Decentralized Runtime Verification of LTL Specifications in Distributed Systems

Menna Mostafa
School of Computer Science
University of Waterloo, Canada
Email: menna.mostafa@uwaterloo.ca

Borzoo Bonakdarpour
Department of Computing and Software
McMaster University, Canada
Email: borzoo@mcmaster.ca

***Abstract*—Runtime verification is a lightweight automated formal method for specification-based runtime monitoring as well as testing of large real-world systems. While numerous techniques exist for runtime verification of sequential programs, there has been very little work on specification-based monitoring of distributed systems. In this paper, we propose the first *sound* and *complete* method for runtime verification of asynchronous distributed programs for the 3-valued semantics of LTL specifications defined over the global state of the program. Our technique for evaluating LTL properties is inspired by distributed computation *slicing*, an approach for abstracting distributed computations with respect to a given predicate. Our monitoring technique is fully decentralized in that each process in the distributed program under inspection maintains a replica of the monitor automaton. Each monitor may maintain a set of possible verification verdicts based upon existence of concurrent events. Our experiments on runtime monitoring of a simulated swarm of flying drones show that due to the design of our Algorithm, monitoring overhead grows only in the *linear order* of the number of processes and events that need to be monitored.**

***Keywords*-Runtime monitoring; Asynchronous distributed systems; Formal methods.**

I. Introduction

Correctness refers to the assertion that a computing system satisfies its specification expressed in some logic. Achieving correctness in distributed systems is particularly challenging due to their inherent complexity caused by non-determinism in execution of distributed processes, occurrence of different types of faults, and uncertainty in cyber and physical implementation of communication primitives. Thus, there is a pressing need for designing techniques that ensure correctness of distributed applications.

Runtime verification is a lightweight technique where a *monitor* checks at run time whether or not the execution of a system under inspection satisfies a given correctness property. Runtime verification complements exhaustive verification methods such as model checking and theorem proving, as well as incomplete solutions such as testing and debugging. Designing a decentralized runtime monitor for a distributed system is an especially difficult task since it involves dealing with computing global snapshots and estimating the total order of events in order for the monitor to reason about the temporal behavior of the system.

This is an extended version of the paper appeared in IPDPS'15.

A. Related Work

The following lines of work are closely related to runtime verification of distributed applications:

- Lattice-theoretic centralized and decentralized online predicate detection in distributed systems has been studied in [5], [10]. However, this line of work does not address monitoring properties with temporal requirements. This shortcoming is addressed in [11] for a fragment of temporal operators, but for offline monitoring.
- In [13], the authors design a method for monitoring safety properties in distributed systems using the past-time linear temporal logic (PLTL). This work, however, is not sound, meaning that valuation of some predicates and properties may be overlooked. This is because monitors gain knowledge about the state of the system by piggybacking on the existing communication among processes. That is, if processes rarely communicate, then monitors exchange very little information and, hence, some violations of properties may remain undetected.
- Runtime verification of LTL for synchronous distributed systems, where processes share a single global clock, has been studied in [2], [6].

Also, in [3], the authors introduce parallel algorithms for runtime verification of sequential programs. Finally, in [7], the authors introduce a lower-bound on the number of values that a logic must have in order to monitor safety properties in distributed algorithms in a decentralized fashion in the presence of crash faults.

B. Contributions

The main contribution of this paper is a novel decentralized algorithm for runtime verification of distributed programs. In our setting, a distributed program consists of a set of asynchronous processes that communicate using message-passing primitives over reliable channels. Our algorithm conducts runtime verification for the 3-valued semantics of the linear temporal logic (LTL_3) [1], designed for reasoning about LTL properties for finite executions. It indeed addresses the shortcomings of the related work: it (1) does not assume a global clock; (2) is able to verify temporal properties and not just safety predicates at run time, and (3) is sound and complete.

Intuitively, our technique works as follows. Each process in the program is composed with a *monitor process*. Each

monitor process is augmented with a *monitor automaton* for each LTL$_3$ property under inspection. The monitor automaton is a deterministic finite state Moore machine that defines how the monitor process should evaluate a property. The states of the automaton are labeled by evaluation verdicts while the transitions are labeled by global-state predicates (see Fig. 4 for an example). Thus, each monitor process should be able to evaluate these predicates. To this end, we adapt the lattice-theoretic technique proposed in [5] for detecting global-state predicates at run time. However, due to the existence of concurrent events (e.g., see lines 5 and 6 of the processes in Fig. 1), a monitor process may construct different finite executions of consistent global states. Consequently, the monitor process maintains a *set* of possible evaluation verdicts. This set evolves over time, meaning that monitoring verdicts may be added or removed depending upon the truthfulness of predicates and the structure of the monitor automaton. Adding monitoring verdicts only happens due to the existence of concurrent events. We argue that maintaining a set of possible verification verdicts due to unresolvable nondeterminism is indeed what a decentralized monitor should propose for verification of distributed programs, since it may uncover intricate existing bugs. We emphasize that the size of this set is generally very small (usually less than five even for very sophisticated properties), even if there is high degree of concurrency. Finally, merging takes place, if over time, a monitoring verdict is not possible any more. Our algorithm is *sound*, meaning that if the total order of events can be somehow constructed, then our algorithm identifies the verification verdict for the total order as one of the possible verdicts. It is also *complete*, meaning that among the set of verdicts computed by local monitors, at least one corresponds to the verdict for the total order of events.

Our algorithm is fully implemented and we report the results of sophisticated experiments on a case study on runtime monitoring of simulated swarm of unmanned aerial vehicles (UAVs). Our experiments to measure the communication and memory overhead of monitoring as well as the delay in evaluating different properties on the swarm clearly shows that although our algorithm explores all possible interleavings, it does not result in an explosion in communication or local memory usage. In particular, due to the design of our algorithm, monitoring overhead grows only in the *linear order* of the number of processes and events that need to be monitored. These results clearly show that our algorithm elegantly works as a *lightweight* automated formal method for online reasoning about the correctness of a distributed application.

Organization: The rest of the paper is organized as follows. Section II presents our computation model for distributed programs. Our specification language LTL$_3$ is described in Section III. We introduce the formal statement of the problem in Section IV. Our runtime verification algorithm is presented in Section V, while the result of experiments are discussed in Section VI. Finally, we make concluding remarks and discuss future work in Section VII. All proofs appear in the appendix.

II. Distributed Programs

Let V be a set of *discrete variables*, where the domain of each variable $v \in V$ is denoted by D_v. A *state* is a mapping

```
{x1=0}
Process P1()
{
  send(P2,m1);
  x1=5;
  x1=10;
  recv(m2);
}
```

```
1 {x2=0}
2 Process P2()
3 {
4   recv(m1);
5   x2=15;
6   x2=20;
7   send(P1,m2);
8 }
```

Fig. 1: A distributed program.

from each variable $v \in V$ to a value in its domain D_v. A *process* P over the set V is defined as a tuple $P = \langle s^0, S, T\rangle$, where S is the local *state space* (i.e., the set of all states), $s^0 \in S$ is the initial state, and $T \subseteq S \times S$ is the set of local *transitions*. An *atomic proposition* is a subset of S. We denote the set of all atomic propositions for a process by $AP = 2^S$. If an atomic proposition $p \in AP$ includes a state $s \in S$, we say that p *holds* in s and write $s \models p$. We define a *local-state trace* of a process $P = \langle s^0, S, T\rangle$ as a finite or infinite sequence of local states $\sigma = s_0 s_1 s_2 \cdots$, such that (1) $s_0 = s^0$, and (2) for all $i \geq 0$, we have $(s_i, s_{i+1}) \in T$.

A *distributed program* $\mathcal{D} = \{P_1, P_2, \ldots, P_n\}$ is a set of n reliable processes. We assume that no two processes share a common variable. Processes communicate with each other over lossless FIFO channels using asynchronous messages whose time of arrival is unbounded. An *event* e in a process $P_i = \langle s_i^0, S_i, T_i\rangle$ in $\mathcal{D}$, where $1 \leq i \leq n$, is either:

- A local transition $(s_0, s_1) \in T$, called an internal event.
- A message send, where the local state of P remains unchanged.
- A message receive, where the local state of P does not change.

Fig. 1 shows our running example of a distributed program that consists of two communicating processes `P1` and `P2`, with local integer variables `x1` and `x2` (both initialized to 0), respectively. Each instruction on Lines 4-7 is an event denoted by $e_0^1 \cdots e_3^1$ (respectively, $e_0^2 \cdots e_3^2$) of process `P1` (respectively, `P2`).

Since send and receive events do not change the local state of a process $P = \langle s^0, S, T\rangle$, we represent their occurrence by a self-loop $(s, s) \in T$, where s is the state of occurrence. Thus, any event e can be associated with one and only state $\mathfrak{s}(e)$ reached by execution of e. An *event-trace* of a process $P_i \in \mathcal{D}$ is a finite or infinite sequence of events $\eta = e_0^i e_1^i \cdots$ iff there exists a state trace $\sigma = s^0 \mathfrak{s}(e_0^i)\mathfrak{s}(e_1^i)\cdots$. We denote the set of all possible events in P_i by E_i and the set of all events of $\mathcal{D}$ by $E_{\mathcal{D}} = \cup_{i=1}^n E_i$. Throughout this paper, we denote the set of all global states of program $\mathcal{D} = \{P_1, P_2, \ldots, P_n\}$ by $\Sigma = S_1 \times, \ldots, \times S_n$; i.e., the Cartesian product of local state space of all processes. A (global-state) *predicate* P is a subset of Σ defined using the following syntax:

$$p ::= p_i \wedge p_j \mid p_i \vee p_j \mid \neg p$$

where $1 \leq i, j \leq n$, $p_i \in AP_i$, and $p_j \in AP_j$. We denote the

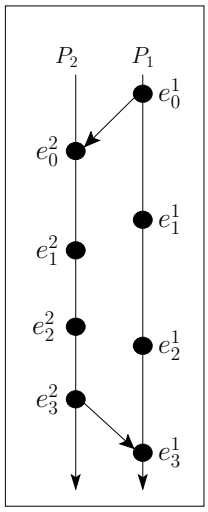


Fig. 2: Visual order of events.

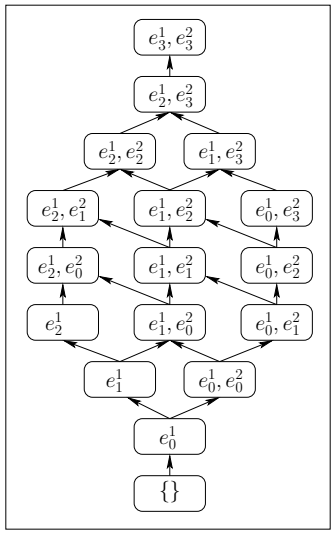


Fig. 3: Computation lattice.

set of all predicates by $Pred$.

Since processes in a distributed program do not share a global clock and do not necessarily execute at the same speed, in order to reason about the global state of the distributed program, we employ Lamport's logical clocks [8].

Definition 1 (Happened Before Relation): A relation $\rightsquigarrow \subseteq E_{\mathcal{D}} \times E_{\mathcal{D}}$ is a *happened before* relation iff for any events $a, b, c \in E_{\mathcal{D}}$, the following conditions hold:

- If $a = (s_0, s_1)$ and $b = (s_1, s_2)$ are two internal events of a process, where s_0, s_1, and s_2 are three distinct local states, then $a \rightsquigarrow b$.
- If a is a send event by a process P and b is the corresponding receive event by some process P', then $a \rightsquigarrow b$.
- If $a \rightsquigarrow b$ and $b \rightsquigarrow c$, then $a \rightsquigarrow c$. ■

Definition 2 (Concurrent Events): Two distinct events $a, b \in E_{\mathcal{D}}$ are called *concurrent events* (denoted $a \parallel b$) iff $(a \not\rightsquigarrow b \wedge b \not\rightsquigarrow a)$. ■

For example, in the program in Fig. 1, we have $e_0^1 \rightsquigarrow e_2^2$ and $e_2^1 \parallel e_1^2$. A visual order of all the events in `P1` and `P2` is shown in Fig. 2. Notice that one can trivially define the happened before relation for states as well.

Monitoring the execution of a distributed program requires reasoning about the *global state* of the program.

Definition 3 (Global State): A *global state* of a distributed program $\mathcal{D} = \{P_1, P_2, \ldots, P_n\}$ is a tuple $g = \langle s_1, s_2, \ldots, s_n \rangle$, where each s_i, $1 \leq i \leq n$, is a local state of process P_i. ■

A global transition is defined as the occurrence of a single event in exactly one process. However, since the program may have concurrent events, m concurrent events will enable m global transitions leading to m next possible global states. Since all global states (as defined in Definition 3) are not necessarily reachable in a distributed program, we define the concept of *consistent cuts*.

Definition 4 (Consistent Cut [4]): Let $\mathcal{D} = \{P_1, P_2, \ldots, P_n\}$ be a distributed program. A *cut* C is a subset of $E_{\mathcal{D}}$ and is represented by a tuple of *frontier events* $FE_C = \langle e^1_{last}, e^2_{last}, \ldots, e^n_{last} \rangle$, such that e^i_{last}, $1 \leq i \leq n$, is the last event occurred in process P_i in C. We say that C is a *consistent cut* iff $\forall e \in C, e' \in E_{\mathcal{D}} : ((e' \rightsquigarrow e) \Rightarrow (e' \in C))$. ■

For example, in Fig. 2, the cut with frontier $\langle e_1^1, e_0^2 \rangle$ is consistent, while the cut with frontier $\langle e_3^1, e_2^2 \rangle$ is not consistent. We denote the set of all consistent cuts in a computation by $\mathcal{CC}$.

Definition 5 (Consistent Global State): We say that $g = \langle s_0, s_1, \ldots, s_n \rangle$ is a *consistent global state* of a distributed program $\mathcal{D} = \{P_1, P_2, \ldots, P_n\}$ iff there exists a consistent cut C with frontier events $FE_C = \langle e_1, e_2, \ldots, e_n \rangle$, such that for all i, $1 \leq i \leq n$, we have $\mathfrak{s}(e_i) = s_i$. We denote the fact that g is the global state corresponding to frontier FE_C by $g = \mathfrak{s}(FE_C)$. ■

For example, the global state, where `x1 = 5` and `x2 = 20` is a consistent global state.

Definition 6 (Computation Lattice [9]): A *computation lattice* $\mathcal{L} = (\mathcal{CC}, \rightsquigarrow)$ is a directed graph where (1) the set of vertices is the set of all consistent cuts $\mathcal{CC}$, (2) the edge relation is the happened before relation, and (3) every pair of consistent cuts have a unique greatest common predecessor and a unique lowest common successor. ■

Fig. 3 shows the computation lattice for the distributed program in Fig. 1. Note that each node in the lattice is the frontier of a consistent cut.

We denote the set of all paths of lattice $\mathcal{L}$ by $\Pi_{\mathcal{L}}$. A computation lattice encodes the set of finite or infinite paths of consistent cut frontiers of the form $FE_{C_0} FE_{C_1} \cdots$, where each lattice step corresponds to exactly the occurrence of one event in exactly one process. This event can be internal, send, or receive event.

Definition 7 (Global-state Trace): Given a computation lattice $\mathcal{L}$ and a (finite or infinite) path $\pi = FE_{C_0} FE_{C_1} \cdots$ in $\Pi_{\mathcal{L}}$, the global-state trace corresponding to π is the sequence $\gamma = \mathfrak{s}(\pi) = g_0 g_1 \cdots$, such that $\forall i \geq 0 : g_i = \mathfrak{s}(FE_{C_i})$. ■

III. Runtime Verification for LTL

A. Linear Temporal Logic (LTL) [12]

Linear temporal logic (LTL) is a popular formalism for specifying properties of (concurrent) programs. Recall that we denoted the set of all global states by Σ. We denote the set of all finite traces over Σ by Σ^* and the set of all infinite traces by Σ^ω. For a finite trace α and a trace γ, we write $\alpha\gamma$ to denote their *concatenation*.

*Definition 8 (*LTL *Syntax):* The set of LTL formulas is inductively defined as follows:

$$\varphi ::= \mathit{true} \mid p \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \bigcirc\varphi \mid \varphi_1 \mathbf{U} \varphi_2$$

where $p \in \mathit{Pred}$, and, $\bigcirc$ (next) and $\mathbf{U}$ (until) are temporal operators. ■

*Definition 9 (*LTL *Semantics):* Let $\gamma = g_0 g_1 \ldots$ be an infinite global-state trace in Σ^ω, i be a non-negative integer, and $\models$ denote the *satisfaction* relation. Semantics of LTL is defined inductively as follows:

$$\begin{array}{lll}
\gamma, i \models \mathit{true} & & \\
\gamma, i \models p & \text{iff} & g_i \models p \text{ (i.e., } g_i \in p) \\
\gamma, i \models \neg\varphi & \text{iff} & \gamma, i \not\models \varphi \\
\gamma, i \models \varphi_1 \wedge \varphi_2 & \text{iff} & \gamma, i \models \varphi_1 \ \wedge \ \gamma, i \models \varphi_2 \\
\gamma, i \models \bigcirc\varphi & \text{iff} & \gamma, i+1 \models \varphi \\
\gamma, i \models \varphi_1 \, \mathbf{U} \, \varphi_2 & \text{iff} & \exists k \geq i : \gamma, k \models \varphi_2 \wedge \\
 & & \forall j : i \leq j < k : \gamma, j \models \varphi_1.
\end{array}$$

In addition, $\gamma \models \varphi$ holds iff $\gamma, 0 \models \varphi$ holds. ■

An LTL formula φ defines a set of traces, denoted by $L(\varphi)$, (i.e., a *language* or a *property*). For simplicity, we introduce abbreviation temporal operators: $\Diamond\varphi$ (*eventually* φ) denotes $\mathit{true}\,\mathbf{U}\,\varphi$, and $\Box\varphi$ (*always* φ) denotes $\neg\Diamond\neg\varphi$. For instance, non-starvation can be expressed by formula $\Box(r \Rightarrow \Diamond g)$, meaning that 'if a process requests entering a critical section, then it is eventually granted'.

Definition 10 (Satisfies): Let $\mathcal{D}$ be a distributed program and φ be an LTL property. We say that a global-state trace γ of $\mathcal{D}$ *satisfies* φ iff $\gamma \in L(\varphi)$. Otherwise, we say that γ *violates* φ. If all global-state traces of $\mathcal{D}$ are in $L(\varphi)$, then we say that $\mathcal{D}$ satisfies φ (denoted $\mathcal{D} \models \varphi$). ■

B. 3-valued LTL

LTL semantics is defined over infinite traces and a running program can only deliver a finite trace at a monitoring point. To formalize satisfaction of LTL properties at run time, in [1], the authors propose semantics for LTL, where the evaluation of a formula ranges over three values $\{\top, \bot, ?\}$ (denoted LTL_3). The value '?' expresses the fact that it is not possible to decide on the satisfaction or violation of a property, given the current program finite trace.

Definition 11 (LTL_3 *semantics):* Let $\alpha \in \Sigma^*$ be a finite trace. The valuation of an LTL_3 formula φ with respect to α, denoted by $[\alpha \models \varphi]$, is defined as follows:

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

■

Note that the syntax '$[\alpha \models \varphi]$' for LTL_3 semantics is defined over finite words as opposed to '$\gamma \models \varphi$' for LTL semantics, which is defined over infinite words. For example, given a finite program trace $\alpha = g_0 g_1 \cdots g_n$, property $\Diamond p$ holds iff

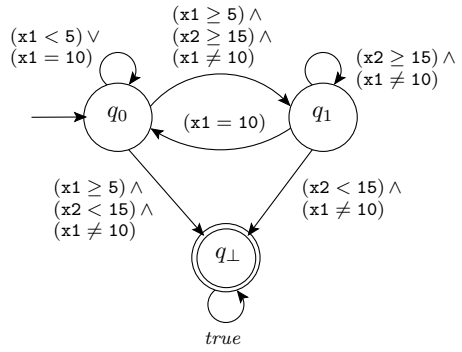


Fig. 4: The monitor automaton for property $\psi = \Box((\texttt{x1} \geq 5) \Rightarrow ((\texttt{x2} \geq 15)\ \mathbf{U}\ (\texttt{x1} = 10)))$.

$g_i \models p$, for some i, $0 \leq i \leq n$. Otherwise, the property evaluates to ?.

Definition 12 (LTL Monitor Automaton): Let φ be an LTL formula over predicates Pred. The *monitor automaton* $\mathcal{A}^\varphi$ of φ is the unique deterministic finite-state automaton (DFA) $\mathcal{A}^\varphi = (\mathit{Pred}, Q, q^0, \delta, \lambda)$, where Q is a set of states, q^0 is the initial state, $\delta \subseteq Q \times \mathit{Pred} \times Q$ is the transition relation, and λ is a function that maps each state in Q to a value in $\{\top, \bot, ?\}$, such that for any finite trace $\alpha \in \Sigma^*$:

$$[\alpha \models \varphi] = \lambda(\delta(q^0, \alpha))$$

■

For example, Fig. 4 shows the monitor automaton for property

$$\psi = \Box((\texttt{x1} \geq 5) \Rightarrow ((\texttt{x2} \geq 15)\ \mathbf{U}\ (\texttt{x1} = 10))),$$

where $\lambda(q_0) = \lambda(q_1) = ?$ and $\lambda(q_\bot) = \bot$. The proposition true denotes the set AP of all propositions. Notice that state $q_\bot$ is a final state with no outgoing transition to other states. This is because once verdicts $\bot$ or $\top$ are reached, according to Definition 11, they cannot change.

IV. Formal Problem Description

Roughly speaking, given a distributed program $\mathcal{D}$ and an LTL_3 property φ, our goal is to construct a set $\mathcal{M}$ of *monitor processes*, such that their composition with $\mathcal{D}$ can evaluate φ at run time in a sound, complete, and decentralized fashion. However, since the occurrence of events in a distributed program does not form a total order, valuation of φ at each time instance cannot be a single value. I.e., since there are multiple possible execution paths in the computation lattice, multiple monitoring verdicts may be reached.

In order to formalize the problem, let us assume that at run time, an *oracle* can construct the total order of events occurred in physical time. This total order, which corresponds to one path in the computation lattice, is a finite global-state trace α. Thus, the oracle can accurately compute $[\alpha \models \varphi]$. However, distributed processes may compute different monitoring verdicts depending upon their view of the computation lattice that corresponds to α. For example, consider property

$$\psi = \Box((\texttt{x1} \geq 5) \Rightarrow ((\texttt{x2} \geq 15)\ \mathbf{U}\ (\texttt{x1} = 10)))$$

In the computation lattice in Fig. 3, any path that goes through node $\langle e_1^1 \rangle$ will evaluate to $\perp$, while for path $\beta = \langle\{\}\rangle\langle e_0^1\rangle\langle e_0^1, e_0^2\rangle\langle e_0^1, e_1^2\rangle\langle e_0^1, e_2^2\rangle\langle e_1^1, e_2^2\rangle\langle e_1^1, e_3^2\rangle\langle e_2^1, e_3^2\rangle\langle e_3^1, e_3^2\rangle$, we have $[\beta \models \psi] =?$. On the contrary, if we consider property

$$\psi' = \Box((\texttt{x1} \geq 5) \Rightarrow ((\texttt{x2} = 15)\ \mathbf{U}\ (\texttt{x1} = 10)))$$

then all paths will evaluate to $\perp$. Thus, depending upon the LTL$_3$ property and the reconstructed path in the lattice, the monitor may compute a different valuation. This is simply due to the nature of concurrent events and the fact that determining the total order of events is not possible. We argue that maintaining a *set* of possible verification verdicts due to unresolvable non-determinism is indeed what a decentralized monitor should propose for verification of distributed programs, since it may uncover intricate existing bugs.

Following the above discussion, we argue that the goal of runtime verification for LTL$_3$ in a distributed system must be designing a decentralized runtime monitor, such that the verdict identified by the oracle is computed by at least one of the monitor processes.

Decentralized LTL3 monitoring problem: Given a distributed program $\mathcal{D} = \{P_1, P_2, \ldots, P_n\}$, a finite global-state trace $\alpha \in \Sigma^*$, and an LTL$_3$ property φ, the goal is to identify a set of *monitor processes* $\mathcal{M} = \{M_1, M_2, \ldots, M_n\}$, such that

if each monitor process M_i

- can read the local state of P_i in one atomic step;
- can communicate with other monitor processes, and
- maintains a monitor automaton $\mathcal{A}_i^{\varphi} = (Pred, Q_i, q_i^0, \delta_i, \lambda_i)$

then

- there exists a monitor process M_i, $1 \leq i \leq n$, that can reconstruct a trace $\alpha' \in \Sigma^*$, such that $\lambda_i(\delta_i(q_i^0, \alpha')) = [\alpha \models \varphi]$.

We note that for brevity, the notions of "communicating" and "reading the state" are not formalized, but their meaning complies with the common understanding of these concepts. Also, we emphasize that if one can design an algorithm that solves the above problem, then it is straightforward to extend the algorithm, so that *all* monitors maintain the same set of verdicts, including the one that would be determined by the oracle.

We also note that the above problem statement is for soundness. With regard to completeness, the problem statement would trivially be the opposite; i.e., among the set of verdicts computed by local monitors at each time instance, at least one corresponds to the verdict for the total order of events at physical time. We skip the formal statement of completeness for reasons of space.

V. Monitoring Algorithm Design

A. Main Idea – Algorithm Sketch

First, we present the sketch of our algorithm that solves the problem of decentralized LTL$_3$ monitoring as stated in Section IV. For a distributed program $\mathcal{D} = \{P_1, \ldots, P_n\}$, we compose each process in P_i with a monitor process M_i that can read the local state of P_i in one atomic step. Recall that given a property φ, each monitor M_i attempts to evaluate φ by constructing the global-state trace through communicating with other monitor processes in $\mathcal{M} = \{M_1, \ldots, M_n\}$. This requires dealing with three problems, for each, we propose a solution:

1) *(Predicate detection)* Since each transition in a monitor automaton is labeled by a global-state predicate, a monitor process has to be able to evaluate such predicates. For example, in the monitor automaton in Fig. 4, for the program in Fig. 1, if the current monitor automaton state is q_1, upon the occurrence of a local event in P_1, monitor process M_1 has to evaluate predicates that label outgoing transitions $q_1 \rightarrow q_\perp$ and $q_1 \rightarrow q_0$. For transition $q_1 \rightarrow q_0$, monitor process M_1 can verify proposition $(\texttt{x1} = 10)$ locally, hence, no need to communicate with monitor process M_2. On the contrary, for transition $q_1 \rightarrow q_\perp$, M_1 can only verify proposition $(\texttt{x1} \neq 10)$. Thus, if $(\texttt{x1} \neq 10)$ is true, M_1 has to evaluate $(\texttt{x2} < 15)$ as well, by consulting M_2.
   To address this problem we adapt the distributed computation *slicing* algorithm in [5] for distributed online detection of conjunctive predicates[1]. Computation slicing [10] is a technique to find abstract representations, called *slices*, that encode all concurrent and consecutive global states that satisfy a predicate without explicitly enumerating them. For example in Fig. 1, slices for predicate $(\texttt{x1} \geq 0 \ \wedge\ \texttt{x2} \neq 20)$ are represented by the set of events $\{e_1^1, e_1^2\}$ and $\{e_1^1, e_1^2, e_2^1\}$. If a slicer is distributed, then different processes may output different event-sets of the slice [5].
   In the context of runtime monitoring using a monitor automaton, we need to detect different predicates at different times, based on the current state of the monitor automaton and the outgoing transitions from that state. Hence, we only need to detect concurrent global states and not consecutive ones, which results in the difference between our predicate detection algorithm and the distributed slicing algorithm in [5]. For example, in Fig. 4, if the current monitor automaton state is q_0, then we need to detect predicates $((\texttt{x1} \geq 5) \wedge (\texttt{x2} \geq 15) \wedge (\texttt{x1} \neq 10))$ and $((\texttt{x1} \geq 5) \ \wedge\ (\texttt{x2} < 15) \ \wedge\ (\texttt{x1} \neq 10))$. And, if the current state of the monitor is q_1, then we need to detect predicates $((\texttt{x1} \neq 10) \ \wedge\ (\texttt{x2} < 15))$ and $(\texttt{x1} = 10)$. Also, unlike the framework of [5], in our setting, we do not need to detect all consistent cuts of a slice. Thus, our modified algorithm is on one hand, more sophisticated, as the set of predicates for detection changes over time, and on the other is simpler, as all consistent cuts are not required to be stored.
2) *(Concurrent events)* As mentioned earlier, existence of concurrent events results in the possibility of having

[1]We emphasize that detecting only conjunctive predicates does not impose any restrictions, since transitions in LTL$_3$ monitor automata are only labeled by conjunctive predicates (i.e., monitor transition labeled by disjunctive predicates are handled by splitting them into multiple transitions, one per each disjunct).

multiple global-state traces. In order to prevent exploring all possible interleavings, we only explore the global-state traces that can trigger a transition (excluding self-loops) in the monitor automaton. Given the current state of the monitor automaton, if different concurrent events enable more than one monitor transition at the same time, then we keep all possible next states of the monitor automaton as current monitor states. We call this *forking*. For example, in the monitor automaton in Fig. 4, for the program in Fig. 1, events e_2^1 and e_1^2 are concurrent and as explained in Section IV, paths that go through these events result in different verdicts. Hence, for each such path, its corresponding verdict is stored. Note that the algorithm does not attempt to explore all possible orders of the concurrent events. It rather attempts to explore orders that enable outgoing transitions of the current state of monitor automaton. Also note that we avoid exploring self-loops transitions since the monitor automaton state does not change when a self-loop is enabled. We emphasize that since the number of states LTL$_3$ monitors even for very complex properties is handful (normally less then five), keeping all possible verdicts does not lead to any implementation issues.

3) *(Merging verdicts)* Obviously forking monitor automaton states can lead to degrading performance, since more predicates have to be evaluated. Thus, monitor processes attempt to merge monitor states, if starting from a set of possible monitor states, enabled transitions reach the same monitor state. After merging, the maximum number of global-state traces should be bounded by the number of states in the monitor automaton. For example, in the monitor automaton in Fig. 4, if the current set of monitor states is $\{q_0, q_1\}$, and both outgoing transitions to $q_\perp$ are enabled, this yields to merging the two current monitor states into $\{q_\perp\}$.

B. Detailed Description of the Algorithm

We assume the reader is familiar with the notion of *logical clocks* [8]. In a distributed program $\mathcal{D} = \{P_1, P_2, \cdots, P_n\}$, a vector $VC_k^i = \langle v_1, v_2, \ldots, v_n \rangle$ is the *vector clock* locally stored in process P_i, $1 \leq i \leq n$, at or after the k^{th} event occurs in P_i, and before the $(k+1)^{th}$ event occurs. v_j is the value of the logical clock of process P_j, $1 \leq j \leq n$, as per the knowledge of process P_i. In other words, $VC_k^i[i] = k$, implies that VC_k^i is a vector clock that is aware of the occurrence of the k^{th} local event. When a process P_i sends a message to a process P_j, the vector clock of P_i is piggybacked with the message.

In a process P_i, we represent an event e by a tuple $e = \langle T, D, VC, sn \rangle$, where $T \in \{\mathsf{send}, \mathsf{receive}, \mathsf{internal}\}$ is the type of the event, D is the valuation of all local variables $D = \{(v, t) \mid v \in V \text{and } t \text{ is the value of } v\}$, $VC = VC_k^i$ is the vector clock of process P_i when the event e_k^i occurs, and sn is the sequence number of the event.

Monitoring Loop (Algorithm 1)

Let a *global view* be a tuple $gv = \langle gstate, gcut, q, tokens, p\_events, p\_trans \rangle$ where:

- $gstate$ is a vector of length n, containing the last state of each process known to M_i.
- $gcut$ is the vector clock that represents the consistent cut that the monitor attempts to reconstruct.
- q is the current state of the monitor automaton.
- $tokens$ is the set of token messages sent to other monitor processes (but not yet returned).
- p_events is the queue of pending events occurred in process P_i, while $tokens$ was not empty.
- p_trans is a set of transitions that M_{owner} requested $M_{destination}$ to evaluate for gv.

The algorithm first invokes Procedure INIT (Line 1). In particular, *history*, a vector representing the sequence of the local events occurred in P_i, is initialized to the empty sequence. Waiting tokens w_tokens, the set of the pending messages for processing received from other monitor processes, is initialized to the empty set. The initial global view gv_0 contains the initial global state. Then, we compute the next state of the monitor automaton by applying the global initial state on the transition relation δ from initial monitor automaton state q^0. In Lines 3 –6, M_i receives token messages from other monitor processes (respectively, reads local events of P_i) in an infinite loop in a non-blocking fashion and dispatches them to RECEIVETOKEN (respectively, RECEIVEEVENT).

Local Event Handler (Algorithm 2)

RECEIVEEVENT($event$): The goal of this procedure is to process local events only for global views whose pending events queue has already been processed (i.e., it is empty). To this end, first, M_i adds $event$ to its history. Then, in Lines 3 – 7, M_i invokes PROCESSTOKEN for each $token$ in w_tokens, where $token.newTargetEventNumber = event.sn$. A waiting *token* is a token sent by other monitor processes, to get information about the local state of P_i. A token is a tuple $t = \langle owner, destination, targetEventNumber, gcut, p\_trans, conjuncts\ targetProcessState, sent, returned \rangle$ where:

- $owner$ is the process that created t.
- $destination$ is the process that t is sent to.
- $targetEventNumber$ is the original event sn in process $P_{destination}$ that t targets.
- $newTargetEventNumber$ is the modified event sn in process $P_{destination}$ which the token is waiting for.
- $gcut$ is the vector clock that represents the logical clock at the event that triggered sending the token.
- f_trans is a set of transitions that are forbidden by $P_{destination}$.
- $conjuncts$ is a set of conjuncts containing the conjuncts owned by $P_{destination}$ which appears in f_trans's predicates.
- $targetProcessState$ is the local state of $P_{destination}$ retrieved by the token after returning from $P_{destination}$.
- $sent$ / $returned$ are flags set to $true$ when the token is sent to / received from $P_{destination}$, respectively.

In Line 8, M_i attempts to merge all global views by invoking MERGESIMILARGLOBALSTATES (described below). In Lines 9 – 15, M_i calls PROCESSEVENT for each global view gv whose set of tokens is empty. If $gv.tokens$ is non-empty,

Algorithm 1 Monitor Process M_i

Input: Monitor automaton $\mathcal{A}^{\varphi} = (Pred, Q, q^0, \delta, \lambda)$ and initial global state $init\_gstate$.

1: INIT();
2: **while** 1 **do**
3: $m \leftarrow$recv();
4: **if** m is not empty **then** RECEIVETOKEN(m); **end if**
5: $e \leftarrow$ read();
6: **if** e is not empty **then** RECEIVEEVENT(e); **end if**
7: **end while**

8: **procedure** INIT
9: $history \leftarrow \langle\rangle$; $w\_tokens \leftarrow \{\}$;
10: $gv_0.gstate \leftarrow init\_gstate$; $gv_0.q \leftarrow \delta(q^0, gv_0.gstate)$;
11: $gv_0.p\_trans \leftarrow \{\}$; $gv_0.tokens \leftarrow \{\}$; $GV \leftarrow \{gv_0\}$;
12: **end procedure**

Algorithm 2 Local Event Handler in M_i

1: **procedure** RECEIVEEVENT($event\ e$)
2: $history[e.sn] \leftarrow e$;
3: **for each** $t \in w\_tokens$ **do**
4: **if** $t.newTargetEventNumber = e.sn$ **then**
5: PROCESSTOKEN(t, e); $w\_tokens \leftarrow w\_tokens \setminus \{t\}$;
6: **end if**
7: **end for**
8: MERGESIMILARGLOBALSTATES();
9: **for each** $gv \in GV$ **do**
10: $gv.p\_events.enqueue(e)$;
11: **if** $gv.tokens = \{\}$ **then**
12: $e' \leftarrow gv.p\_events.dequeue()$;
13: PROCESSEVENT(gv, e');
14: **end if**
15: **end for**
16: **end procedure**

17: **procedure** PROCESSEVENT($GV\ gv, event\ e$)
18: $gv.gcut[i] \leftarrow e.VC[i]$; $gv.gstate[i] \leftarrow \mathfrak{s}(e)$;
19: **if** gv is consistent **then**
20: $q_n \leftarrow gv.q \leftarrow \delta(gv.q, gv.gstate)$;
21: **if** $\lambda(q_n) \in \{\bot, \top\}$ **then** Declare "satisfaction/violation";
22: **end if**
23: **end if**
24: CHECKOUTGOINGTRANSITIONS(gv, e,);
25: **end procedure**

26: **procedure** CHECKOUTGOIGTRANSITIONS($GlobalView\ gv, event\ e$)
27: $consult \leftarrow \langle\rangle$;
28: **for each** q', $pred$ st. $(gv.q \neq q') \wedge (gv.q \xrightarrow{pred} q') \in \delta$ **do**
29: $transition \leftarrow gv.q \xrightarrow{pred} q'$;
30: $par \leftarrow$ GETPARTICIPATINGPROCESSES($pred$);
31: $forb \leftarrow$ GETFORBIDDINGPROCESSES(gv, q');
32: **if** $P_i \notin forb$ **then**
33: $incon \leftarrow$ GETINCONSISTENTPROCESSES(gv);
34: **for each** $P_j \in forb \cup (par \cap incon)$ **do**
35: $gv.p\_trans \leftarrow gv.p\_trans \cup \{transition\}$;
36: $consult[j] \leftarrow consult[j] \cup \{transition\}$;
37: **end for**
38: **end if**
39: **end for**
40: **for** $j = 1 \rightarrow n$ **do**
41: **if** $consult[j] \neq \{\}$ **then**
42: let t be a token with unique id; $t.gv \leftarrow gv$;
43: $t.owner \leftarrow P_i$; $t.destination \leftarrow P_j$;
44: $t.targetEventNumber \leftarrow gv.gcut[j] + 1$;
45: $t.gcut \leftarrow e.VC$; $t.sent \leftarrow false$;
46: $t.conjuncts \leftarrow$GETCONJUNCTS($consult[j]$);
47: $t.f\_trans \leftarrow consult[j]$; $gv.tokens \leftarrow gv.tokens \cup \{t\}$;
48: **end if**
49: **end for**
50: $token \leftarrow$ GETTOKENWITHMAXCONJUNCTS($gv.tokens$);
51: SEND($token, token.destination$); $token.sent \leftarrow true$;
52: **end procedure**

Algorithm 3 Token Handler in M_i

1: **procedure** RECEIVETOKEN($token\ t$)
2: **if** $t.owner = i$ **then**
3: $token \leftarrow$ GETLOCALTOKEN($t.id$); $gv \leftarrow token.gv$;
4: $token.returned \leftarrow true$; $token.gcut \leftarrow t.gcut$;
5: $token.targetProcessState \leftarrow t.targetProcessState$;
6: **for each** $trans \in token.f\_trans$ **do**
7: // get other tokens sent for same transitions
8: $trans\_tokens \leftarrow$ GETTOKENSFORSAMETRANSITION($gv.tokens, trans$)
9: // If all tokens returned & transition conjuncts evaluated to true
10: **if** ENABLED($trans\_tokens$)$\wedge$CONSISTENT($trans\_tokens$) **then**
11: // copy tokens target process state to gv
12: UPDATEGV($gv, trans\_tokens$);
13: $gv_{n_1} \leftarrow gv_{n_2} \leftarrow gv$; $gv_{n_1}.q \leftarrow gv_{n_2}.q \leftarrow trans.to$;
14: **if** $\lambda(gv_{n_1}.q) \in \{\bot, \top\}$ **then** Declare "satisfaction/violation";
15: **end if**
16: $gv_{n_1}.tokens \leftarrow gv_{n_2}.tokens \leftarrow \{\}$;
17: $gv_{n_1}.p\_trans \leftarrow gv_{n_2}.p\_trans \leftarrow \{\}$;
18: $gv.p\_trans \leftarrow gv.p\_trans \setminus \{trans\}$;
19: $GV \leftarrow GV \cup \{gv_{n_1}, gv_{n_2}\}$;
20: PROCESSEVENT($gv_{n_1}, gv_{n1}.p\_events.dequeue()$)
21: PROCESSEVENT($gv_{n_2}, history[gv_{n_2}.gcut[i]]$);
22: **else if** $trans$ is disabled **then**
23: $gv.p\_trans \leftarrow gv.p\_trans \setminus \{trans\}$;
24: SENDREMAININGTOKENS(gv);
25: **else**
26: // Transition is still pending on other tokens.
27: SENDREMAININGTOKENS(gv);
28: **end if**
29: **end for**
30: **if** $gv.p\_trans = \{\}$ **then**
31: **if** gv had at least one enabled transition **then** $GV \leftarrow GV \setminus \{gv\}$;
32: **else** $gv.tokens \leftarrow \{\}$;
33: PROCESSEVENT($gv.p\_events.dequeue()$); **end if**
34: **else**
35: $token \leftarrow$ GETTOKENWITHMAXCONJUNCTS($gv.tokens$);
36: send($token, token.destination$); $token.sent \leftarrow true$;
37: **end if**
38: **else if** $t.owner \neq i$ **then**
39: **if** $(\exists k : history[k].sn = t.targetEventNumber)$ **then**
40: // to keep track of the original target event number
41: $t.newTargetEventNumber \leftarrow t.targetEventNumber$;
42: PROCESSTOKEN(t, e);
43: **else**
44: $w\_tokens \leftarrow w\_tokens \cup \{t\}$;
45: **end if**
46: **end if**
47: **end procedure**

48: **procedure** PROCESSTOKEN($token\ t, event\ e$)
49: **if** $(e.VC \mid\mid t.gcut)$ **then**
50: EVALUATETOKEN(t, e);
51: **else**
52: **for each** $conjunct \in t.conjuncts$ **do**
53: $conjunct.eval \leftarrow false$;
54: **end for**
55: //return the event at the original $t.targetEventNumber$
56: $e' \leftarrow history[t.targetEventNumber]$;
57: $t.targetProcessState \leftarrow \mathfrak{s}(e')$;
58: $t.gcut \leftarrow e'.VC$; send($t, t.owner$);
59: **end if**
60: **end procedure**

61: **procedure** EVALUATETOKEN($token\ t, event\ e$)
62: EVALUATECONJUNCTS($e, t.conjuncts$)
63: **if** /* at least one conjunct satisfied*/ **then**
64: $t.gcut \leftarrow e.VC$;
65: $t.targetProcessState \leftarrow \mathfrak{s}(e)$; send($t, t.owner$);
66: **else**
67: //wait for a satisfying concurrent state.
68: $t.newTargetEventId \leftarrow t.newTargetEventId + 1$;
69: $w\_tokens \leftarrow w\_tokens \cup \{t\}$;
70: **end if**
71: **end procedure**

then e is enqueued in $gv.p\_events$ as a pending event to be processed later.

MERGESIMILARGLOBALVIEWS(t, $event$): This procedure merges global views that have the same monitor automaton state and same logical time. Merging only involves taking the union of the set elements of the two global views.

PROCESSEVENT(gv, $event$): The goal of this procedure is to compute the next state of the monitor automaton, given a local event e and a global view gv. Recall that, depending upon the predicates that label outgoing transitions of the current state of the monitor automaton, the next monitor automaton state can be computed either locally or M_i has to consult other monitor processes. To this end, In Line 18, M_i updates $gv.gcut[i]$, so that gv looks for global views that update the monitor automaton state with events concurrent with e. Then, M_i updates the i^{th} entry in $gv.gstate$ with the new local state observed after the occurrence of e. In Line 19, M_i checks the consistency of $gv.gstate$ for each process P_j by evaluating $(gv.gstate[j].VC \parallel e.VC)$. If $gv.gstate$ is consistent for all processes, then in Line 20, M_i updates the monitor automaton state. Finally, in Line 24, M_i invokes CHECKOUTGOINGTRANSITIONS, described next.

CHECKOUTGOINGTRANSITIONS(gv): The goal of this procedure is to identify concurrent events that can potentially enable outgoing transitions of the monitor automaton. First, vector $consult$ of length n is initialized. Then, M_i repeats Lines 29 – 37 for each outgoing $transition$ from $gv.q$. In Line 30, M_i identifies the set of processes participating in the predicate $pred$ labeling $transition$[2]. Next, M_i identifies the set of *forbidding* processes; i.e., processes that cause $gv.gstate$ to currently falsify $pred$. If P_i is not a forbidding process, then $transition$ may be enabled. In Line 33, M_i constructs the list of inconsistent processes. Now, M_i has to consult all forbidding as well as inconsistent participating processes in the predicate labeling $transition$. Thus, $transition$ is added to each entry corresponding to a process in the vector $consult$. Also, $transition$ is added to $gv.p\_trans$, so that gv can keep track of pending transitions. In Lines 40 – 49, M_i constructs a token t to be sent to each process P_j with non-empty transitions set in the vector $consult$. Token t targets the event next to the last known event in $gv.gcut[j]$. It also carries both the set of all $conjuncts$ from different transitions required to be verified by M_j and the set of forbidding transitions in $consult[j]$, initially the evaluation for each conjunct is $false$. Then, in Lines 50 – 51, M_i selects the process with maximum number of conjuncts and sends a token to it first, if the token is returned with conjuncts evaluated to $true$, then a next token (if exists) will be sent. A copy of the token is added to the set $gv.tokens$. Note that we consolidate the sending of tokens from different global views at the end of each RECEIVEEVENT to decrease the number of sent messages, which indicates that the maximum number of sent tokens after each event in the worst case is $n-1$.

Token Handler (Algorithm 3)

[2]Recall that a predicate consists of local atomic propositions of different processes

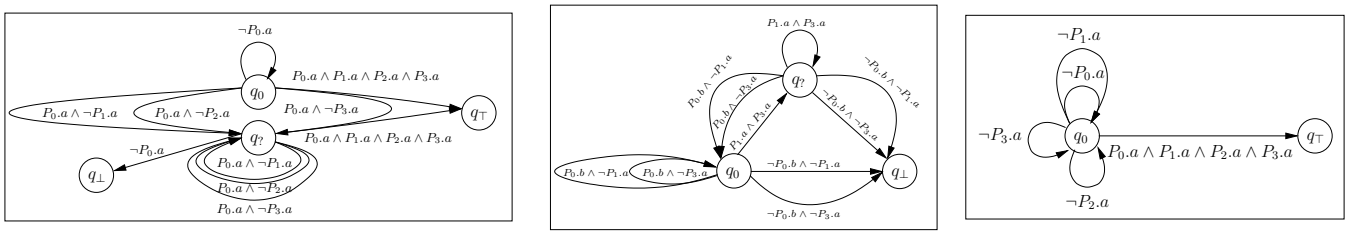
RECEIVETOKEN(t): The goal of this procedure is twofold, (1) enabling/disabling monitor automaton transitions based on the evaluation of the received token, if M_i is the owner of the token, and (2) evaluating the transition that M_j has requested M_i to evaluate. If M_i is the owner of t, then, first, we retrieve its local copy $token$ using procedure GETLOCALTOKEN and then update it with the information that t collected about $M_{t.destination}$ in (Line 5). Recall, that $token$ has multiple pending transitions in the set $token.f\_trans$, hence for every transition whose conjuncts are evaluated to $true$, we do the following: (1) retrieve the tokens holding conjuncts for the same transition for other processes using procedure GETTOKENSFORSAMETRANSITION and if all the $conjuncts$ returned with $true$ evaluation, then the transition is enabled, (2) then two new global view gv_{n_1} and gv_{n_2} are created and added to the set of global views GV (i.e., forking), and (3) gv_{n_1} starts processing pending events (inherited from gv), while gv_{n_2} looks for further possible outgoing transition given the new automaton state and the current global state. In Lines 22 – 24, if at least one conjunct for the transition returned with $eval = false$, then the transition is disabled and the corresponding conjuncts are removed from the pending tokens. In Lines 25 – 27, the next token for the process with the maximum number of $conjuncts$ is sent using procedure SENDREMAININGTOKENS. $gv.p\_trans$ is empty, indicating that all outgoing transitions have been processed and at least one transition was enabled, then gv is removed from the set of global views. If all the transitions were disabled, M_i clears $gv.tokens$ and processes pending events. If M_i is not the owner of t (Lines 38 – 46), then M_i searches for an event with sequence number $t.targetEventNumber$. If the requested event has already occurred, then PROCESSTOKEN is invoked for t and the requested event. Otherwise, t is added to the waiting tokens set.

PROCESSTOKEN(t, $event$): The goal of this procedure is to decide whether t should be returned back to $t.owner$, or should keep waiting at M_i. In Line 49, M_i checks whether or not t is concurrent with e. If so, then EVALUATETOKEN is invoked for t (described below). Otherwise, it could be the case that e is aware of an event in $M_{t.owner}$ that happened after $M_{t.owner}$ has already sent t. In this case, t should be sent back to $M_{t.owner}$ with $t.eval = false$, therefore, in Lines 55 –58, M_i updates the token with the details of the event at the original $t.targetEventNumber$ to avoid requesting its evaluation again, then the token is sent back.

EVALUATETOKEN(t, $event$): This procedure evaluates each $conjunct$ in $t.conjuncts$. If at least one $conjunct$ is satisfied, then t is returned back to $M_{t.owner}$. Otherwise, $t.newTargetEventNumber$ is incremented and t is added to w_tokens and waits for the next event. Also, t stays at M_i until it is no longer concurrent with the new target event or a satisfying event concurrent with t is found.

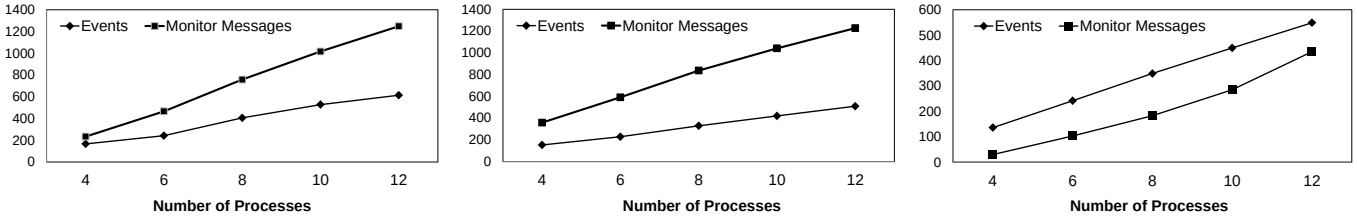
VI. EXPERIMENTAL RESULTS

In this section, we present the results of a set of experiments to evaluate our monitoring algorithm. Subsection VI-A introduces our case study, while Subsections VI-B and VI-C describe the experimental settings and result, respectively.



(a) Property 1 (b) Property 2 (c) Property 3

Fig. 5: Monitor automata for 4 UAVs.



(a) Property 1 (b) Property 2 (c) Property 3

Fig. 6: Monitoring overhead: Number of monitoring messages vs. number of events

A. Case Study

We tested our distributed monitoring algorithm on a simulation of a swarm coordinated unmanned aerial vehicles (UAVs) with a leader for rescue operations. The UAVs' form a grid-like shape with n rows and m columns. The leader is always at the top-left location (i.e., point $(0, 0)$). Each UAV has an initial position, velocity, and a set of checkpoints to visit. The leader issues a command to start the rescue operation once any UAV spots the rescue subject. Each UAV generates two types of events: (1) internal location updates, and (2) communication with neighbors in the grid. The latter are send events that occur after location update events; i.e., each UAV sends its updated location to its neighbors.

We monitor the following LTL properties:

- **Property 1**

$$\neg(leader.at\_checkpoint)\,\mathbf{U}\,((leader.at\_checkpoint)\,\mathbf{U}\,(\forall i : UAV_i.at\_checkpoint))$$

That is, if the leader arrives at the checkpoint, then it should stay at the checkpoint until all other UAVs arrive at the checkpoint. Note that each processes UAV_i participates in the property with the proposition $UAV_i.at\_checkpoint$. Fig. 5a shows the LTL_3 monitor automaton for Property 1 with 4 UAVs. For reasons of space, the leader is denoted by by P_0, other UAVs by $P_1 \cdots P_3$, while proposition $at\_checkpoint$ by a. Since all outgoing transitions (excluding self-loops) contain proposition from the leader process, we expect the leader monitor process to have higher load.

- **Property 2**

$$\neg(leader.rescue)\,\mathbf{U}\,(\forall i : odd(i) :: UAV_i.spots\_subject)$$

That is, the leader should not issue the rescue command until all the odd ranked UAVs make visual confirmations of spotting the rescue subject. Obviously, only the leader and the odd ranked processes participate in the property. Fig. 5b shows the LTL_3 monitor automaton for Property 2 for 4 UAVs, where proposition a denotes $spots\_subject$ and b denotes $rescue$. It is expected the leader monitor process to have higher load.

- **Property 3**

$$\Diamond(leader.at\_checkpoint \wedge \forall i : UAV_i.at\_checkpoint)$$

That is, eventually all the UAVs should be at the checkpoint. Fig. 5c shows the LTL_3 monitor automaton for Property 3 for 4 UAVs. Since the only outgoing transition in this property includes all processes, we expect all monitor processes to have equal load.

We note that the number of propositions that label the transitions of the automata in Fig. 5 increase as the number of processes increase in our experiments. We now show that due to the design of Algorithm 1, this increase will not blow up communication, computation, and local memory usage.

B. Experimental Settings

The algorithm is implemented using MPICH2 library implementation for the MPI standard (Message Passing Interface) in C. All the experiments were performed on a single machine dual core (Intel i5-3320M 2.6Ghz) with 7.5 GB memory. We consider the following experimental parameters: (1) the above 3 properties, (2) different number of UAVS; i.e., 4, 6, 8, 10, and 12 processes, and (3) different distributions for location updates. We measure the following metrics: (1) total number of monitoring messages sent from all monitor processes, (2) average delay in terms of the mean number of events delayed at the monitor process waiting for other monitor processes, and (3) memory utilization. We also study the scalability of our algorithm. Each setting is run three times to gain confidence in the results. We note that in all graphs the distribution of location updates is Poisson with mean $\mu = 3$ seconds. Our

results with other distributions (not shown in the paper) exhibit the same behavior.

C. Results

1) Monitoring Communication Overhead: Fig. 6 shows the monitoring overhead against the number of events (internal and communication) that need to be monitored.

- *(Property 1)* As can be seen, the number of monitoring messages grows in the linear order of the number of processes and (around 2x) events. This is because for every event the number of messages to be sent depends on the number of processes participating in the outgoing transition predicates. However, if an outgoing transition has a conjunctive predicate of propositions from different processes, the algorithm checks the evaluation of the propositions in sequence, and aborts checking if any proposition evaluates to false. Hence, the number of messages sent in the best case is 1 and in the worst case is the number of all other processes $n-1$.
- *(Property 2)* The graphs for Property 1 and Property 2 are similar although the even processes do not participate in Property 2. This is due to the fact that most of the time all the monitor processes are checking the evaluation of the predicates at the leader first and if evaluated to true, they proceed by evaluating next process.
- *(Property 3)* The growth in the number of monitoring messages is sublinear to the number of all events. This is because for the only outgoing transition from the initial state, monitor processes do not have to check other processes unless their local proposition is evaluated to true.

We argue that depending upon the application, since the number of messages grows linearly with the number of events, one can piggyback the monitoring messages on the processes communication and decrease the overhead significantly. Also, for a fixed number of processes, the size of the monitoring messages (i.e., tokens) remains constant at run time.

Fig. 7 (log scale) shows the monitoring messages per process in the experiment with 12 processes. As can be seen for the first 2 properties, the leader monitor process (P_0) sends significantly more messages than the other monitor processes. This is because since the two properties have multiple outgoing transitions with predicates involving propositions from the leader process, the other monitor processes depend on the leader monitor process to evaluate the transition predicates. On the other hand, the number of messages sent to evaluate Property 3 at each process is almost constant due to the fact that all processes participate equally in the outgoing transition.

2) Monitoring Memory Overhead: To demonstrate the bounded memory requirements of the algorithm, we ran the experiment long enough to generate a large number of events. To this end, we added an 'always' temporal operator to Property 3, so that there is no final state in the monitor automaton. Fig. 8 shows that (1) the average maximum memory allocated per process, (2) average total number of global views re-constructed per process (3) average total number of global views removed per process. As can be seen, although the number of events and monitoring messages are linearly

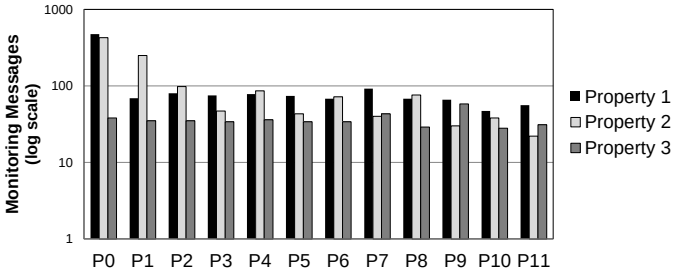


Fig. 7: The monitoring overhead for each process in the 12-process experiment.

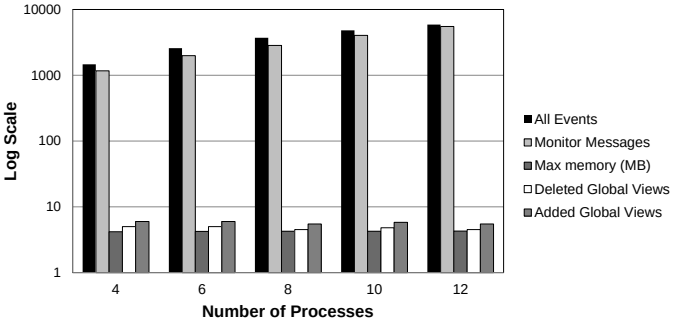


Fig. 8: Monitoring Memory Overhead.

increasing with the increase of the processes in the experiment, the memory usage of each process is almost constant. Also the average number of global views created and deleted are almost constant. Recall that a global view is created whenever a transition to a new monitor automaton state is enabled, and if the previous global view is not valid anymore (not a concurrent global view), the monitor process deletes it. Also, if two global views are at the same monitor automaton state, they will be merged into one global view and the other will be deleted.

3) Scalability: To demonstrate the scalability of the algorithm, we deploy our algorithm such that some monitoring processes are actively verifying the property and the remaining monitors processes are only responding to monitoring messages querying their local state. Fig. 9 shows the monitoring messages overhead for different number of active monitors in the 12-processes experiment for Property 3. As can be seen, the message overhead increases linearly indicating the scalability of the algorithm with adding monitoring processes. We chose Property 3 for this experiment since all the processes participate in the outgoing transition equally.

4) Detection Latency: Fig. 10 shows the average number of delayed events per monitor process for the different properties (i.e., the number of events occurred since the valuation of a property changes until it is detected by some monitor). We note that the average number of delayed events for the first 2 properties are much higher than the third property, this is due to the fact that Property 3 has a single outgoing transition, while the other properties have multiple outgoing transition for each monitor automaton state.

VII. Conclusion

In this paper, we proposed an algorithm for runtime verification of distributed applications. Our technique addresses the shortcomings of the state of the art. It works on asynchronous

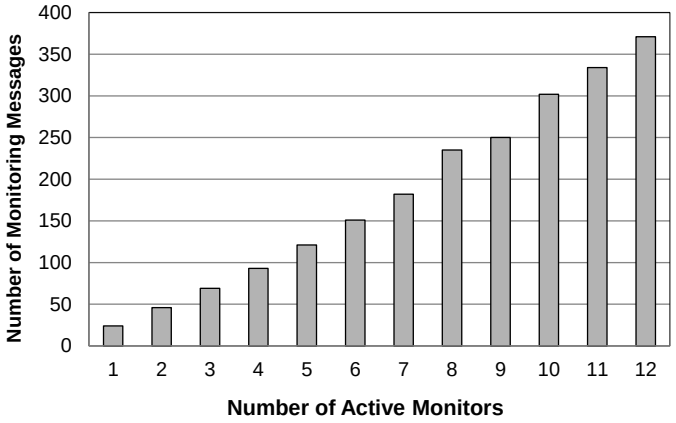


Fig. 9: The monitoring overhead for different number of active monitors in the 12 processes experiment for Property 3.

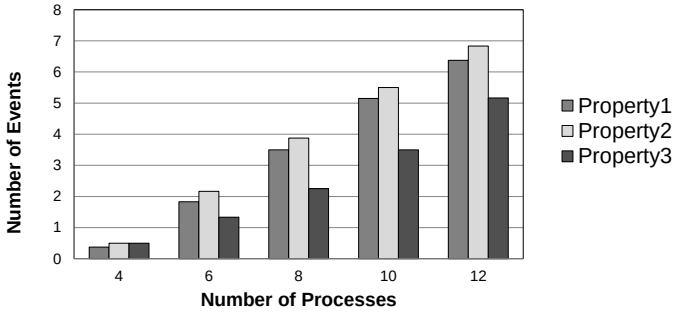


Fig. 10: Mean number of delayed events per process.

distributed systems, where a global clock is not assumed. Our specification language is full LTL and, hence, our method can monitor temporal properties as well. Finally, our algorithm is sound and complete, meaning that if a total order of events in the system under inspection can be constructed, its verification verdict will be determined by our algorithm as well (and vice versa). Our experiments to measure the communication and memory overhead of monitoring as well as the delay in evaluating different properties on a swarm of drones clearly shows that although our algorithm explores all possible interleavings for outgoing transitions, it does not result in an explosion in communication or local memory usage. In particular, due to the design of our Algorithm, monitoring overhead grows only in the *linear order* of the number of processes and events that need to be monitored.

We are currently working on optimizations to reduce the number of monitoring messages using static analysis. Another research avenue is to design monitoring algorithms for dynamic networks, where processes can join and leave. Finally, one can devise monitoring algorithms for properties that involve global arithmetic expressions and optimization objectives.

VIII. Acknowledgment

This work was partially sponsored by Canada NSERC Discovery Grant 418396-2012 and NSERC Strategic Grant 430575-2012.

References

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

[2] A. K. Bauer and Y. Falcone. Decentralised LTL monitoring. In *Proceedings of the 18th International Symposium on Formal Methods (FM)*, pages 85–100, 2012.

[3] S. Berkovich, B. Bonakdarpour, and S. Fischmeister. GPU-based runtime verification. In *Proceedings of the 27th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1025–1036, 2013.

[4] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, Feb 1985.

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

[6] C. Colombo and Y. Falcone. Organising LTL monitors over distributed systems with a global clock. In *Proceedings of the 14th International Conference on Runtime Verification (RV)*, pages 140–155, 2014.

[7] P. Fraigniaud, S. Rajsbaum, and C. Travers. On the number of opinions needed for fault-tolerant run-time monitoring in distributed systems. In *Proceedings of the 14th International Conference on Runtime Verification (RV)*, pages 92 – 107, 2014.

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

[9] F. Mattern. Virtual time and global states of distributed systems. In *Proceedings of the Workshop on Distributed Algorithms (WDAG)*, pages 215–226, 1989.

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

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

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

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

Appendix

In this section, we present the proof of correctness of Algorithm 1.

Lemma 1: Any token t created by M_i and sent to M_j for the purpose of constructing a consistent cut C for event e^i, will eventually return to M_i in finite time.

Proof: When M_i receives a local event e^i, it attempts to progress the state of the monitor automaton by constructing all concurrent consistent cuts that can enable any outgoing transition, such that $e^i \in FE_C$, i.e. the frontier of each constructed consistent cut C contains event e^i. Hence, M_i creates a token t for each conjunct in the transition predicate that can not be satisfied locally, and sends the created tokens in sequence to the corresponding forbidding monitor process that might satisfy the conjunct. When M_j receives the token, it searches for local events that can construct the desired consistent cut. Token t stays at M_j until either:

- The vector clock for the last event at M_j is greater than M_i's event e
- A consistent cut that satisfies the conjunct is found
- P_j terminates

Since we assume all processes are terminating processes, hence t eventually returns to M_i in finite time. Moreover, if t fails in finding a satisfying consistent cut (case 1 or 3), other tokens scheduled to be sent in sequence are not sent, since one

of the conjuncts is false, making the whole predicate false. ■

Lemma 2: The routine for constructing consistent cuts never deadlocks.

Proof: The routine for constructing consistent cuts for a transition predicate starts by: (1) sending out tokens in sequence to processes that forbid the transition predicate form becoming true, and then (2) the monitor process waits to receive the tokens back. As shown in the previous lemma, each token returns in finite time, hence the routine is guaranteed to never deadlock. ■

Lemma 3: Algorithm 1 never deadlocks.

Proof: The main functions of a monitor process M_i is: (1) receive token messages and (2) read the local state of process P_i. None of the previous operations are blocking. In case reading the local state of P_i results in truthfulness of a local proposition, at most $n-1$ tokens will be sent out in sequence. And as shown in Lemma 1, these tokens are sent back to M_i in infinite time. However, while waiting for the tokens to return, M_i enqueues the new local states read from process P_i and resumes its normal operation for other global views. In other words, in Algorithm 1 (and, consequently, all other procedures), no process is waiting forever for other processes. Therefore, the algorithm never reaches a deadlock situation. ■

Lemma 4: Algorithm 1 eventually terminates when it monitors a terminating program.

Proof: First, we note that our algorithm is designed for terminating programs, also, note that a terminating program only produces finite computations, hence, finite number of local events in all normal processes. In order to prove the lemma, let us imagine that when the program terminates, it sends a *stop* signal to all normal and monitor processes. When such a signal is received by a normal process, it will not produce new events. When received by a monitor process M_i, it starts processing all the events stored in its $history$ variable. There can be two cases with respect to the waiting tokens list w_tokens in M_i. If processing events in $history$ results in truthfulness of a local proposition for which there exists a waiting token in w_tokens, then the token will be sent back to the owner of the token with $true$ valuation for the local proposition. Otherwise, if $history$ becomes empty, then all waiting tokens should be sent back to the owners with value $false$ for the corresponding propositions. This would simply result in termination of all pending actions and, hence, the algorithm. ■

Lemma 5: In a monitor process M_i, with a global view gv pointing at consistent cut C and monitor automaton state q, if q has an outgoing transition $q \xrightarrow{p} q'$, then Algorithm 1 always forks to monitor automaton state q', if a consecutive consistent cut C' exists, where $C' \models p$.

Proof: The proof is divided into two subproofs: (1) if C' exists, M_i detects it, and (2) if M_i detects C', q' is added to the set of possible verdicts. The proof goes as follows:

1) The first proof obligation can be discharged following the result in [5], where a slicer process is guaranteed to output the consistent cut C' that satisfies p (Theorem 3 in [5]). Intuitively, after the monitor process updates the global view gv with a new event e, there are two cases:
   - If gv has an enabled transition with label p, then the resulted consistent cut C' where $C' \models p$ is detected.
   - If gv has a possible enabled transition with label p, such that P_i is not in the forbidding processes for p, then M_i prepares and sends tokens to the forbidding processes in sequence, and once they all return to M_i (guaranteed by Lemma 1) and the truthfulness of p is verified, M_i updates gv with the forbidding processes new states and vector clocks, and the resulted consistent cut C' where $C' \models p$ is detected.
2) The proof of the second part is straightforward. If such C' is indeed reached, where $C' \models p$, it means that all tokens associated with p (if any) has been returned to the owner. This implies that Line 15 in Algorithm 3 executes, which in turn, adds a new global view gv' to GV, pointing at C' and q'. Note that gv will be deleted after all outgoing transitions have been checked and at least one transition was enabled.

■

Corollary 1: Let $\mathcal{L}$ be a computation lattice and p be a predicate. In a monitor process M_i, current monitor automaton state q and an outgoing transition $q \xrightarrow{p} q'$, Algorithm 1 forks to state q', in all consistent global states g of $\mathcal{L}$, where $g \models p$.

Proof: Proof trivially follows from Lemma 5. ■

Theorem 1: Algorithm 1 solves the problem stated in Section IV.

Proof: Let $\mathcal{L}$ be a computation lattice. Following Corollary 1, any predicate that is met in a global state of $\mathcal{L}$ is detected, a monitor process always updates its set of possible states with respect to all enabled outgoing transitions in a sound fashion. This implies that the valuation of the corresponding LTL_3 property with respect to all paths in $\mathcal{L}$ are added to GV. This clearly includes the valuation of the total order constructed by an oracle as well. ■