diamondsuit b)$. Now, if in some trace in a computation, the monitor observes a, then for the extensions of computations, it is enough to monitor the rewritten formula, $\varphi' = \diamondsuit b$, as the final verdict is no longer dependent on the occurrence of a. We call this method of rewriting formula **Progression**, which we discuss in length later on. In the next two subsections, we present the SMT entities and constraints with respect to *one* monitor path and a distributed computation.

4.2 SMT Entities

SMT entities represent the sub-formulas of an LTL formula and a distributed computation. After the verdicts from all the sub-formulas are generated, we construct our rewritten formula by attaching the said verdicts to their corresponding parent formulas in the parse tree and then performing an in-order traversal starting from the root of the parse tree. At the end of the traversal, the resulting formula is, in fact, the progression for the next computation. We now introduce the entities that represent a path in an LTL_3 monitor $\mathcal{M}_\varphi = (\Sigma, Q, q_0, \delta, \lambda)$ for LTL formula φ and distributed computation $(\mathcal{E}, \rightsquigarrow)$. It should be noted that the SMT entities in this subsection are used in both the automata-based and the progression-based approaches.

Monitor automaton. Let $q_0 \xrightarrow{s_0} q_1 \xrightarrow{s_1} \cdots (q_j \xrightarrow{s_j} q_j)^* \cdots \xrightarrow{s_{m-1}} q_m$ be a path of monitor $\mathcal{M}_\varphi$, which may or may not include a self-loop. We include a non-negative integer variable k_i for each transition $q_i \xrightarrow{s_i} q_{i+1}$, where $i \in [0, m-1]$ and $s_i \in \Sigma$. This is also true for the self-loop $q_j \xrightarrow{s_j} q_j$, for which we include a non-negative interger k_j.

Distributed computation. In our SMT encoding, the set of events, $\mathcal{E}$ are represented by a bit vector, where each bit corresponds to an individual event in the distributed computation, $(\mathcal{E}, \rightsquigarrow)$. We conduct a *pre-processing* of the distributed computation, during which we create an $\mathcal{E} \times \mathcal{E}$ matrix, `hbSet` to incorporate the additional happen-before relations obtained by the clock-synchronization algorithm. Afterwards, we populate the `hbSet` with 0's and 1's, such that $\texttt{hbSet}[i][j] = 1$ if $\mathcal{E}[i] \rightsquigarrow \mathcal{E}[j]$, and $\texttt{hbSet}[i][j] = 0$ otherwise. We introduce a function $\mu : \mathcal{E} \times \mathsf{AP} \to \{\texttt{true}, \texttt{false}\}$ in order to establish a relation between each event and the atomic propositions in it. In the event that other variables or constants are used in defining the predicates (e.g. $x_1 + x_2 \geq 2$), μ is constructed accordingly. Finally, we introduce an uninterpreted function $\rho : \mathbb{Z}_{\geq 0} \to 2^{\mathcal{E}}$ that identifies a sequence of consistent cuts from $\{\}$ to $\{\mathcal{E}\}$ for reaching a verdict, while satisfying a number of given constraints explained in 4.3.

4.3 SMT Constraints

Once we define the necessary SMT entities, we move onto the SMT constraints. We first define the common SMT constraints for consistent cuts that are enforced on both the automata-based and the progression-based approaches. Afterwards we define the SMT constraints that are more dependant on the methodology.

Consistent cut constraints over ρ. In order to ensure that the uninterpreted function ρ identifies a sequence of consistent cuts, we enforce certain consistent cut constraints. The first constraint enforces that each element in the range of ρ is in fact a consistent cut:

$$\forall i \in [0, m]. \forall e, e' \in \mathcal{E}. \Big((e' \rightsquigarrow e) \wedge (e \in \rho(i)) \Big) \rightarrow \Big(e' \in \rho(i) \Big)$$

Next, we enforce that the sequence of consistent cuts identified by ρ start from an empty set of events, and each successor cut of the sequence contains one more new event than its predecessor.

$$\forall i \in [0, m].\ |\rho(i+1)| \ = \ |\rho(i)| + 1$$

Finally, we ensure that each successive consistent cut is immediately reachable in $(\mathcal{E}, \rightsquigarrow)$ by enforcing a subset relation:

$$\forall i \in [0, m].\ \rho(i) \subseteq \rho(i+1)$$

Once a sequence of consistent cuts have been generated, we check if the sequence satisfies the specification. This is done using (1) progression-based approach, where the LTL formula is represented by a SMT constrain and (2) LTL_3 automata-based approach, where a path on the automata is represented as an SMT constraint. This is repeated for all sub-formulas of the original LTL formula and all paths in the LTL_3 automata respectively as discussed below.

Constraints for LTL_3 automata over ρ. These constraints are responsible for generating a valid sequence of consistent cuts given a distributed computation $(\mathcal{E}, \rightsquigarrow)$ that runs on monitor path $q_1 \xrightarrow{s_1} q_2 \cdots q_j^* \cdots \xrightarrow{s_{m-1}} q_m$. We begin with interpreting $\rho(k_m)$ by requiring that running $(\mathcal{E}, \rightsquigarrow)$ ends in monitor state q_m. The corresponding SMT constraint is:

$$\mu(\mathsf{front}(\rho(k_m)), s_{m-1})$$

For every monitor state q_i, where $i \in [0, m-1]$, if q_i does not have a self-loop, the corresponding SMT constraint is:

$$\mu(\mathsf{front}(\rho(k_{i+1} - 1)), s_i) \ \wedge \ (k_i = k_{i+1} - 1)$$

For every monitor state q_j, where $j \in [0, m-1]$, suppose q_j has a self-loop (recall that a cycle of r transitions in the monitor automaton is collapsed into a self-loop labeled by a sequence of r letters). Let us imagine that this self-loop executed z number of times for some $z \geq 0$. Furthermore, we denote the sequence of letters in the self-loop as $s_{j_1} s_{j_2} \cdots s_{j_r}$. The corresponding SMT constraint is:

$$\bigwedge_{i=1}^{z} \bigwedge_{n=1}^{r} \mu\Big(\mathsf{front}\big(\rho(k_j + r(i-1) + n)\big), s_{j_n}\Big)$$

Again, since z is a free variable in the above constraint, the solver will identify some value $z \geq 0$ which is exactly what we need. To ensure that the domain of ρ starts from the empty consistent cut (i.e., $\rho(0) = \emptyset$), we add:

$$k_0 = 0.$$

Finally, let C denote the conjunction of all the above constraints. Recall that this conjunction is with respect to only one monitor path from q_0 to q_m. Since there may be multiple paths in the monitor automaton that can reach q_m from q_0, we replicate the above constraints for each such path. Suppose there are n such paths and let $C_1, C_2, \ldots, C_n$ be the corresponding SMT constraints for these n paths. We include the following constraint:

$$C_1 \vee C_2 \vee C_3 \vee \cdots \vee C_n$$

This means that if the SMT instance is satisfiable, then computation $(\mathcal{E}, \rightsquigarrow)$ can reach monitor state q_m from q_0.

Constraints for LTL progression over ρ. Given a distributed system $(\mathcal{E}, \rightsquigarrow)$, the aforementioned constraints may generate a valid sequence of consistent cuts that may yield different verdicts based on the ordering of the concurrent events. Therefore, in order to avoid false positives, all possible outcomes are explored when evaluating an LTL formula φ on $(\mathcal{E}, \rightsquigarrow)$. We achieve this by checking for both satisfaction and violation in the sequence of consistent cuts $C_0 C_1 C_2 \cdots C_m$ interpreted by the uninterpreted function $\rho(m)$. Note that monitoring any LTL formula using our progression rules will result in monitoring sub-formulas with only atomic propositions, globally and eventually temporal operators:

$$\begin{aligned}
\varphi = p \qquad & \mathsf{front}(\rho_i) \models p, \text{ for } p \in \mathsf{AP} \text{ (satisfaction, i.e.,} \top) \\
\varphi = \Box \phi \qquad & \exists i \in [0, m].\ \mathsf{front}(\rho_i) \not\models \phi \text{ (violation, i.e.,} \bot) \\
\varphi = \Diamond \phi \qquad & \exists i \in [0, m].\ \mathsf{front}(\rho_i) \models \phi \text{ (satisfaction, i.e.,} \top)
\end{aligned}$$

Opposite cases result in a rewritten formula that will progress to the next segment. In general, the verdict for any LTL formula will be derived using our progression rules in Section 3.

5 Optimization

5.1 Segmentation of Distributed Computation

RV is known to be an NP-complete problem in the number of processes in a distributed setting [16]. The complexity exhibits even more exponential blowup during verifying formulas with nested temporal operators. In order to cope with this complexity, we divide our computation into smaller *segments*, $(seg_1, \rightsquigarrow)(seg_2, \rightsquigarrow) \cdots (seg_{l/g}, \rightsquigarrow)$ to create smaller, albeit more SMT problems. Given a distributed computation $(\mathcal{E}, \rightsquigarrow)$ of length l, we divide it into $\frac{l}{g}$ smaller segments length g. The set of events in segment j, where $j \in [1, \frac{l}{g}]$, is the following:

$$seg_j = \left\{ e^n_{\tau,\sigma,\omega} | \sigma \in [\max\{0, (j-1) \times g - \epsilon\}, j \times g] \wedge n \in [1, |\mathcal{P}|] \right\}$$

Note that each segment (barring seg_0) has to be constructed starting at ϵ time units before the previous segments ending point. This creates an overlap of ϵ time units between each pair of adjacent segments. Doing so ensures that no pair of possible concurrent become non-concurrent due to the splits caused by segmentation. Therefore, dividing the actual computation into segments does not have any effect on the final verdict of the said computation. We also use parallelization (similar to the one discussed in [8]) to make our algorithm perform faster, while utilizing most of the computation power modern processors are capable of handling.

Lemma 3 A distributed computation, $(\mathcal{E}, \rightsquigarrow)$, of length l satisfies an LTL formula, φ, if and only if the distributed computation, $(\mathcal{E}, \rightsquigarrow)$, is divided into $\frac{l}{g}$ segments of length g satisfies φ using the automata-based approach. That is,

$$[(\mathcal{E}, \rightsquigarrow) \models_3 \varphi] \iff [(seg_1.seg_2. \cdots .seg_{\frac{l}{g}}, \rightsquigarrow) \models_3 \varphi]$$

Proof Let us assume $[(\mathcal{E}, \rightsquigarrow) \models_3 \varphi] \neq [(seg_1.seg_2. \cdots .seg_{\frac{l}{g}}, \rightsquigarrow) \models_3 \varphi]$, that is, $\{\alpha \models_3 \varphi \mid \alpha \in \mathsf{Tr}(\mathcal{E}, \rightsquigarrow)\} \neq \{\alpha \models_3 \varphi \mid \alpha \in \mathsf{Tr}(seg_1.seg_2. \cdots .seg_{\frac{l}{g}}, \rightsquigarrow)\}$ (Recall Section 2.3).

($\Rightarrow$) Let C_k be a consistent cut such that C_k is in $\mathsf{Tr}(\mathcal{E}, \rightsquigarrow)$, but not in $\mathsf{Tr}(seg_1.seg_2. \cdots .seg_{\frac{l}{g}}, \rightsquigarrow)$ for some $k \in [0, |\mathcal{E}|]$. This implies that the frontier of C_k, $\mathsf{front}(C_k) \not\subseteq seg_1$ and $\mathsf{front}(C_k) \not\subseteq seg_2$ and $\cdots$ and $\mathsf{front}(C_k) \not\subseteq seg_{\frac{l}{g}}$. However, this is not possible, as according to the segmentation construction, there must be a seg_j where $1 \leq j \leq \frac{l}{g}$ such that $\mathsf{front}(C_k) \subseteq seg_j$. Therefore, such C_k cannot exist, and $\{\alpha \models_3 \varphi \mid \alpha \in \mathsf{Tr}(\mathcal{E}, \rightsquigarrow)\} \subseteq \{\alpha \models_3 \varphi \mid \alpha \in \mathsf{Tr}(seg_1.seg_2. \cdots .seg_{\frac{l}{g}}, \rightsquigarrow)\}$. By extension, $[(\mathcal{E}, \rightsquigarrow) \models_3 \varphi] \Rightarrow [(seg_1.seg_2. \cdots .seg_{\frac{l}{g}}, \rightsquigarrow) \models_3 \varphi]$

($\Leftarrow$) Let C_k be a consistent cut such that C_k is in $\mathsf{Tr}(seg_1.seg_2. \cdots .seg_{\frac{l}{g}}, \rightsquigarrow)$, but not in $\mathsf{Tr}(\mathcal{E}, \rightsquigarrow)$ for some $k \in [0, |\mathcal{E}|]$. This implies, $\mathsf{front}(C_k) \subseteq seg_j$

and $\mathsf{front}(C_k) \not\subseteq \mathcal{E}$ for some $j \in [1, \frac{l}{g}]$. However, this is not possible due to the fact that $\forall j \in [1, \frac{l}{g}]$. $seg_j \subseteq \mathcal{E}$. Therefore, such C_k cannot exist, and $\{\alpha \models_3 \varphi \mid \alpha \in \mathsf{Tr}(seg_1.seg_2.\cdots.seg_{\frac{l}{g}}, \rightsquigarrow)\} \subseteq \{\alpha \models_3 \varphi \mid \alpha \in \mathsf{Tr}(\mathcal{E}, \rightsquigarrow)\}$. By extension, $[(seg_1.seg_2.\cdots.seg_{\frac{l}{g}}, \rightsquigarrow) \models_3 \varphi] \Rightarrow [(\mathcal{E}, \rightsquigarrow) \models_3 \varphi]$

Therefore, $[(\mathcal{E}, \rightsquigarrow) \models_3 \varphi] \iff [(seg_1.seg_2.\cdots.seg_{\frac{l}{g}}, \rightsquigarrow) \models_3 \varphi]$.

■

Lemma 4 A distributed computation $(\mathcal{E}, \rightsquigarrow)$ of length l satisfies an LTL formula φ if and only if the distributed computation, $(\mathcal{E}, \rightsquigarrow)$, is divided into $\frac{l}{g}$ segments of length g satisfies φ using the progression-based approach. That is,

$$[(\mathcal{E}, \rightsquigarrow) \models_F \varphi] \iff [(seg_1.seg_2.\cdots.seg_{\frac{l}{g}}, \rightsquigarrow) \models_F \varphi]$$

Proof Using Lemma 1 and Lemma 3, we can trivially prove, $[(\mathcal{E}, \rightsquigarrow) \models_F \varphi] \iff [(seg_1.seg_2.\cdots.seg_{\frac{l}{g}}, \rightsquigarrow) \models_F \varphi]$. ■

5.2 Parallelized Monitoring

Many cloud services use clusters of computers equipped with multiple processors and computing cores. This allows them to deal with high data rates and implement high-performance parallel/distributed applications. Monitoring such applications should also be able to exploit the massive infrastructure. To this end, we now discuss parallelization of our SMT-based monitoring technique.

Let G be a sequence of g segments $G = seg_1 seg_2 \cdots seg_g$. Our idea is to create a job queue for each available computing core, and then distribute the segments evenly across all the queues to be monitored by their respective cores independently. However, simply distributing all the segments across cores is not enough for obtaining a correct result. For example, consider formula $\varphi = a\,\mathcal{U}\,b$ and two segments, seg_1 and seg_2 across two cores, Cr_1 and Cr_2, respectively. In order for the monitor running on Cr_2 to give the correct verdict, it must know the result of the monitor running on Cr_1. In a scenario, where Cr_1 observes one or more $\neg a$ in seg_1, a violation must be reported even if Cr_2 does not observe b and no $\neg a$. Generally speaking, the temporal order of events makes independent evaluation of segments impossible for LTL formulas. Of course, some formulas such as *safety* (e.g., $\Box p$) and *co-safety* (e.g., $\Diamond q$) properties are exceptions.

For our automata-based approach, we address this problem in two steps. Let $\mathcal{M}_\varphi = (\Sigma, Q, q_0, \delta, \lambda)$ be an LTL_3 monitor. Our first step is to create a 3-dimensional reachability matrix RM by solving the following SMT decision problem: given a current monitor state $q_j \in Q$ and segment seg_i, can this segment reach monitor state $q_k \in Q$, for all $i \in [1, g]$, and $j, k \in [0, \mathsf{Q} - 1]$. If the answer to the problem is affirmative, then we mark $RM[i][j][k]$ with `true`, otherwise with `false`. This is illustrated in Fig. 9 for the monitor shown in

	seg_1			seg_2			seg_3			seg_4		
	q_0	$q_\top$	$q_\perp$	q_0	$q_\top$	$q_\perp$	q_0	$q_\top$	$q_\perp$	q_0	$q_\top$	$q_\perp$
q_0	T	F	F	T	T	F	T	T	T	T	T	T
$q_\top$	F	F	F	F	T	F	F	T	F	F	T	F
$q_\perp$	F	F	F	F	F	T	F	F	T	F	F	T

Fig. 9 Reachability Matrix for $a\,\mathcal{U}\,b$

Fig. 5, where the grey cells are filled arbitrarily with the answer to the SMT problem. This step can be made embarrassingly parallel, where each element of RM can be computed independently by a different computing core. One can optimize the construction of RM by omitting redundant SMT executions. For example, if $RM[i][j][\top] = \texttt{true}$, then $RM[i'][\top][\top] = \texttt{true}$ for all $i' \in [i,\ \mathsf{Q}-1]$. Likewise, if $RM[i][j][\perp] = \texttt{true}$, then $RM[i'][\perp][\perp] = \texttt{true}$ for all $i' \in [i, \mathsf{Q}-1]$.

The second step is to generate a verdict reachability tree from RM. The goal of the tree is to check if a monitor state $q_m \in Q$ can be reached from the initial monitor state q_0. This is achieved by setting q_0 as the root and generating all possible paths from q_0 using RM. That is, if $RM[i][k][j] = \texttt{true}$, then we create a tree node with label q_j and add it as a child of the node with the label q_k. Once the tree is generated, if q_m is one of the leaves, only then we can say q_m is reachable from q_0. In general, all leaves of the tree are possible monitoring verdicts. Note that creation of the tree is achieved using a sequential algorithm. For example, Fig.10 shows the verdict reachability tree generated from the matrix in Fig. 9.

For our progression-based approach, we adhere to a similar technique for parallelized monitoring as our automata-based approach. The key difference being, in the progression-based approach subformulas are used, whereas in the automata-based approach different states are used. As an example, the previous formula $\varphi = a\,\mathcal{U}\,b$ will be broken into two subformulas $\varphi_1 = \Box a$ and $\varphi_2 = \Diamond b$, before creating the reachibility matrix, and then generating the verdict for both these subformulas.

Lemma 5 A distributed computation $(\mathcal{E}, \rightsquigarrow)$ of length l satisfies an LTL formula φ if and only if the parallelized monitoring technique satisfies φ. That is,

$$\top \in [(\mathcal{E}, \rightsquigarrow) \models_3 \varphi] \iff \lambda(q) = \top$$

and,

$$\perp \in [(\mathcal{E}, \rightsquigarrow) \models_3 \varphi] \iff \lambda(q) = \perp$$

Where $q \in Q$ is some leaf node in the verdict reachability tree generated from RM during the parallelized monitoring process and λ is the labelling function in $\mathcal{M}_\varphi$.

Base Case: Let us first consider the case where there is only one segment. That is, $l = g$.

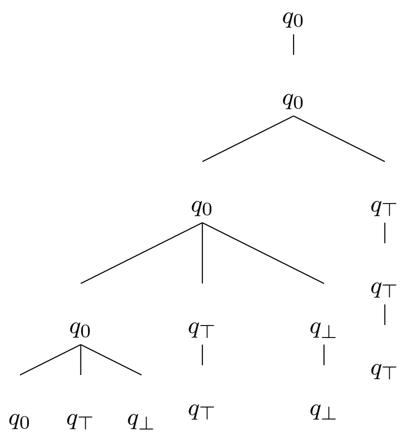


Fig. 10 Reachability Tree for $a\,\mathcal{U}\,b$

($\Rightarrow$) If $\top \in [(\mathcal{E}, \rightsquigarrow) \models_3 \varphi]$ (resp., $\bot \in [(\mathcal{E}, \rightsquigarrow) \models_3 \varphi]$), then according to the construction of the corresponding verdict reachability tree made from the RM, the root node q_0 must have a child $q_\top$ (resp., $q_\bot$), such that, $\lambda(q_\top) = \top$ (resp., $\lambda(q_\bot) = \bot$). This child is also a leaf node, as the height of a verdict reachability tree is 2 when there is only one segment.

($\Leftarrow$) We can trivially show that if $\lambda(q_\top) = \top$ (resp., $\lambda(q_\bot) = \bot$), that is, if $q_\top$ (resp., $q_\bot$) is reachable from q_0, then $\top \in [(\mathcal{E}, \rightsquigarrow) \models_3 \varphi]$ (resp., $\bot \in [(\mathcal{E}, \rightsquigarrow) \models_3 \varphi]$).

Hypothesis: Let us assume the proof as been established for $l = g \times k$. Now we consider $l = q \times (k+1)$ as the segment length.

($\Rightarrow$) If $\top \in [(\mathcal{E}, \rightsquigarrow) \models_3 \varphi]$ (resp., $\bot \in [(\mathcal{E}, \rightsquigarrow) \models_3 \varphi]$), then according to our assumption, there must be at least one node at height $k+1$ (height of the leaf nodes where there are k segments), such that $\lambda(q_\top) = \top$ (resp., $\lambda(q_\bot) = \bot$). Now for $k+1$ number of segments, according to the construction of the corresponding verdict reachability tree made from the RM, the node $q_\top$ (resp., $q_\bot$) can only have the child $q_\top$ (resp., $q_\bot$). Therefore, there must be at least one node at height $k+2$ (height of the leaf nodes when there are $k+1$ segments), such that $\lambda(q_\top) = \top$ (resp., $\lambda(q_\bot) = \bot$).

($\Leftarrow$) We can trivially show that if $\lambda(q_\top) = \top$ (resp., $\lambda(q_\bot) = \bot$), that is, if $q_\top$ (resp., $q_\bot$) is reachable from q_0, then $\top \in [(\mathcal{E}, \rightsquigarrow) \models_3 \varphi]$ (resp., $\bot \in [(\mathcal{E}, \rightsquigarrow) \models_3 \varphi]$).

6 Case Studies and Evaluation

In this section, we emphasize on analyzing our SMT-based solution without digressing into analyzing other dimensions such as instrumentation, data collection, data transfer, monitoring, etc., as given the distributed setting, runtime will be the dominant factor over any other kind of overhead. We evaluate our proposed technique using synthetic experiments, Cassandra (a distributed database), and the RACE dataset from NASA [9].

6.1 Implementation and Experimental Setup

Each experiment can be divided into three phases: (1) data generation, (2) data collection and (3) data verification. For data-generation, we develop a synthetic program that randomly generates a distributed computation (i.e., the behavior of a set of programs in terms of their local computations and inter-process communication). Generating synthetic experimental data offer benefits that enable us to draw comparison between different parameters and their effect on the approach. For example, generating data for different values of ε is beneficial to study its effect on the runtime and the number of false warning verdicts of our approach.

When developing the synthetic distributed system as part of our experiment, we ensure a partially-synchronous setting by including an HLC implementation. We use a uniform distribution $(0, 2)$ to define the type of event (local computation, send and receive message) and a flip-coin distribution for computing the atomic propositions that are true at each local computation event. Although the events in our synthetic experiments in Section 6.2 are uniformly distributed over the length of the trace, the event distribution as part of the Cassandra experiments in Section 6.3 are affected by the network latency and other external factors. In addition, we assume that that there is an external data collection program which keeps track of the data/states of the system under verification. It generates the trace logs which is used by the monitoring program to verify against the given LTL specifications mentioned in Figure 11b.

For data verification, we consider the following parameters: (1) number of processes ($|\mathcal{P}|$), (2) computation duration (l secs), (3) segment length (g), (4) event rate (r events/process/sec), (5) maximum clock skew (ϵ), (6) depth of the automaton (d) and number of nested temporal operators ($|\phi|$) for the LTL formula under monitoring. The main *metric* is to measure the runtime of SMT solving for each configuration of the parameters. Note that the time axis is shown in log-scale in all the plots presented in this section. When we analyze the effects of one parameter by holding the value of all the other parameters at a relevant constant value. In all the graphs, we compare the runtime of our automata-based approach against the progression based approach. We use a MacBook Pro with Intel i7-7567U(3.5Ghz) processor, 16GB RAM, 512 SSD and g++ Apple clang version 12.0.5 (clang-1205.0.22.9) interface to the Z3 SMT-solver [17] to generate the traces. To evaluate our parallel algorithm, we use a server with 2x Intel Xeon Platinum 8180 (2.5GHz) processor, 768GB RAM, 112 vcores and g++(GCC) 9.3.1 interface to the Z3 SMT-solver [17]. Unless specified otherwise, the system under consideration has $|\mathcal{P}| = 2$, $l = 2$ sec, $g = 250ms$, $r = 10$ events/process/sec, $\epsilon = 250ms$ and $d = 3$.

6.2 Analysis of Results – Synthetic Experiments

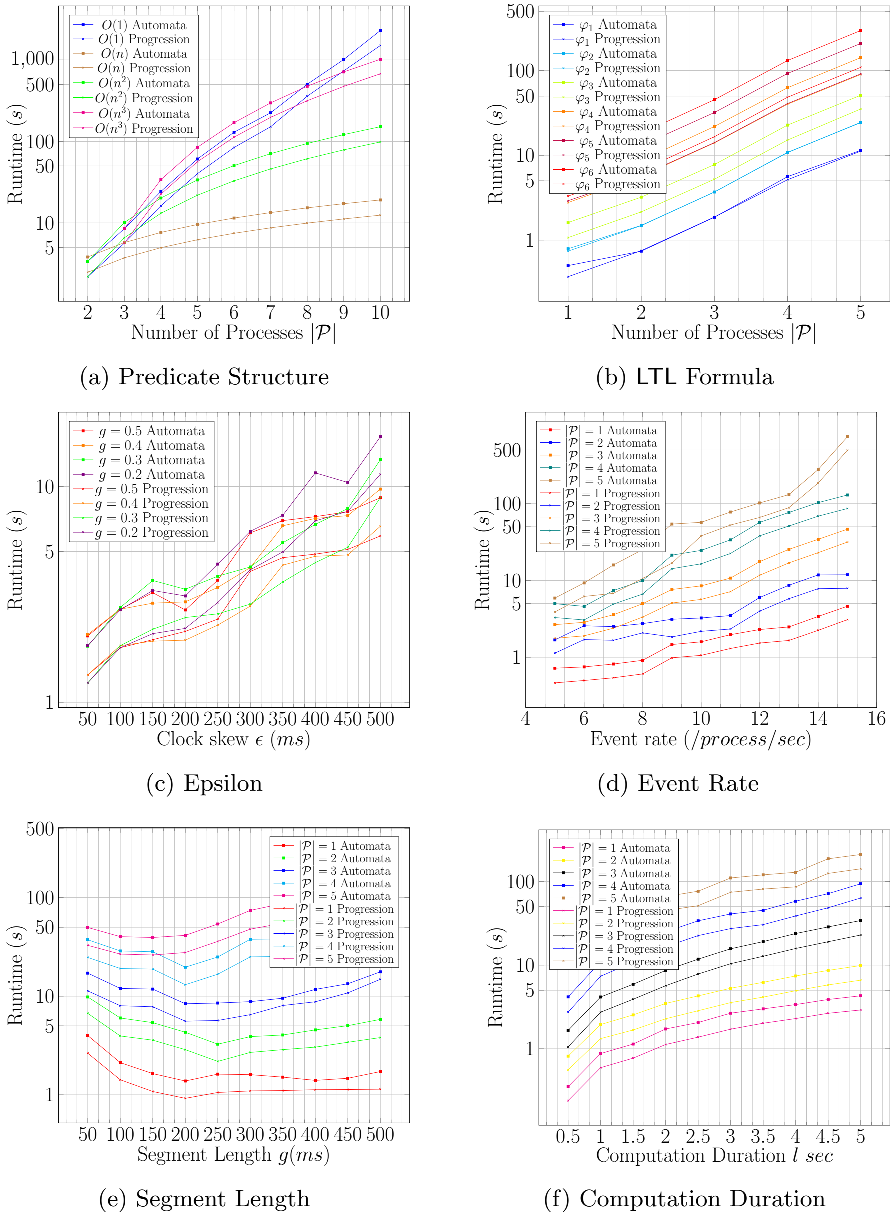
In this set of experiments, we exhaust all the available parameters and note how it affects SMT solving. We test each parameter individually to study its effect on runtime. As our generated synthetic data does not depend on any external factors, we induce a delay to not only limit the number of events happening at every time unit, but also to ensure uniform distribution of events over the execution of each process. We use a uniform distribution $(0, |\Sigma|)$ to assign a value to each local computation event in each process. We only use one CPU core for the following experimental results.

Overall, we notice an improvement of around 35% when the progression based technique is compared to the other automata based approach. This improvement in performance owes to two main reasons: (1) compared to the automata-based approach, the LTL constrains in our progression-based approach is less demanding in terms of computational complexity. Each sub-formula consists of mostly one atomic proposition as opposed to multiple atomic propositions in each path of the automaton, which in turn speeds up the overall verification process, and (2) the total number of SMT-instances needed is fewer due to the less number of sub-formulas compared to automaton paths given the same specification. We now analyze the results in detail.

Impact of predicate structure. In this experiment (Figure 11a), we consider different predicate distribution over AP for the formula, φ_1, i.e., how many processes are involved with a particular predicate. We consider different predicate structures: $O(1)$, $O(n)$, $O(n^2)$ and $O(n^3)$ which signifies the order of the number of SMT-encodings that need to be generated for the given distribution of predicates. As can be seen, the progression based technique outperforms the automata-based technique overall by 35% on average.

Having said that, during our experiments when comparing the runtime of our monitoring approach for increasing number of sub-formulas, we observe a slight decrease in the overall efficiency in runtime when using the progression-based approach compared to the automata-based approach. Since the progression-based approach is based on evaluating each sub-formula, there exists an LTL formula where the number of sub-formulas is more than the number of paths in the corresponding automata, and thus, the the progression-based approach might not be as efficient as the automata-based approach in such a scenario.

For example, consider a formula, $\varphi = \Diamond a \vee \Diamond b \vee \Diamond c$, where the automata has two states, which makes the number of paths to be 2. However, the progression involves 3 sub-formulas, which makes the progression based approach less efficient than its automata counterpart. We would like to point out that the formula can be rewritten as $\Diamond(a \vee b \vee c)$, which makes both the approaches yield similar results. Thus we hypothesize that for all LTL formulas, the progression-based approach will be more (if not equally) efficient to that of the automata-based approach.



(a) Predicate Structure (b) LTL Formula

(c) Epsilon (d) Event Rate

(e) Segment Length (f) Computation Duration

Fig. 11 Synthetic experiments – impact of different parameters.

Impact of LTL formula. Given an LTL formula, the depth of nested temporal operators plays an important role as suggested by Fig. 11b. We experimental with the following LTL formula and the progression based technique achieved an average improvement of 32.8% compared to the automta-based one.

$$\begin{aligned}
&\varphi_1 = \Box p && d=2 \quad |\phi|=1\\
&\varphi_2 = \Box(q \rightarrow \Box p) && d=3 \quad |\phi|=2\\
&\varphi_3 = \Box((q \wedge \Diamond r) \rightarrow (\neg p \,\mathcal{U}\, r)) && d=4 \quad |\phi|=3\\
&\varphi_4 = \Box((q \wedge \Diamond r) \rightarrow (\neg p \,\mathcal{U}\, (r \vee (s \wedge \neg p \wedge \bigcirc(\neg p \,\mathcal{U}\, t))))) && d=5 \quad |\phi|=8\\
&\varphi_5 = \Diamond r \rightarrow (s \wedge \bigcirc(\neg r \,\mathcal{U}\, t) \rightarrow \bigcirc(\neg r \,\mathcal{U}\, (t \wedge \Diamond p))) && d=6 \quad |\phi|=8\\
&\varphi_6 = \Box((q \wedge \Diamond r) \rightarrow (s \wedge \bigcirc(\neg r \,\mathcal{U}\, t) \rightarrow \bigcirc(\neg r \,\mathcal{U}\, (t \wedge \Diamond p)))\,\mathcal{U}\, r) && d=7 \quad |\phi|=9
\end{aligned}$$

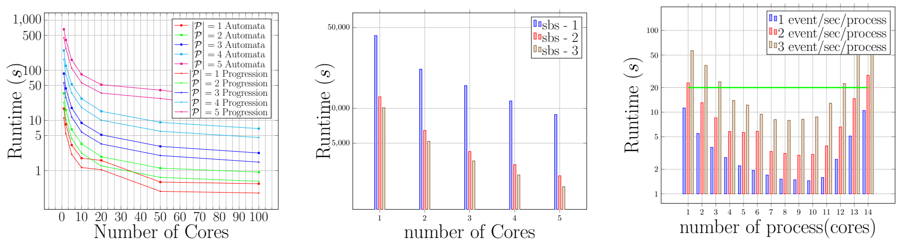
Impact of partial synchrony. Figure 11c shows an expected result where increasing clock skew ϵ results in greater runtime as the number of possible concurrent events across processes increases exponentially. When comparing with the automata-based approach , the progression-based technique yields us an improvement of 33.36%.

Impact of event rate. Figure. 11d shows that our approach breaks even with the computation duration for $|\mathcal{P}| = 3$ for an event rate of $5 events/process/sec$. However, increasing the event rate increases the search space for the SMT solver. Overall we improve by 34.4% by using the progression-based technique compared to the automata-based technique.

Impact of segment count. Increasing the segment length increases the number of events to be worked with, and therefore, exponentially increasing the runtime of our approach. In Fig. 11e, we do not see much improvement for $|\mathcal{P}| = 1, 2$, since the number of events is not large enough to make an impact. However, we see better performance with low segment length for higher number of processes. Note, the runtime increases for very small segment length, since the time taken to generate a higher number of SMT encodings outweigh the performance gain from smaller segments. Here too, we notice an improvement of 32.6% for the progression-based technique over the automata-based technique.

Impact of computation duration. In this experiment (Fig. 11f), we increase computation duration and measure its effect on runtime. With increasing computation duration, the number of segments needed to verify the longer computation increases, and thereby resulting in a linear increase of the runtime. The progression-based approach improves the runtime by 33.1% when compared to the automata-based approach.

Impact of parallelization. Distributing the verification among multiple cores improves the performance of the approach by a considerable amount. As seen in Figure 12a, increasing the number of cores from 1 to 10 improves the performance by a huge margin. However, increasing it further shows little improvement, as the time taken for generating the SMT encodings starts to dominate the time taken to solve it. An improvement of 33.8% is achieved for progression-based approach when compared to automata-based approach.



(a) Synthetic Data (b) SBS Data (c) Google Data

Fig. 12 Impact of parallelization on different data

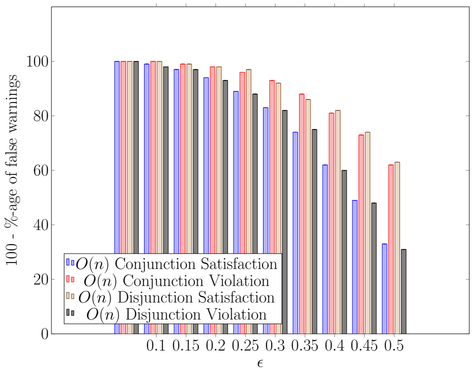


Fig. 13 False Warnings for Synthetic Data

Impact of ϵ on false warnings. As discussed in Section 2.4, the monitor does not have access to the global clock, it can report events as concurrent, when in reality, one happened before the other in the system under observation. However, during this experiment, we keep track of the global clock values separately, which gives us full knowledge over the total ordering of all events. Thus, allowing us to study and report the *real verdicts* alongside the *reported verdicts*. We observe that the monitor sometimes report *false warnings*, that is, it reports both verdicts (satisfaction and violation), when in reality, only one has occurred. Note that the monitor never fails to report real verdicts. However, it may report false warnings alongside real verdicts on some occasions. Although this does not change the correctness of the approach, it may still include false warnings as part of the set of evaluated results.

In Figure 13, we observe that with the increase of the maximum clock skew ϵ, the number of false warnings increases. The increase in false warnings is attributed to the fact that as the value of ϵ increases, so does the number of events considered as concurrent by the monitor.

Additionally, we observe that the number of false warning is greatly influenced by the predicate structure of the LTL formula, as evident from Figure 13. For $O(n)$ conjunctive satisfaction formula monitoring and $O(n)$ disjunctive violation formula monitoring, false warnings might occur if any one of the n

sub-formulas are violated or satisfied, respectively. Therefore, we see a higher number of false warnings. Similarly, for $O(n)$ disjunctive satisfaction formula monitoring and $O(n)$ conjunctive violation formula monitoring, false warnings might occur if all of the n sub-formulas are violated or satisfied, respectively. Therefore, we see a lower number of false warnings.

6.3 Case Study 1: Cassandra

Cassandra [18] is a No-SQL distributed database management system. We simulate a distributed database with two data-centers: one cluster consisting of 4 nodes, and the other cluster consisting of 3 nodes, with one node from each cluster serving as the seed node. All data is replicated among every node in both the clusters. Each node runs on *Red Hat OpenStack Platform* using 4 VCPUs, 4GB RAM, Ubuntu 1804, Cassandra 3.11.6, and Java 1.8.0_252. We have also simulated a system of multiple processes where each process is responsible for the basic database operations (read, write and update). These processes are also capable of inter-process communication that allows for informing other processes in case of a write of a new entry to the database.

To make our simulated database realistic, we tested the latency of our system to the ones offered by Google Cloud, Microsoft Azure and Amazon Web Services. The fastest response was clocked at $41ms$ compared to $100ms$ from our system. The reason behind such a high latency when compared to the industry standard owes to the slow bandwidth and infrastructure differences. We consider a latency of $100ms$ for all our experiments, and fix maximum clock skew ϵ at $250ms$.

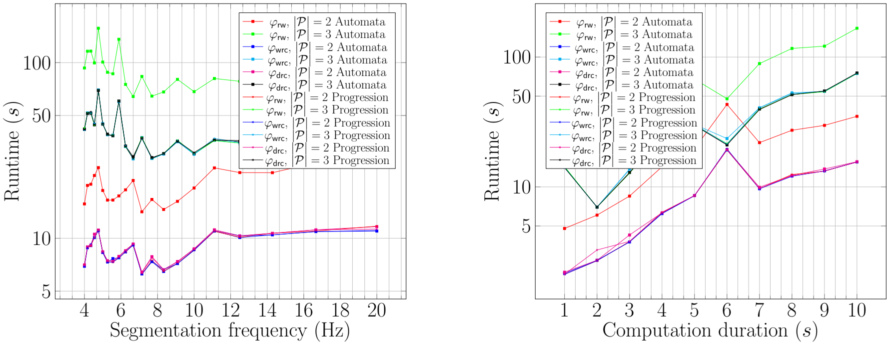
We design the processes such that each process is capable of reading, writing, or updating the entries in the database. We use a $(0, 2)$ uniform distribution to select the type of the operation that is to be performed by the process. Once there is any kind of addition from the write operation, the change is notified to the other processes using the inter-process communications. We consider no loss of messages in transmission and all messages are read by the receiving process immediately once they are received.

In a database, consistency level helps maintain the minimum of replications that needs to be performed on an operation in order to consider the operation to be successfully executed. According to the recommendations from Cassandra the sum of read and write consistency should be more than the replication factor so as to remove any chances of read or write anomaly in the database. We aim to monitor and identify *read/write anomalies* in the database using runtime monitoring techniques. The corresponding LTL specification becomes:

$$\varphi_{rw} = \bigwedge_{i=0}^{n} \Big(write(i) \rightarrow \Diamond read(i)\Big)$$

where n is the number of read/write operations.

One of the challenges for using a distributed database such as Cassandra is the lack of normalization (of database) capabilities. Therefore, we aim to



(a) Segment Length (b) Computation Duration

Fig. 14 Cassandra experiments

monitor *write reference check* and *delete reference check*. We introduce two tables:

$$\textsc{Student}(id, name) \qquad \textsc{Enrollment}(id, course)$$

We enforce the write and delete reference check on the tables above. For a write in the ENROLLMENT table, it should always be preceded by a write in the STUDENT table with the same *id*. Similarly, for a delete from the STUDENT table, it should always be preceded by a delete from the ENROLLMENT table with the same *id*. These enforces no insertion and deletion anomaly, and therefore, leads to the following LTL specification:

$$\varphi_{wrc} = \neg\Big(\neg write(\textsc{Student}.id)\,\mathcal{U}\,write(\textsc{Enrollment}.id)\Big)$$

$$\varphi_{drc} = \neg\Big(\neg delete(\textsc{Enrollment}.id)\,\mathcal{U}\,delete(\textsc{Student}.id)\Big)$$

Extreme load scenario. Figure 14b and 14a plot runtime vs computation duration and runtime vs segmentation frequency respectively, under full read/write load allowed by our network. When compared with the results from that of the synthetic experiments, these results are slightly noisier. This owes to the fact that in the synthetic experiments, the events were evenly spread over the entire computation duration, whereas here they are not uniform. Database operations involving network communications (read, write and update) takes an average of 100ms, however sending and receiving of messages are inter-process communication, and takes about 10-15ms, making the overall event distribution non-uniform. When comparing with the automata-based approach, we do not see much improvement when monitoring φ_{wrc} or φ_{drc} using progression based approach. However, when monitoring φ_{rw}, we observe an average improvement of 55.53%.

Moderate load scenario. In Figure 14b, we were able to make even for number of processes as low as 2. Now, to look for a real-life example with moderate database operations we consider Google Sheets API, which allows a maximum of 500 requests per 100 seconds per project and a 100 requests per second per user, i.e., on an average 5 events/sec per project and a user can only generate 1 event/sec. To evaluate how our approach performs in such a scenario, we increase the number of processes and the number of cores available to monitor such a system to study the time taken to verify the trace generated by such a system. We plot our findings in Fig. 12c, and notice that we break even for an event rate of 3 events/sec/user considering the progression-based approach. This is a significant improvement over the automata-based approach, where we could only break even for an event rate of 2 events/sec/user. Our algorithm performs wells when the number of processes are 7, 8 and 9 which is much more than what is permitted by Google. This allows for us to be confident that our approach can pave way for implementation in a real-life settings.

6.4 Case Study 2: RACE

Runtime for Airspace Concept Evaluation (RACE) [9] is a framework developed by NASA that is used to build an event based, reactive airspace simulation. We use a dataset developed using this RACE framework[2]. This dataset contains three sets of data collected on three different days. Each set was recorded at around 37° N Latitude and 121° W Longitude. The dataset includes all 8 types of messages being sent by the SBS unit by using a Telnet application to listen to port 30003, but we only use the messages with ID $MSG - 3$ which is the Airborne Position Message and includes a flight's latitude, longitude and altitude using which we verify the mutual separation of all pairs of aircraft.

On analyzing the dataset, we observe that the time difference between the time message was generated to the time message was logged is usually less than a second apart, thus we considered an $\epsilon = 1s$ over the time message was generated. Furthermore, calculating the distance between two coordinates is computationally expensive, as we need to factor in parameters such as curvature of earth. In order to speed up distance related calculations, we consider a constant latitude of 111.2km and longitude of 87.62km, at the cost of a negligible error margin. We use these as constants and multiply them by the difference in latitude and longitude, and factor in the altitude to get the distance between two aircrafts. We verify *mutual separation* by considering the minimum separation between every pair of aircrafts to be 500m. From the dataset, we observe that each aircraft generates a message on at least 1 sec intervals. There are 3 separate datasets: sbs-1 consists of 293 aircrafts, 168,283 messages spread over 3 hours and 28 minutes and 58seconds; sbs-2 consists of 110 aircrafts, 64,218 messages spread over 1 hour 1 minute and 46 seconds; sbs-3 consists of 97 aircrafts, 64,162 messages spread over 49 minutes and 42 seconds.

In Fig. 12b, we compare our achieved runtime against the three datasets available from RACE (labelled sbs-1, sbs-2 and sbs-3). We monitor the data in *real time*, with 10s long segments and ϵ of 1s. We test our approach using

[2] https://github.com/NASARace/race-data

the parallelization technique introduced in 5.2 by using more number of cores on the processor and utilize all available cores. Our results break even for 4 cores. This makes our approach desirable for aircraft monitoring and similar systems such as IoT.

7 Related Work

Both centralized and decentralized monitoring approaches have been extensively studied for synchronous and asynchronous systems. In the sequel, we focus on reviewing the existing work on monitoring of distributed system, and subsequently proceed to compare them with ours.

Synchronous Distributed System

Monitoring of LTL formulas in a synchronous distributed system is studied in [2, 19, 20] under the assumption of a global clock shared across all components. Designing large distributed systems with a common global clock shared across all the processes that are usually in different geographical locations, is challenging and expensive.

Other monitoring approaches include [19–22], where the authors employ a decentralized monitor to evaluate the health of a system. In [20], the authors introduce a way of organizing sub-monitors for LTL subformulas in a synchronous distributed system, called choreography. In particular, the monitors are organized as a tree across the distributed system, and each child feeds intermediate results to its parent in a manner similar to diffusing using computation. They formalize choreography-based decentralized monitoring by showing how to synthesize a network from an formula, and give a decentralized monitoring algorithm working on top of an LTL network.

The authors in [21], study a decentralized monitoring approach for stream-based runtime verification technique with respect to a stream-based specification language LOLA [23]. LOLA allows other data-types and aggregated functions like average, median, etc. In [22], the authors deal with an extra challenge where the monitors are only available to read a sub-computation and are vulnerable to crash-faults. The authors show that by replacing few transition in the monitor automata, it is possible to remove non-determinism that were introduced due to monitors only having a partial view of the system. We present a more robust solution in this paper, where $\varepsilon = 0$ would allow for each event occurring in the distributed system to be totally ordered, and therefore replicating the behavior of that of a synchronous distributed system.

Asynchronous Distributed System

Lattice-theoretic centralized and decentralized online predicate detection in asynchronous distributed systems has been extensively studied in [24, 25], and extended to a more generalized temporal operator in [3, 4]. In contrast, our paper is under a more practical assumption, where a clock synchronization algorithm is utilized to limit the time window of asynchrony, as a result, limiting the number of possible concurrent events. In [26], predicate detection is shown to be a powerful tool in solving combinatorial optimization problems. In Figure 12c, we show that our approach is effective in solving predicate detection as well. Predicate detection using SMT has been studied in [27, 28], whereas in our work, we use a more expressive temporal logic (LTL) to express the specification of the system. Predicate detection for asynchronous system is studied in [29], however the assumption to evaluate happened-before relationship between events is too strong.

Knowledge-based monitoring of distributed processes is studied in [30], where the authors propose a method for monitoring safety properties in distributed systems using past-time LTL. The main draw of this approach is being able to produce false negative results. In [31], the authors were able to solve the problem of decentralized monitoring of asynchronous distributed system even when the monitors were vulnerable to crash-faults. The methodology include introducing a family of logics called LTL $_{2k+4}$ that refines the 4-valued LTL by incorporating $2k + 4$ truth values, for each $k \geq 0$, where k is the maximum number of crash-faults that the system could face. The truth values of LTL $_{2k+4}$ can be effectively used by each monitor to reach a consistent global set of verdicts for each given formula, provided k is sufficiently large.

A synthetic benchmarking tool is proposed in [32] for assessing monitoring overhead. While this tool is generally for systems that execute on a single node, the authors show that it is possible to extend its applications to a distributed setting where the system under inspection do not share memory or a global clock, and only communicate through asynchronous message passing. In contrast, we consider a partially synchronous setting in our work ensured by a clock synchronization algorithm. Furthermore, we allow local computation across processes, not just message send and message receive events, which adds to the overall complexity of the system under observation.

In our work, we present both the automata-based approach discussed in [8] and a completely new approach of progression, which monitors the same distributed system, but with increased efficiency. We assume that the processes do not share a global clock and there by can be implemented as part of a large geographically separated system. However, the presence of the clock synchronization algorithm makes way for a bounded maximum clock skew which reduces the size of the state space considerably when compared to that of an asynchronous system. Through our experiments we study the effects of multiple parameters, including clock-synchronization constant and segmentation, on the overall runtime of the approach. To the best of our knowledge there is no runtime monitoring approach that assumes partial-synchrony among processes and yet includes system specification in LTL. This is the reason behind us comparing results internally between the automata and progression based approach. Low values of ε correspond to a synchronous system, where as large values of ε can be considered to follow the behavior of an asynchronous distributed system We show an average of 35% speedup using our progression approach when compared with the automata-based approach over a wide range of experiments. Our solution is also SMT based, and therefore, scalable and efficient.

8 Conclusion and Future Work

In this paper, our focus was on distributed runtime monitoring. Both of our proposed techniques take as input an LTL formula and a distributed computation, and by assuming a bounded clock skew among all processes, they first chop the computation into multiple segments, and then apply either the automata-based monitoring algorithm, or progression-based monitoring algorithm implemented as an SMT decision problem in order to verify the correctness of the said formula. We conducted rigorous synthetic experiments, as well as case studies on monitoring consistency conditions in Cassandra and a NASA air traffic control dataset. Our experiments demonstrate up to 35% improvement in performance in our progression-based algorithm over our automata-based algorithm.

For future work, we plan to study the tradeoff between accuracy and scalability in our approach. Another important extension of our work is distributed RV for timed temporal logics. Such expressiveness will allow us to monitor distributed applications that are sensitive to explicit timing constraints, such as low-level blockchain and cross-chain functions.

9 Acknowledgment

This work is sponsored by the United States National Science Foundation (NSF) awards FMitF 2102106 and SHF 2118356.

References

[1] Corbett, J.C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J.J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., Hsieh, W., Kanthak, S., Kogan, E., Li, H., Lloyd, A., Melnik, S., Mwaura, D., Nagle, D., Quinlan, S., Rao, R., Rolig, L., Saito, Y., Szymaniak, M., Taylor, C., Wang, R., Woodford, D.: Spanner: Google's globally distributed database. ACM Trans. Comput. Syst. **31**(3) (2013). https://doi.org/10.1145/2491245

[2] Bauer, A., Falcone, Y.: Decentralised LTL monitoring. Formal Methods in System Design **48**(1-2), 46–93 (2016)

[3] Ogale, V.A., Garg, V.K.: Detecting temporal logic predicates on distributed computations. In: Proceedings of the 21st International Symposium on Distributed Computing (DISC), pp. 420–434 (2007)

[4] Mostafa, M., Bonakdarpour, B.: Decentralized runtime verification of LTL specifications in distributed systems. In: Proceedings of the 29th IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 494–503 (2015)

[5] Mills, D.: Network time protocol version 4: Protocol and algorithms specification. RFC 5905, RFC Editor (June 2010)

[6] Bauer, A., Leucker, M., Schallhart, C.: Runtime Verification for LTL and TLTL. ACM Transactions on Software Engineering and Methodology (TOSEM) **20**(4), 14–11464 (2011)

[7] Havelund, K., Rosu, G.: Monitoring Programs Using Rewriting. In: Automated Software Engineering (ASE), pp. 135–143 (2001)

[8] Ganguly, R., Momtaz, A., Bonakdarpour, B.: Distributed runtime verification under partial asynchrony. In: Proceedings of the 24nd International Conference on Principles of Distributed Systems (OPODIS), pp. 20–12017 (2020)

[9] Mehlitz, P., Giannakopoulou, D., Shafiei, N.: Analyzing airspace data with race. In: 2019 IEEE/AIAA 38th Digital Avionics Systems Conference (DASC), pp. 1–10 (2019). https://doi.org/10.1109/DASC43569.2019.9081664

[10] Pnueli, A.: The temporal logic of programs. In: Symposium on Foundations of Computer Science (FOCS), pp. 46–57 (1977)

[11] Manna, Z., Pnueli, A.: Temporal Verification of Reactive Systems: Safety. Springer, Berlin, Heidelberg (1995)

[12] Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Communications of the ACM **21**(7), 558–565 (1978)

[13] Kulkarni, S.S., Demirbas, M., Madappa, D., Avva, B., Leone, M.: Logical physical clocks. In: Proceedings of the 18th International Conference on Principles of Distributed Systems, pp. 17–32 (2014)

[14] Bauer, A., Falcone, Y.: Decentralised LTL monitoring. Formal Methods in System Design **48**(1), 46–93 (2016)

[15] Bauer, A., Leucker, M., Schallhart, C.: Comparing ltl semantics for runtime verification. Journal of Logic and Computation **20**(3), 651–674 (2010). https://doi.org/10.1093/logcom/exn075

[16] Garg, V.K.: Elements of Distributed Computing. John Wiley & Sons, Inc., USA (2002)

[17] de Moura, L.M., Bjørner, N.: Z3: An efficient SMT solver. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS), pp. 337–340 (2008)

[18] Lakshman, A., Malik, P.: Cassandra: A decentralized structured storage system. SIGOPS Oper. Syst. Rev. **44**(2), 35–40 (2010). https://doi.org/10.1145/1773912.1773922

[19] El-Hokayem, A., Falcone, Y.: On the monitoring of decentralized specifications: Semantics, properties, analysis, and simulation. ACM Transactions on Software Engineering Methodologies **29**(1), 1–1157 (2020)

[20] Colombo, C., Falcone, Y.: Organising LTL monitors over distributed systems with a global clock. Formal Methods in System Design **49**(1-2), 109–158 (2016)

[21] Danielsson, L.M., Sánchez, C.: Decentralized stream runtime verification. In: Proceedings of the 19th International Conference on Runtime Verification (RV), pp. 185–201 (2019)

[22] Kazemloo, S., Bonakdarpour, B.: Crash-resilient decentralized synchronous runtime verification. In: Proceedings of the 37th Symposium on Reliable Distributed Systems (SRDS), pp. 207–212 (2018)

[23] D'Angelo, B., Sankaranarayanan, S., Sanchez, C., Robinson, W., Finkbeiner, B., Sipma, H.B., Mehrotra, S., Manna, Z.: Lola: runtime monitoring of synchronous systems. In: 12th International Symposium on Temporal Representation and Reasoning (TIME'05), pp. 166–174 (2005). https://doi.org/10.1109/TIME.2005.26

[24] Chauhan, H., Garg, V.K., Natarajan, A., Mittal, N.: A distributed abstraction algorithm for online predicate detection. In: Proceedings of the 32nd IEEE Symposium on Reliable Distributed Systems (SRDS), pp. 101–110 (2013)

[25] Mittal, N., Garg, V.K.: Techniques and applications of computation slicing. Distributed Computing **17**(3), 251–277 (2005)

[26] Garg, V.K.: Predicate detection to solve combinatorial optimization problems. In: Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures, pp. 235–245 (2020)

[27] Valapil, V.T., Yingchareonthawornchai, S., Kulkarni, S.S., Torng, E., Demirbas, M.: Monitoring partially synchronous distributed systems using SMT solvers. In: Proceedings of the 17th International Conference on Runtime Verification (RV), pp. 277–293 (2017)

[28] Momtaz, A., Basnet, N., Abbas, H., Bonakdarpour, B.: Predicate monitoring in distributed cyber-physical systems. In: International Conference on Runtime Verification, pp. 3–22 (2021). Springer

[29] Stoller, S.D.: Detecting global predicates in distributed systems with clocks. In: Mavronicolas, M., Tsigas, P. (eds.) Distributed Algorithms, pp. 185–199. Springer, Berlin, Heidelberg (1997)

[30] Sen, K., Vardhan, A., Agha, G., G.Rosu: Efficient decentralized monitoring of safety in distributed systems. In: Proceedings of the 26th International Conference on Software Engineering (ICSE), pp. 418–427 (2004)

[31] Bonakdarpour, B., Fraigniaud, P., Rajsbaum, S., Rosenblueth, D.A., Travers, C.: Decentralized asynchronous crash-resilient runtime verification. In: Proceedings of the 27th International Conference on Concurrency Theory (CONCUR), pp. 16–11615 (2016)

[32] Aceto, L., Attard, D.P., Francalanza, A., Ingólfsdóttir, A.: On benchmarking for concurrent runtime verification. In: Guerra, E., Stoelinga, M. (eds.) Fundamental Approaches to Software Engineering, pp. 3–23. Springer, Cham (2021)