Crash-Resilient Decentralized Synchronous Runtime Verification

Shokoufeh Kazemlou
Department of Computing and Software
McMaster University, Canada
Email: kazemlos@mcmaster.ca

Borzoo Bonakdarpour
Department of Computer Science
Iowa State University, USA
Email: borzoo@iastate.edu

***Abstract*—*Runtime verification* is a technique, where a *monitor* process extracts information from a running system in order to detect executions violating or satisfying a given correctness specification. In this paper, we consider runtime verification of synchronous distributed systems, where a decentralized set of monitors that only have a partial view of the system are subject to crash failures. In this context, it is unavoidable that monitors may have different views of the underlying system, and, therefore, have different opinions about the correctness property. We propose an automata-based synchronous monitoring algorithm that copes with t crash monitor failures. Moreover, local monitors do not communicate their explicit reading of the underlying system. Rather, they emit a *symbolic verdict* that efficiently encodes their partial views. This significantly reduces the communication overhead. To this end, we also introduce an (offline) SMT-based monitor synthesis algorithm, which results in minimizing the size of monitoring messages[1].**

I. Introduction

In the past three decades, achieving system-wide dependability and reliability has substantially benefited by incorporating rigorous formal methods. Such reliability and dependability is especially critical in the domain of distributed systems that inherently consist of complex algorithms and intertwined concurrent components. Given the complexity of today's computing systems, deploying exhaustive verification techniques such as *model checking* and *theorem proving* come at a high cost in terms of time, resources, and expertise. In many cases, formal verification may not even scale to a realistic size to analyze the system's correctness. Moreover, exhaustive verification techniques may overlook bugs due to unanticipated stimuli from the environment, internal bugs in virtual machines, or operating systems as well as hardware faults. On the other side of the spectrum, *testing* is a best-effort method to examine the correctness, which scrutinizes only a subset of behaviors of the system. Due to its under-approximate nature, testing often does not reveal obscure corner cases that complex systems may reach at run time. In a distributed setting, the inherent uncertainty about an exponential number of orderings of events makes testing techniques often blind to concurrency bugs.

Runtime verification (RV) is a lightweight popular technique [1], [2], where a *monitor* continually inspects the health of a system under inspection at run time with respect to a formally specified set of properties. The formal specification is normally in the form of some language with clear syntax and semantics, such as regular expressions or some form of temporal logics. RV is a crucial complement to exhaustive verification and testing. Monitoring distributed systems and distributed monitoring has recently gained traction [3]–[13] as a technique to discover latent bugs in concurrent settings. Efficient detection of such bugs is quite challenging for three reasons: (1) a monitor may have to reconstruct a large set of serializations from its observations, (2) local monitors have only a partial view of the entire system and, (3) monitors like any other process may be subject to faults. While the first difficulty has been studied in various forms [3], [4], [9]–[13], the body of literature on the latter two is limited to the results in [5], [14], [15], where the authors show that runtime monitors need to employ enough number of *opinions* (instead of the conventional binary valuations) to consistently reason about distributed tasks in a consistent manner. These results are generally in an asynchronous wait-free setting, which is a bit far from reality of widely used point-to-point message passing networks.

With this motivation, in this paper, we introduce an RV technique for fault-tolerant decentralized monitoring that inspects an underlying distributed system. Our RV framework has the following properties:

- We assume that a set of monitors are distributed over a *synchronous* communication network. The network is a complete graph allowing all monitors to communicate with each other using point-to-point message passing in synchronous rounds.
- Each monitor is subject to *crash* failures. A crashed monitor halts permanently and never recovers.
- Each monitor has only a *partial view* of the underlying system. More specifically, given a set AP of atomic propositions that describe the global state of the system, each monitor can read an arbitrary proper subset of AP.
- The formal specification language is the *linear temporal logic* (LTL) [16], where formulas are inductively constructed using the propositions in AP.

Our goal is to design an algorithm with the following features:

- *Soundness.* Upon termination, all local monitors com-

[1]This is an extended version of the paper appeared in the 37th IEEE International Symposium on Reliable Distributed Systems (SRDS'18)

pute the same monitoring verdict as a centralized monitor that can atomically observe the global state of the system.
- *Low overhead.* One way for local monitors to share their observation of the underlying system is to communicate their reading of AP with each other in synchronous communication rounds. However, this will incur a message size of $O(|\mathsf{AP}|)$, which is exponential in the number of system variables. Thus, our goal is to find a more efficient way for local monitors to communicate their partial observations without compromising soundness.

Our main contribution in this paper is a decentralized synchronous t-resilient RV algorithm, where t is the upper bound on the number of crash failures of monitors. Given a new global state, each monitor process computes a *symbolic* representation of its reading of AP and starts $t+1$ rounds of synchronous communication with other monitors in the network. The number of rounds is inspired by solutions to the consensus problem in synchronous networks, though in our problem, the monitors need to agree on a verdict that is not known a priori and they collaboratively compute the verdict during the rounds of communication. The symbolic representation is computed by employing a deterministic finite state automaton for monitoring formulas in the linear temporal logic (LTL). We show that the monitor automaton as constructed using the algorithm in [17] cannot guarantee soundness in a distributed synchronous setting. Subsequently, we propose an algorithm that transforms the automaton into another by adding a minimum number of extra states and transitions to address cases where local monitors run into indistinguishable states due to their partial observations. In order to minimize the size of the transformed automaton, we formulate an offline optimization problem in *satisfiability modulo theory* (SMT)[2]. The size of the SMT instance is expected to be small, as most practical LTL formulas are known to have at most just a few nested temporal operators. Even if the size of the transformed monitor is not minimized the size of each message will be $O(\log|\mathsf{AP}|)$. In short, our RV framework has message complexity

$$O\Bigg(\log\Big(|\mathcal{M}_3^{\varphi}|\cdot|\mathsf{AP}|\Big)n^2\Big(t+1\Big)\Bigg).$$

for evaluating each global state, where n is the number of distributed monitors and $\mathcal{M}_3^{\varphi}$ denotes the finite state automaton for monitoring an LTL formula φ as constructed in [17]. An important implication of our results is that unlike the asynchronous fault-prone setting, where we need to increase the number of truth values in the specification language to design consistent distributed monitors [5], [14], [15], in this paper, we show that in a fault-prone synchronous setting, the number of truth values is irrelevant.

Organization: The rest of the paper is organized as follows. We introduce the preliminary concepts in Section II. Section III presents our model of computation for decentralized crash-resilient synchronous RV. We present the general idea behind our RV algorithm in Section IV and subsequently elaborate on the details in Section V. Related work is discussed in Section VI. Finally, we make concluding remarks and discuss future work in Section VII. All proofs appear in the appendix.

[2]Satisfiability Modulo Theories (SMT) are decision problems for formulas in first-order logic with equality combined with additional background theories such as linear arithmetic, arrays, bit-vectors, etc.

II. Preliminaries

In this section, we review the preliminary concepts.

A. Linear Temporal Logic

Let AP be a set of *atomic propositions* and $\Sigma = 2^{\mathsf{AP}}$ be the *alphabet*. We call each element of Σ an *event*. For example, for $\mathsf{AP} = \{a, b\}$, event $s = \{\}$ means that both propositions a and b are not true in s and event $s' = \{a\}$ means that only proposition a is true in s'. A *trace* is a sequence

$$s_0 s_1 s_2 \cdots$$

where $s_i \in \Sigma$, for every $i \geq 0$. The set of all finite (respectively, infinite) traces over Σ is denoted by Σ^* (respectively, Σ^ω). Throughout the paper, we denote finite traces by the letter α, and infinite traces by the letter σ. For a finite trace $\alpha = s_0 s_1 \cdots s_n$, by α^i, we mean trace suffix $s_i s_{i+1} \cdots s_n$ of α.

LTL Syntax: Formulas in the *linear temporal logic* (LTL) [16] are defined using the following grammar:

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

where $p \in \mathsf{AP}$ is an atomic proposition, $\mathcal{U}$ is the "until" operator, and $\bigcirc$ is the "next operator". Additionally, we allow the following operators as syntactic sugar, each of which is defined in terms of the above ones: $\mathsf{true} = p \vee \neg p, \mathsf{false} = \neg\mathsf{true}, \varphi_1 \wedge \varphi_2 = \neg(\neg\varphi_1 \vee \neg\varphi_2), \Diamond\varphi = \mathsf{true}\,\mathcal{U}\,\varphi$, and $\Box\varphi = \neg\Diamond(\neg\varphi)$, where $\Diamond$ and $\Box$ are the 'eventually' and 'always' temporal operators, respectively.

LTL Semantics: The semantics of LTL is defined with respect to infinite traces. Let $\sigma = s_0 s_1 \cdots$ be an infinite trace in Σ^ω, $i \geq 0$ be a non-negative integer, and $\models$ denote the *satisfaction* relation. The semantics of LTL is defined as follows:

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

Also, $\sigma \models \varphi$ holds iff $\sigma, 0 \models \varphi$ holds. For example, consider the following *request/acknowledgment* LTL formula:

$$\varphi_{ra} = \Box\Big(\neg a \wedge \neg r\Big) \vee \Big((\neg a\,\mathcal{U}\,r) \wedge \Diamond a\Big)$$

This formula requires that (1) if a request r is emitted, then it should eventually be acknowledged by a, and (2) an acknowledgment happens only in response to a request.

B. 3-valued LTL for Runtime Verification

The semantics of LTL is defined over infinite traces. In the context of runtime verification, since a system only generates finite traces, the standard LTL semantics does not seem to be the appropriate formalism. The 3-valued LTL (denoted LTL_3 [17]) allows us to reason about finite traces for verifying properties at run time with an eye on possible future extensions. The syntax of LTL_3 is identical to that of LTL and the semantics is based on three truth values:

$$\mathbb{B}_3 = \{\top, \bot, ?\}$$

where '$\top$' (respectively, '$\bot$') denotes that the formula is *permanently* satisfied (respectively, violated), no matter how the current execution extends, and '?' denotes the *unknown* truth value, i.e., there exists a future extension that can falsify the formula, and another that can truthify the formula.

Now, let $\alpha \in \Sigma^*$ be a non-empty finite trace. The truth value of an LTL formula φ with respect to α in the 3-valued semantics, denoted by $[\alpha \models_3 \varphi]$, is defined as follows:

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

For example, consider formula $\varphi = a\,\mathcal{U}\,b$ and the following three finite traces:

- $u_1 = \{a\}\{a\}\{a\}$
- $u_2 = \{a\}\{a\}\{a\}\{\}$
- $u_3 = \{a\}\{a\}\{a\}\{b\}$

Here, we have $[u_1 \models_3 \varphi] = ?$, as this finite trace can be extended to traces that result in violation or satisfaction of φ. Two such traces are u_2 and u_3, respectively, i,e., $[u_2 \models_3 \varphi] = \bot$ and $[u_3 \models_3 \varphi] = \top$.

Definition 1: The LTL_3 *monitor* of a formula φ is the unique deterministic finite state machine $\mathcal{M}_3^\varphi = \langle \Sigma, Q, q_0, \delta, \lambda \rangle$, where Q is a set of states, $q_0 \in Q$ 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 $\alpha \in \Sigma^*$. ■

In [17], the authors introduce an algorithm that takes as input an LTL formula and constructs as output an LTL_3 monitor. For example, Fig. 1 shows the LTL_3 monitor for formula $\varphi = a\,\mathcal{U}\,b$, where $\lambda(q_0) = ?$, $\lambda(q_\bot) = \bot$, and $\lambda(q_\top) = \top$. It is easy to observe that finite traces u_1, u_2, and u_3 terminate at monitor states q_0, $q_\bot$, and $q_\top$, respectively.

III. Model of Computation

An LTL_3 monitor as defined in Definition 1 can evaluate an LTL formula φ with respect to a finite execution, where each event represents the full view of the system under inspection. From now on, we refer to such events as *global events*, where the value of all propositions in the event is known. While this model is realistic in a centralized setting, it is too abstract in a distributed setting. We now present our computation model.

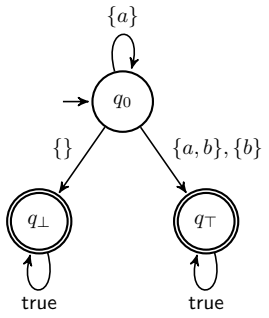

Fig. 1. LTL_3 monitor for $\varphi = a\,\mathcal{U}\,b$.

A. Overall Picture

We consider a distributed monitoring system comprising of a fixed number n of *monitor processes* $\mathcal{M} = \{M_1, M_2, \ldots, M_n\}$ that communicate with each other by sending and receiving messages through point-to-point bidirectional communication links[3]. We assume that the communication graph is synchronous and complete. Each communication link is reliable, that is, we assume no loss or alteration of messages. Each monitor process locally executes identical sequential algorithms. Each run of a monitor process consists of a sequence of *rounds* that are identified by the successive positive integers 1, 2, etc. The round number is a global variable and its progress is ensured by the synchrony assumption [18]. Each round is made up of three consecutive steps: *send*, *receive*, and *local computation*. The principle property of the round-based synchronous model is the fact that a message sent by a monitor M_i to another monitor M_j, for all $i, j \in [1, n]$, during a round r is received by M_j at the very same round r. Each monitor process can start a new round when the current is complete.

Throughout this section, the system under inspection produces a finite trace $\alpha = s_0 s_1 \cdots s_k$, and is inspected with respect to an LTL formula φ by a set of synchronous distributed monitor processes. Informally, our synchronous distributed monitoring architecture works as follows. For every $j \in [0, k]$, between each two consecutive global events s_j and s_{j+1}, each monitor process M_i, where $i \in [1, n]$:

1) reads the value of a subset of propositions in s_j, which results in a *partial* observation of s_j;
2) at every synchronous round, *broadcasts* a message containing its current observation of the underlying system, and then waits to receive similar messages from other monitor processes;
3) based on the messages received at each round updates its current observation by incorporating partial observations of other monitor processes, and composes a message to be sent at next round, and
4) finally, after $t+1$rounds of communication, evaluates φ and emits a truth value from $\mathbb{B}_3$, where t is the upper bound on the number of monitor process crash failures.

[3]To prevent confusion, we refer to monitors in $\mathcal{M}$ as 'monitor processes' and the one defined in Definition 1 as 'LTL_3 monitor'.

Data: LTL formula φ and finite trace $s_0 s_1 \cdots s_k$
Result: A verdict from $\mathbb{B}_3$

1 **for** $j = 0$ **to** k **do**
2 Let $\mathcal{S}_i^{s_j}$ be the initial partial view of the monitor;
3 $LS_i^1 \leftarrow \mu(\mathcal{S}_i^{s_j}, \varphi)$;
4 **for** $r = 1$ **to** $t + 1$ **do**
5 ***Send:*** broadcasts symbolic view LS_i^r;
6 ***Receive:*** $\Pi_i^r \leftarrow \{LS_j^r\}_{j \in [1,n]}$;
7 ***Computation:*** $LS_i^{r+1} \leftarrow LC(\Pi_i^r)$;
8 **end**
9 **end**
10 Emit a verdict from $\mathbb{B}_3$;

Algorithm 1: Behavior of Monitor M_i, for $i \in [1, n]$

B. Detailed Description

We now delve into the details of our computation model (see Algorithm 1). When an event s_j is reached in a finite trace $\alpha = s_0 s_1 \cdots s_k$, each monitor process $M_i \in \mathcal{M}$, where $i \in [1, n]$, attempts to read s_j (Line 2 in Algorithm 1). Due to distribution, this results in obtaining a partial view $\mathcal{S}_i^{s_j}$ defined next.

Definition 2: A *partial view* is a function $\mathcal{S} : \mathsf{AP} \mapsto \{\mathsf{true}, \mathsf{false}, \natural\}$, i.e, a mapping from the set of atomic propositions to values true, false, or $\natural$. The latter denotes an *unknown* value for a proposition. ■

Notice that the unknown value '$\natural$' for a proposition is different from the unknown truth value '?' in the LTL_3 semantics.

Definition 3: We say that a partial view $\mathcal{S}$ is *consistent* with a global event $s \in \Sigma$ (denoted $\mathcal{S} \sqsubseteq s$), if for every atomic proposition $p \in \mathsf{AP}$, we have:

$$(\mathcal{S}(p) = \mathsf{true} \Rightarrow p \in s) \wedge (\mathcal{S}(p) = \mathsf{false} \Rightarrow p \notin s).$$

■

Hence, in a partial view $\mathcal{S}$ is consistent with event s, if the value of an atomic proposition is *not* unknown, then it has to be consistent with s.

Monitor processes observe the system under inspection by reading partial views. We denote the partial view of a monitor process M_i from event $s \in \Sigma$ by $\mathcal{S}_i^s$ and assume that $\mathcal{S}_i^s \sqsubseteq s$. This implies that two monitors M_i and M_l cannot have inconsistent partial views of the same global event. That is, for any event s and partial views $\mathcal{S}_i^s$, $\mathcal{S}_l^s$, and for every $p \in \mathsf{AP}$, we have:

$$(\mathcal{S}_i^s(p) \neq \mathcal{S}_l^s(p)) \Rightarrow (\mathcal{S}_i^s(p) = \natural \vee \mathcal{S}_l^s(p) = \natural).$$

In Algorithm 1, one way for monitor processes to share their observation of the system is to communicate their partial views. This way, after several rounds of communication (due to the occurrence of faults), all monitor processes can construct the full global event. Although this idea works in principle, it is quite inefficient, as the size of each message will have to be at least $|\mathsf{AP}|$ bits. Our goal is to design a technique, where monitor processes can communicate their observations without sending and receiving their full observation of all atomic propositions. To this end, we introduce the notion of a *symbolic view* that intends to represent the partial view of a monitor processes M_i without losing information. We denote the symbolic view of a partial view $\mathcal{S}_i^s$ with respect to an LTL formula φ by $LS_i = \mu(\mathcal{S}_i^s, \varphi)$ (see Line 3 in Algorithm 1). In Section IV, we will present a concrete way of computing μ.

Let LS_i^r denote the symbolic view of monitor process M_i at the beginning of round r. In Line 5, each monitor process sends its current symbolic view to all other monitor processes and then receives the symbolic view of all monitor processes in Line 6. Let

$$\Pi_i^r = \left\{ LS_l^r \right\}_{l \in [1,n]}$$

be the set of all messages received[4] by monitor process M_i during round r. Then (Line 7), the monitor computes the new symbolic view from the messages it received using a function LC (described in detail in Section IV). This new view will be broadcast during the next round.

In order to achieve sound monitoring, we assume the full event in the system is observed by the set $\mathcal{M}$ of monitor processes. We call this assumption *event coverage*. More specifically, we say that a set of monitor processes *cover* a global event if and only if the collection of partial views of these monitor processes cover the value of the all atomic propositions.

Definition 4: A set $\mathcal{M} = \{M_1, M_2, \ldots, M_n\}$ satisfies *event coverage* for an event s if and only if for every $p \in \mathsf{AP}$, there exists $M_i \in \mathcal{M}$ such that $\mathcal{S}_i^s(p) \neq \natural$. ■

C. Fault Model

Each monitor process is subject to *crash* faults, i.e., it may halt and never recover. We assume that up to t monitor processes can crash, where $t < |\mathcal{M}|$. A monitor process may crash at any round. To ensure the event coverage, we assume that if there is a proposition $p \in \mathsf{AP}$, such that at round r monitor process M_i is the only monitor aware of p, then the message sent by M_i at round r, must be received by at least one non-faulty monitor in round r. This is a reasonable assumption and can be implemented by including redundant monitors. That is, there is enough number of monitors that ensure event coverage (e.g., by using triple modular redundancy).

D. Problem Statement

Our formal problem statement is the termination requirement for Algorithm 1. Roughly speaking, we require that when a non-faulty monitor process runs Algorithm 1 to the end, it

[4]We note that if some monitor process crashes while another monitor is receiving messages in Line 6, this monitor will not receive n messages as prescribed by the algorithm. In synchronous algorithms, by the synchrony assumption, a crash failure can be easily detected and hence, the accurate value of n can be determined for receiving messages.

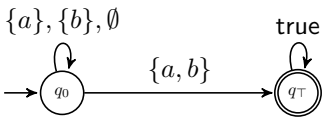

Fig. 2. LTL$_3$ monitor for $\varphi = \Diamond(a \wedge b)$.

emits a verdict that a centralized monitor that has global view of the system would compute. This termination condition is formally the following

$$\forall i \in [1,n] : M_i \text{ is non-faulty } \Rightarrow \nu_i = [\alpha \models_3 \varphi]$$

where α is a finite trace, φ is an LTL formula, and ν_i is the truth value emitted by monitor M_i at the end of running Algorithm 1.

It is easy to see that our decentralized synchronous monitoring problem, where monitor processes are subject to crash faults is in spirit similar to the uniform *consensus* problem [18]. The main difference is that in consensus, processes need to agree on one values that they own. In our problem, they should particularly agree on the value $[\alpha \models_3 \varphi]$, while none of the monitors necessarily has this value before the inner for-loop. In Section V, we will show that similar to synchronous consensus, if t monitors may fail, $t+1$ rounds of communication are sufficient to agree on the final verdict.

IV. The General Idea and Motivating Example

In Algorithm 1, we provided the skeleton of our synchronous monitoring algorithm. What remains to be done is identifying concrete μ and LC. Our general idea is described in the sequel and is reflected in Algorithm 2, which refines Algorithm 1.

A. Symbolic View μ

As mentioned in Section III, sharing explicit partial views is not space efficient, as each message will need at least $|\mathsf{AP}|$ bits. To tackle this problem, our idea is that each monitor process employs an LTL$_3$ monitor, as defined in Definition 1 and the symbolic view of a monitor process consists of the set of *possible* LTL$_3$ monitor states that corresponds to its partial view. Formally, let q be the current state of the LTL$_3$ monitor and $\mathcal{S}$ be the partial view of the monitor process. The set of possible next LTL$_3$ monitor states can be computed as follows:

$$\mu(\mathcal{S}, q) = \Big\{ q' \mid \exists s \in \Sigma. \big(\mathcal{S} \sqsubseteq s \wedge \delta(q,s) = q'\big) \Big\} \quad (1)$$

Recall that δ denotes the transition function in LTL$_3$ monitors. For example, consider the following LTL formula:

$$\varphi = \Diamond(a \wedge b).$$

The LTL$_3$ monitor of this formula is shown in Fig. 2, where $\lambda(q_0) = ?$ and $\lambda(q_\top) = \top$. Let us imagine that (1) a monitor process M_1 is currently in state q_0, (2) the global event is $s = \{a, b\}$, and (3) the current partial view of M_1 is $\mathcal{S}_1^s(a) = \mathsf{true}$ and $\mathcal{S}_1^s(b) = \mathsf{true}$. This implies that monitor M_1 considers $q_\top$ as the only possible next LTL$_3$ monitor state, i.e.,

$$\mu(\mathcal{S}_1^s, q_0) = \{q_\top\}.$$

Data: LTL$_3$ monitor $M_3^\varphi = \langle \Sigma, Q, q_0, \delta, \lambda \rangle$, finite trace $s_0 s_1 \cdots s_k$.
Result: Verdict from $\mathbb{B}_3$

1 $q_{current} \leftarrow q_0$;
2 **for** $j = 0$ **to** k **do**
3 Let $\mathcal{S}_i^{s_j}$ be the initial partial view of the monitor;
4 $LS_i^1 \leftarrow \mu(\mathcal{S}_i^{s_j}, q_{current})$; /* Equation (1) */
5 **for** $r = 1$ **to** $t+1$ **do**
6 ***Send:*** broadcasts symbolic view LS_i^r;
7 ***Receive:*** $\Pi_i^r \leftarrow \{LS_j^r\}_{j \in [1,n]}$;
8 ***Computation:*** $LS_i^{r+1} \leftarrow LC(\Pi_i^r)$; /* Equation (2) */
9 **end**
10 $q_{current} \leftarrow LS_i^{r+1}$;
11 **end**
12 **return** $\lambda(q_{current})$;

Algorithm 2: Updated behavior of Monitor M_i, for $i \in [1,n]$

However, considering another partial view $\mathcal{S}_1^s(a) = \mathsf{true}$ and $\mathcal{S}_1^s(b) = \natural$, monitor process M_1 will have to consider $\{q_0, q_\top\}$ as possible next LTL$_3$ monitor states. This is because it has to consider two possibilities for proposition b. That is,

$$\mu(\mathcal{S}_1^s, q_0) = \{q_0, q_\top\}.$$

We use μ as defined in Equation (1) to compute the concrete symbolic view in Line 4 of Algorithm 2.

B. Computing LC

Given a set of possible LTL$_3$ monitor states computed by μ, in Line 7 of Algorithm 2, each monitor process receives a set of possible states from all other monitors, denoted by LS_i^r for each monitor process M_i, where $i \in [1,n]$ and each communication round r. Our idea to compute LC from these sets is to simply take their *intersection*. The intuition behind intersection is that it represents the conjunction of all partial views of all monitors. That is, in Line 8 of Algorithm 2, we have:

$$LC(\Pi_i^r) = \bigcap_{i \in [1,n]} LS_i^r. \quad (2)$$

C. Motivating Example

The above general ideas for computing μ and LC has one problem. In Line 10, one final LTL$_3$ monitor state should determine the final output, but in some cases, the partial views of two monitors are too coarse and applying intersection on them cannot compute the LTL$_3$ monitor states that represent the aggregate knowledge of the monitors. For example, consider again the LTL$_3$ monitor for formula $\Diamond(a \wedge b)$ in Fig. 2. Suppose that we have a global event $s = \{a, b\}$, two monitors M_1 and M_2, both at initial state q_0, and two partial views,

where M_1 knows the value of a and M_2 knows the value of b. That is,

$$\mathcal{S}_1^s(a) = \mathsf{true} \qquad \mathcal{S}_1^s(b) = \natural$$
$$\mathcal{S}_2^s(a) = \natural \qquad \mathcal{S}_2^s(b) = \mathsf{true}$$

These monitors will compute μ as follows:

$$\mu(\mathcal{S}_1^s, q_0) = \mu(\mathcal{S}_2^s, q_0) = \{q_0, q_\top\}.$$

Applying intersection on $\mu(\mathcal{S}_1^s, q_0)$ and $\mu(\mathcal{S}_2^s, q_0)$ will result in the same set $\{q_0, q_\top\}$. At this point, no matter how many times the monitor processes communicate, at the end of the inner for-loop, LS will not become a singleton and in Line 11 $q_{current}$ cannot be determined properly. This scenario is in particular, problematic since the collective knowledge of M_1 and M_2 (i.e., the fact that a and b are both true) should result in reconstructing $s = \{a, b\}$. Surprisingly, this problem does not stem from the way we compute μ and LC. It is mainly due to the structure of the LTL$_3$ monitor as defined in Definition 1. Although the definition works for centralized monitoring, it needs to be refined for distributed monitors that have only a partial view of the underlying system. In Section V, we will present a technique to transform an LTL$_3$ monitor into one capable of encoding enough information for monitor processes with partial views.

V. Monitor Transformation Algorithm

The discussion in Section IV reveals the source of the problem on the structure of the monitor in Fig. 2. The self-loop on state q_0 prescribes that state q_0 is reachable by three events: $\{a\}$, $\{b\}$, or $\{\}$, while a partial view of $\{a, b\}$ may intersect with both $\{a\}$ and $\{b\}$, which are indistinguishable from each other. If we can somehow split q_0 to two states to explicitly distinguish the cases where either of a or b are true, then applying intersection will effectively solve the problem presented in Section IV-C. More specifically, consider the LTL$_3$ monitor shown in Fig. 3 for formula $\varphi = \Diamond(a \wedge b)$, where state q_0 is split in two states q_{01} and q_{02}. State q_{02} is reached when a is true and b is false. Analogously, State q_{01} is reached when b is true or both a and b are false. Now, recall the two monitors M_1 and M_2 and their partial views in Section IV-C:

$$\mathcal{S}_1^s(a) = \mathsf{true} \qquad \mathcal{S}_1^s(b) = \natural$$
$$\mathcal{S}_2^s(a) = \natural \qquad \mathcal{S}_2^s(b) = \mathsf{true}$$

These monitors will compute μ as follows:

$$\mu(\mathcal{S}_1^s, q_0) = \{q_{02}, q_\top\}$$
$$\mu(\mathcal{S}_2^s, q_0) = \{q_{01}, q_\top\}$$

Applying intersection on $\mu(\mathcal{S}_1^s, q_0)$ and $\mu(\mathcal{S}_2^s, q_0)$ will now result in the singleton $\{q_\top\}$, which is indeed the correct verdict for global event $\{a, b\}$. We call the monitor shown in Fig. 3 an *extended* LTL$_3$ monitor.

In this section, we present an algorithm that takes as input an LTL$_3$ monitor and generates as output an extended LTL$_3$ monitor. We prove that by plugging an extended LTL$_3$ monitor in the distributed RV Algorithm 2, it will produce a verdict identical to that of a centralized LTL$_3$ monitor.

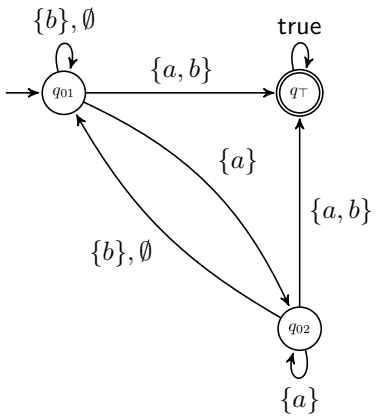

Fig. 3. Extended LTL$_3$ monitor for $\varphi = \Diamond(a \wedge b)$.

A. The Challenge of Generating an Extended LTL$_3$ Monitor

Let $\mathcal{M}_3^\varphi = \langle \Sigma, Q, q_0, \delta, \lambda \rangle$ be the LTL$_3$ monitor of an LTL formula φ. To simplify our notation, we denote transitions of δ by

$$q \xrightarrow{\mathcal{L}(q,q')} q',$$

where the set $\mathcal{L}(q, q')$ of *labels* is formally defined as follows:

$$\mathcal{L}(q, q') = \Big\{ s \in \Sigma \mid \delta(q, s) = q' \Big\}.$$

When it is clear from the context, we refer to the set of labels $\mathcal{L}(q, q')$ simply by $\mathcal{L}$.

Now, suppose that $\mathsf{AP} = \{a, b, c\}$, an LTL$_3$ monitor has a transition of the form:

$$q_0 \xrightarrow{\{a\},\{b,c\},\{a,c\}} q_1,$$

the global event is $s = \{a, b, c\}$, and the partial view of each process M_i, where $i \in [1, n]$, has the value of at most one atomic proposition (i.e., the value of other propositions are unknown). It is straightforward to see that for any global event $s \in \Sigma - \{\{a\}, \{b, c\}, \{a, c\}\}$, the monitor state q_1 appears in the symbolic view of every monitor process M_i, i.e., $q_1 \in \mu(\mathcal{S}_i^s, q_0)$, and consequently, it is impossible for LS_i to become a singleton. Note that q_1 is not the correct verdict. Hence, we need to split q_1 into two new states q_{11} and q_{12}, which can be done in one of the following ways:

(1) $q_0 \xrightarrow{\{a\},\{b,c\}} q_{11}$ and $q_0 \xrightarrow{\{a,c\}} q_{12}$
(2) $q_0 \xrightarrow{\{a\}} q_{11}$ and $q_0 \xrightarrow{\{b,c\},\{a,c\}} q_{12}$
(3) $q_0 \xrightarrow{\{a\},\{a,c\}} q_{11}$ and $q_0 \xrightarrow{\{bc\}} q_{12}$

In scenarios (1) and (2) above: we further need to split q_{11} and q_{12}, respectively. But in scenario (3), there is no need to split q_{11} or q_{12}. Thus, the choice of splitting the monitors' blind spot, has an impact on the size of the extended LTL$_3$ monitor. In order to minimize the number of new states that

```
1  Function SPLIT(ℒ):
2      CV ← 1;
3      for each p ∈ AP do
4          if (∃s, s' ∈ ℒ.p ∈ s ∧ p ∉ s') then
5              CV ← CV + 1;
6          end
7      end
8      if (2^CV > |ℒ|) then
9          return true;
10     end
11     return false;
```

Algorithm 3: Function to determine whether a transition has to split.

are added to the extended LTL$_3$ monitor, we need to compute the minimum-size split. Finding the minimum-size split is a combinatorial optimization problem very similar to the set cover or the hitting set problem [19]. In the next subsection, we present an SMT-based technique to obtain the minimum-size transition split.

B. Identifying the Minimum-size Split

Definition 5: We say that a transition $q \xrightarrow{\mathcal{L}} q'$ *covers* an event $s \in \Sigma$ if and only if

$$\forall p \in \mathsf{AP} : \exists s' \in \mathcal{L} : (p \in s \;\Leftrightarrow\; p \in s')$$

■

Observe that if a transition covers an event, it does not mean that the event is in the label set of the transitions. It only means that all of its propositions are covered.

Definition 6: We say that an event s is *opaque* to a transition $q \xrightarrow{\mathcal{L}} q'$, if (1) $s \notin \mathcal{L}$, but (2) $q \xrightarrow{\mathcal{L}} q'$ covers s. ■

For example, event $\{a, b\}$ is opaque to transition $q_0 \xrightarrow{\{a\},\{b\},\emptyset} q_\top$ in the LTL$_3$ monitor in Fig. 2. It is easy to observe that two partial views of an opaque event to a transition may result in identical possible sets of LTL$_3$ monitor states. When one monitor only reads a and another monitor reads only b, then the resulting set of possible states (i.e., $\{q_0, q_\top\}$) are not distinguishable from each other, because both propositions a and b are in event $\{a, b\}$. Indeed, this is the main reason in creating ambiguity in distributed monitor processes with partial views and such transitions need to be split in order to resolve possible ambiguities. Function SPLIT (see Algorithm 3) determines whether or not a transition should be split. The variable CV in the function computes the number of events covered by the input transition label set. In the above example, the value of 2^{CV} for transition $q_0 \xrightarrow{\{a\},\{b\},\emptyset} q_0$ is 4 which is strictly greater than $|\mathcal{L}| = 3$. This means that the transition needs to be split.

Our goal is to minimize the number of splits for a transition, as the number of splits determines the final size of the extended LTL$_3$ monitor. Formally, given an event $s \in \Sigma$ opaque to a transition $q \xrightarrow{\mathcal{L}} q'$, we aim at splitting the transition to transitions $q \xrightarrow{\mathcal{L}_1} q_1$ to $q \xrightarrow{\mathcal{L}_n} q_n$ such that (1) $\bigcup_{i\in[1,n]} \mathcal{L}_i = \mathcal{L}$, (2) s is opaque to none of these transitions, and (3) n is minimum. It is straightforward to see that this is a combinatorial optimization problem that involves generating all subsets of $\mathcal{L}$ to find the best choice for $\mathcal{L}_1$ to $\mathcal{L}_n$, i.e., a bad choice can result in more future splits. To solve this problem, we transform it into an SMT instance to utilize powerful SMT-solvers.

We now define the constants, variables, constraints, and the optimization objective of our SMT instance. The input is a transition $q \xrightarrow{\mathcal{L}} q'$ and the output are two transitions $q \xrightarrow{\mathcal{L}_1} q_1$ and $q \xrightarrow{\mathcal{L}_2} q_2$ such that minimum number of global events are opaque to the transition.

Constants. For every atomic proposition $p \in \mathsf{AP}$ and every global event $s \in \mathcal{L}$, we employ a Boolean constant a_s^p defined as follows:

$$a_s^p = \begin{cases} \mathsf{true} & \text{if} \quad p \in s \\ \mathsf{false} & \text{if} \quad p \notin s \end{cases}$$

Variables and functions. For every global event $s \in \mathcal{L}$, we define two Boolean variables $x_s^{\mathcal{L}_1}$ and $x_s^{\mathcal{L}_2}$, meaning that $x_s^{\mathcal{L}_1} = \mathsf{true}$, if $s \in \mathcal{L}_1$, otherwise $x_s^{\mathcal{L}_1} = \mathsf{false}$. Likewise, $x_s^{\mathcal{L}_2} = \mathsf{true}$, if $s \in \mathcal{L}_2$, otherwise $x_s^{\mathcal{L}_2} = \mathsf{false}$. We define an operator $\circ$ between a Boolean variable x and a constant a as follows:

$$x \circ a = \begin{cases} a & \text{if} \quad x = \mathsf{true} \\ \mathsf{true} & \text{if} \quad x = \mathsf{false} \end{cases}$$

For each atomic proposition $p \in \mathsf{AP}$, we introduce two Boolean variables $y_{\mathcal{L}_1}^p$ and $y_{\mathcal{L}_1}^{\neg p}$ with the following meaning:

$$y_{\mathcal{L}_1}^p = \begin{cases} \mathsf{true} & \text{if} \quad \forall s \in \mathcal{L}_1 : p \in s \\ \mathsf{false} & \quad\;\; \text{otherwise} \end{cases}$$

$$y_{t_1}^{\neg p} = \begin{cases} \mathsf{true} & \text{if} \quad \forall s \in \mathcal{L}_1 : p \notin s \\ \mathsf{false} & \quad\;\; \text{otherwise} \end{cases}$$

Analogously, for each atomic proposition $p \in \mathsf{AP}$, we introduce Boolean variables $y_{\mathcal{L}_2}^p$ and $y_{\mathcal{L}_2}^{\neg p}$. We also include two Booleans $v_{\mathcal{L}_1}^p$ and $v_{\mathcal{L}_2}^p$, whose meaning is explained later in the set of SMT constraints. For each event $s \in \mathcal{L}$, we define two binary integer variables $w_{\mathcal{L}_1}^p$ and $w_{\mathcal{L}_2}^p$ (for the purpose of counting and optimization) as follows:

$$w_{\mathcal{L}_1}^p = \begin{cases} 0 & \text{if} \quad v_{\mathcal{L}_1}^p = \mathsf{true} \\ 1 & \quad\;\; \text{otherwise} \end{cases}$$

$$w_{\mathcal{L}_2}^p = \begin{cases} 0 & \text{if} \quad v_{\mathcal{L}_2}^p = \mathsf{true} \\ 1 & \quad\;\; \text{otherwise} \end{cases}$$

Constraints. Informally, an event appears either in $\mathcal{L}_1$ or in $\mathcal{L}_2$. Hence, we add the following constraint for each $s \in \mathcal{L}$:

$$x_s^{\mathcal{L}_2} = \neg x_s^{\mathcal{L}_1}$$

The constraints to encode the meaning of variables $y^p_{\mathcal{L}_1}$ and $y^{\neg p}_{\mathcal{L}_1}$ are as follows:

$$y^p_{\mathcal{L}_1} = \bigwedge_{s \in \mathcal{L}} (x^{\mathcal{L}_1}_s \circ a^p_s)$$
$$y^{\neg p}_{\mathcal{L}_1} = \bigwedge_{s \in \mathcal{L}} (x^{\mathcal{L}_1}_s \circ a^{\neg p}_s)$$

It is easy to verify that $y^p_{\mathcal{L}_1}$ evaluates to true if and only if for every event $s \in \mathcal{L}_1$, we have $p \in s$, and $y^{\neg p}_{\mathcal{L}_1}$ evaluates to true if and only if for every event $s \in \mathcal{L}_1$, we have $p \notin s$. Likewise, for variables $y^p_{\mathcal{L}_2}$ and $y^{\neg p}_{\mathcal{L}_2}$, we add the following constraints:

$$y^p_{\mathcal{L}_2} = \bigwedge_{s \in \mathcal{L}} (x^{\mathcal{L}_2}_s \circ a^p_s)$$
$$y^{\neg p}_{\mathcal{L}_2} = \bigwedge_{s \in \mathcal{L}} (x^{\mathcal{L}_2}_s \circ a^{\neg p}_s)$$

Finally, we need to count the number of opaque events in $y^p_{\mathcal{L}_1}$ and $y^{\neg p}_{\mathcal{L}_1}$ (respectively, $y^p_{\mathcal{L}_2}$ and $y^{\neg p}_{\mathcal{L}_2}$). Hence, we add the following assertions:

$$v^p_{\mathcal{L}1} = y^p_{\mathcal{L}_1} \vee y^{\neg p}_{\mathcal{L}_1}$$
$$v^p_{\mathcal{L}2} = y^p_{\mathcal{L}_2} \vee y^{\neg p}_{\mathcal{L}_2}$$

Optimization objective. Our objective is to minimize the total number of opaque events to transition labels $\mathcal{L}_1$ and $\mathcal{L}_2$:

$$\min \sum_{p \in \mathsf{AP}} \left(w^p_{\mathcal{L}_1} + w^p_{\mathcal{L}_2} \right)$$

We remark that although SMT-solvers cannot directly handle optimization objectives such as the above, a common practice is to find the minimum of the above sum using a simple binary search over a coarse range.

C. The Complete Transformation Algorithm

We now know how to split a transition to two transitions with minimum number of opaque events. All we need to do at this point is to design an algorithm that takes as input an LTL$_3$ monitor $\mathcal{M}^{\varphi}_3 = \langle \Sigma, Q, q_0, \delta, \lambda \rangle$ and transforms it into an extended monitor $\mathcal{M}^{\varphi}_e = \langle \Sigma, Q_e, q_0, \delta_e, \lambda_e \rangle$ as output using the above SMT-based optimization technique. We now describe the details of this transformation in Algorithm 4:

- In Lines 2 – 28, we examine each outgoing transition of each state q of the input LTL$_3$ monitor transitions for splitting.
- If a transition does not need to be split, we simply add the original transition to the extended monitor (Lines 25 and 26).
- For each transition that should be split, we apply the above SMT-based optimization technique described in Section V-B. We first add the new states to the set of states of the extended monitor (Line 7). Then, we distinguish two cases:
  - If the transition that needs to be split, say $q \xrightarrow{\mathcal{L}} q'$ is *not* a self-loop (Lines 10 – 13), then two transitions $q \xrightarrow{\mathcal{L}_1} q_1$ and $q \xrightarrow{\mathcal{L}_2} q_2$ with the labels returned by the SMT-solver are included in the extended monitor (see Fig. 4). We also add all the outgoing transitions from q' to q_1 and q_2 (Line 13).
  - If the transition that needs to be split is a self-loop, say $q \xrightarrow{\mathcal{L}} q$, (Lines 15 – 18), then two transitions $q_1 \xrightarrow{\mathcal{L}_1} q_1$ and $q_1 \xrightarrow{\mathcal{L}_2} q_2$ with the labels returned by the SMT-solver are included in the extended monitor (see Fig. 5). We also add all the outgoing transitions from q to q_1 and q_2 (Line 18) for the events not in the original self-loop.
  - Finally, we include the incoming transitions to each state (Line 25) and remove labels that are have no opacity issues (Line 28).
- We repeat the loop until no transition needs to be split.

The reader can test that running Algorithm 4 on the LTL$_3$ monitor in Fig 2, will result in the extended LTL$_3$ monitor in Fig. 3.

We now show the soundness of Algorithm 2 (as defined in the problem statement in Section III-D) when augmented by an extended LTL$_3$ monitor as constructed by Algorithm 4.

Theorem 1: Let $\alpha \in \Sigma^*$ be finite trace and φ be an LTL formula. The return value of Algorithm 2 augmented with an

Input: $\mathcal{M}^{\varphi}_3 = \langle \Sigma, Q, q_0, \delta, \lambda \rangle$
Output: $\mathcal{M}^{\varphi}_e = \langle \Sigma, Q_e, q_0, \delta_e, \lambda_e \rangle$
1 $Q_e \leftarrow Q$;
2 **for** *every* $q \in Q_e$ **do**
3 $\quad \mathcal{L}_q \leftarrow \left\{ \mathcal{L}(q, q') \mid \exists q' \in Q . q \xrightarrow{\mathcal{L}} q' \right\}$;
4 $\quad$ **for** *every* $\mathcal{L}(q, q') \in \mathcal{L}_q$ **do**
5 $\quad\quad$ **if** *SPLIT* $(\mathcal{L}(q, q'))$ **then**
6 $\quad\quad\quad \{\mathcal{L}(q, q_1), \mathcal{L}(q, q_2)\} \leftarrow \mathsf{SMT}(\mathcal{L}(q, q'))$;
7 $\quad\quad\quad Q_e \leftarrow (Q_e \cup \{q_1, q_2\}) - \{q'\}$;
8 $\quad\quad\quad \mathcal{L}_q \leftarrow \mathcal{L}_q \cup \{\mathcal{L}(q, q_1), \mathcal{L}(q, q_2)\}$;
9 $\quad\quad\quad \lambda_e(q_1), \lambda_e(q_2) \leftarrow \lambda(q')$;
10 $\quad\quad\quad$ **if** $q \neq q'$ **then**
11 $\quad\quad\quad\quad \delta_e(q, s) \leftarrow q_1$ for all $s \in \mathcal{L}(q, q_1)$;
12 $\quad\quad\quad\quad \delta_e(q, s) \leftarrow q_2$ for all $s \in \mathcal{L}(q, q_2)$;
13 $\quad\quad\quad\quad \delta_e(q_1, s), \delta_e(q_2, s) \leftarrow \delta(q', s)$ for all $s \in \Sigma$;
14 $\quad\quad\quad$ **end**
15 $\quad\quad\quad$ **if** $q = q'$ **then**
16 $\quad\quad\quad\quad \delta_e(q_1, s) \leftarrow q_1$ for all $s \in \mathcal{L}(q, q_1)$;
17 $\quad\quad\quad\quad \delta_e(q_1, s) \leftarrow q_2$ for all $s \in \mathcal{L}(q, q_2)$;
18 $\quad\quad\quad\quad \delta_e(q_1, s), \delta_e(q_2, s) \leftarrow \delta(q', s)$ for every $s \in \Sigma - \mathcal{L}(q, q')$;
19 $\quad\quad\quad$ **end**
20 $\quad\quad\quad$ **for** *every* q'' *such that* $\delta(q'', s) = q'$ **do**
21 $\quad\quad\quad\quad \delta_e(q'', s) \leftarrow q_1$;
22 $\quad\quad\quad$ **end**
23 $\quad\quad$ **end**
24 $\quad\quad$ **else**
25 $\quad\quad\quad \delta_e(q, s) \leftarrow q'$ for every $s \in \mathcal{L}(q, q')$;
26 $\quad\quad\quad \lambda_e(q') \leftarrow \lambda(q')$;
27 $\quad\quad$ **end**
28 $\quad\quad \mathcal{L}_q \leftarrow \mathcal{L}_q - \{\mathcal{L}(q, q')\}$;
29 $\quad$ **end**
30 **end**

Algorithm 4: Extended LTL$_3$ Monitor Construction

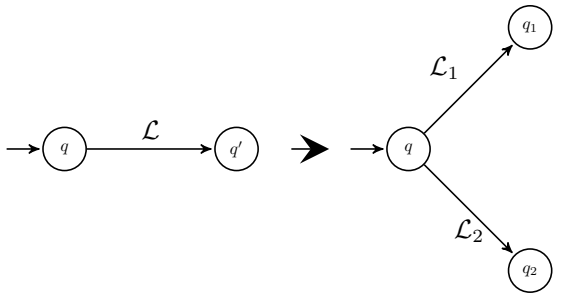

Fig. 4. Splitting a transition to two.

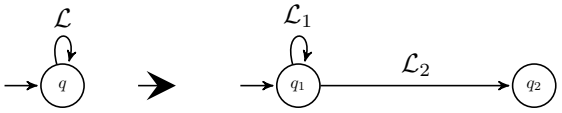

Fig. 5. Splitting a self-loop to two.

extended LTL_3 monitor as constructed in Algorithm 4 is $[\alpha \models_3 \varphi]$ by every monitor process in the presence of up to t crash failures.

Proof: Please refer to the appendix. ∎

Theorem 2: Let φ be an LTL formula and $\alpha \in \Sigma^*$ be a finite trace. The message complexity of Algorithm 2 using an extended LTL_3 monitor is

$$O\left(\log\left(|\mathcal{M}_3^\varphi| \cdot |\mathsf{AP}|\right)|\alpha|(t+1)n^2\right).$$

Proof: Please refer to the appendix. ∎

VI. Related Work

In the sequel, we focus on reviewing the work on monitoring distributed systems and distributed monitoring.

A. Synchronous Distributed Monitoring

The most relevant work to this paper is the algorithms proposed in [6], [7]. The algorithm in [6] for monitoring synchronous distributed systems with respect to LTL formulas is designed such that satisfaction or violation of specifications can be detected by local monitors alone. The framework employs disjoint alphabet for each process in the system. Thus, a local monitor in [6] can only evaluate subformulas that include its own propositions and if the subformula contains propositions of other processes, it sends a proof obligation to the corresponding monitor to resolve the obligation. This technique is called formula *progression*. This implies that if multiple proof obligations exist, the formula needs to be progressed by multiple monitors in a sequence of communication rounds. Each round may increase the size of the formula to remember what happened in the past.

In [7], 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 computation. They formalize choreography-based decentralized monitoring by showing how to synthesize a network from an LTL formula, and give a decentralized monitoring algorithm working on top of an LTL network.

The approach in these articles are different from our work in this paper in the following fundamental ways: (1) the framework in [6], [7] is fault-free, (2) we do not assume that components have disjoint alphabet and can indeed observe a shared set of propositions. This creates ambiguity among monitors and brings inconsistency issues to the problem, which are absent in [6], [7].

B. Fault-tolerant Distributed Monitoring

In [14], [15] the authors show that if runtime monitors employ enough number of *opinions* (instead of the conventional binary valuations), then it is possible to monitor distributed tasks in a consistent manner. Building on the work in [14], [15], [20], the authors in [5] show that employing the four-valued LTL [21] will result in inconsistent distributed monitoring for some formulas. They subsequently introduce a family of logics, called LTL_{2k+4}, that refines the 4-valued LTL by incorporating $2k+4$ truth values, for each $k \geq 0$. 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. In this paper, we showed that in a synchronous setting, we do not need to change the number of truth values.

C. Distributed Monitoring for Past-time LTL

In [8], the authors propose a decentralized monitoring algorithm that monitors a distributed program with respect to safety properties in PT-DTL, a variant of the past-time linear temporal logic. PT-DTL expresses temporal properties of distributed systems by drawing relation to particular processes and their knowledge of the local state of other processes at any point in time. In the monitoring algorithm, monitors gain knowledge about the state of the system by piggybacking on the existing communication among processes. In such a framework, the valuation of some predicates and properties may be overlooked. That is, if processes rarely communicate, then monitors exchange little information and, hence, some violations of properties may remain undetected.

D. Lattice-theoretic Distributed Monitoring

Predicate detection is the problem of identifying states of a distributed computation that satisfy a predicate [9], [10]. The problem is in general NP-complete [22]. *Computation slicing* [23] is a technique for reducing the size of the computation and, hence, the number of global state to be analyzed for detecting a predicate. The slice of a computation with respect to a predicate is the sub-computation satisfying the following two conditions: (1) it contains all global states for which the predicate evaluates to true, and (2) among all computations that satisfy the first condition, it contains

the least number of consistent cuts. In [23], the authors propose an algorithm for detecting *regular* predicates. This idea is then extended to a full blown distributed algorithm for distributed monitoring [3]. One shortcoming of this line work is that it does not address monitoring properties with temporal requirements. This shortcoming is partially addressed in [11] for a fragment of temporal operators. In [4], the authors propose the first sound method for runtime verification of asynchronous distributed programs for the 3-valued semantics of LTL specifications defined over the global state of the program. In the proposed setting, monitors are not subject to faults. The technique for evaluating LTL properties is inspired by distributed computation slicing described above. The monitoring technique is fully decentralized. LTL formulas in this work are in terms of conjunctive predicates.

Lattice-based techniques may suffer from the existence of too many concurrent states. To tackle this problem in [12], the authors propose an algorithm and analytical bounds if a combination of logical and physical clocks (called *hybrid* clocks) are used. This method is enriched with SAT solving techniques in [13].

VII. Conclusion

In this paper, we proposed a runtime verification algorithm, where a set of decentralized synchronous monitors that have only a partial view of the underlying system continually evaluate formulas in the linear temporal logic. We assume that the communication network is a complete graph and each monitor is subject to crash failures. Our algorithm is sound in the sense that upon termination, all local monitors compute the same monitoring verdict as a centralized monitor that can atomically observe the global state of the system. The monitors do not share their full observation of the underlying system. Rather, they communicate a symbolic representation of their partial observations without compromising soundness. This symbolic observation is the set of possible LTL_3 monitor states. Since LTL_3 monitors may not be able to resolve indistinguishable cases due to partial observations, we also proposed an SMT-based transformation algorithm to obtain minimum size LTL_3 monitors. Our SMT-based algorithm increases the size of an LTL_3 monitor only by a factor of $O(\log |\mathsf{AP}|)$ (communicating explicit observations would require $O(|\mathsf{AP}|)$ bits), where AP is the set of atomic propositions that describe the global state of the underlying system.

As for future work, we plan to study the same problem where the communication network graph is not complete (e.g., a tree or a ring). Another interesting research avenue is to consider other types of faults, e.g., when monitors are subject to *Byzantine* faults and may misrepresent their observation of the underlying system.

VIII. Acknowledgment

We would like to thank César Sánchez of IMDEA Software Institute, Spain, for sharing his idea of computing monitor state intersections in private communication. This work was partially supported by Canada NSERC Discovery Grant 418396-2012 and by NSERC Strategic Grants 430575-2012 and 463324-2014.

References

[1] K. Havelund and G. Rosu, "Monitoring Programs Using Rewriting," in *Automated Software Engineering (ASE)*, 2001, pp. 135–143.

[2] E. Bartocci, Y. Falcone, B. Bonakdarpour, C. Colombo, N. Decker, F. Klaedtke, K. Havelund, Y. Joshi, R. Milewicz1, G. Reger, G. Rosu, J. Signoles, D. Thoma, E. Zalinescu, and Y. Zhang, "First international competition on runtime verification," *Software Tools for Technology Transfer (STTT)*, 2018.

[3] H. Chauhan, V. K. Garg, A. Natarajan, and N. Mittal, "A distributed abstraction algorithm for online predicate detection," in *IEEE 32nd Symposium on Reliable Distributed Systems (SRDS)*, 2013, pp. 101–110.

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

[5] B. Bonakdarpour, P. Fraigniaud, S. Rajsbaum, D. A. Rosenblueth, and C. Travers, "Decentralized asynchronous crash-resilient runtime verification," in *Proceedings of the 27th International Conference on Concurrency Theory (CONCUR)*, 2016, pp. 16:1–16:15.

[6] A. Bauer and Y. Falcone, "Decentralised LTL monitoring," *Formal Methods in System Design*, vol. 48, no. 1-2, pp. 46–93, 2016.

[7] C. Colombo and Y. Falcone, "Organising LTL monitors over distributed systems with a global clock," *Formal Methods in System Design*, vol. 49, no. 1-2, pp. 109–158, 2016.

[8] K. Sen, a. Vardhan, G. Agha, and G. Rosu, "Efficient decentralized monitoring of safety in distributed systems," in *International Conference on Software Engineering (ICSE)*, 2004, pp. 418–427.

[9] V. K. Garg, *Elements of distributed computing*. Wiley, 2002.

[10] S. D. Stoller and F. B. Schneider, "Verifying programs that use causally-ordered message-passing," *Sci. Comput. Program.*, vol. 24, no. 2, pp. 105–128, 1995. [Online]. Available: https://doi.org/10.1016/0167-6423(95)00002-A

[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)*, 2007, pp. 420–434.

[12] S. Yingchareonthawornchai, D. N. Nguyen, V. T. Valapil, S. S. Kulkarni, and M. Demirbas, "Precision, recall, and sensitivity of monitoring partially synchronous distributed systems," in *Runtime Verification - 16th International Conference, RV 2016, Madrid, Spain, September 23-30, 2016, Proceedings*, 2016, pp. 420–435.

[13] V. T. Valapil, S. Yingchareonthawornchai, S. S. Kulkarni, E. Torng, and M. Demirbas, "Monitoring partially synchronous distributed systems using SMT solvers," in *Runtime Verification - 17th International Conference, RV 2017, Seattle, WA, USA, September 13-16, 2017, Proceedings*, 2017, pp. 277–293.

[14] P. Fraigniaud, S. Rajsbaum, and C. Travers, "On the number of opinions needed for fault-tolerant run-time monitoring in distributed systems," in *Runtime Verification (RV)*, 2014, pp. 92–107.

[15] P. Fraigniaud, S. Rajsbaum, M. Roy, and C. Travers, "The opinion number of set-agreement," in *Principles of Distributed Systems - 18th International Conference (OPODIS)*, 2014, pp. 155–170.

[16] Z. Manna and A. Pnueli, "The modal logic of programs," in *Proceedings of the 6th Colloquium on Automata, Languages and Programming (ICALP)*, 1979, pp. 385–409.

[17] A. Bauer, M. Leucker, and C. Schallhart, "Runtime Verification for LTL and TLTL," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 20, no. 4, pp. 14:1–14:64, 2011.

[18] N. Lynch, *Distributed Algorithms*. San Mateo, CA: Morgan Kaufmann Publishers, 1996.

[19] M. Garey and D. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York: W. H. Freeman, 1979.

[20] P. Fraigniaud, S. Rajsbaum, and C. Travers, "Locality and checkability in wait-free computing," *Distributed Computing*, vol. 26, no. 4, pp. 223–242, 2013.

[21] A. Bauer, M. Leucker, and C. Schallhart, "Comparing LTL Semantics for Runtime Verification," *Journal of Logic and Computation*, vol. 20, no. 3, pp. 651–674, 2010.

[22] N. Mittal and V. K. Garg, "On detecting global predicates in distributed computations," in *Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS 2001), Phoenix, Arizona, USA, April 16-19, 2001*, 2001, pp. 3–10.

[23] ——, "Techniques and applications of computation slicing," *Distributed Computing*, vol. 17, no. 3, pp. 251–277, 2005.

APPENDIX

A. Proof of Theorem 1

We prove Theorem 1 in three steps, similar to the proof technique for consensus in synchronous networks (e.g., the *FloodSet* algorithm) [18]. First, we prove that at then end of inner for-loop LS includes only one state. Then, we show that if no crash faults occur, in one round, all monitors will compute a monitor state q, where $\lambda(q)$ is the same as what a centralized monitor that can read the global event in one atomic step would compute. Finally, we show that if up to t monitors crash, all active monitors return $\lambda(q)$ as described in the previous step. We now delve into these three steps:

- **Step 1.** Let us assume that the monitor processes in $\mathcal{M}$ are evaluating event s_j for some $j \in [0, k]$. Formally, we are going to show that if no crash faults occur, then in Line 10 of Algorithm 2, we have $|LS_i^1| = 1$, for all $i \in [1, n]$. First, note that if no faults occur, all monitors send and receive all the messages in one clean round. Thus, in the subsequent rounds all messages will be identical. We now prove this claim by contradiction. Suppose we have $|LS_i^1| = 2$ (the case for > 2 can be trivially generalized). This means that at least two monitor processes sent a message containing two possible LTL_3 monitor states, say $\{q_1, q_2\}$. This can be due to two scenarios:
  - The first scenario is that q_1 and q_2 are possible LTL_3 monitor states, because the value of some atomic proposition $p \in \mathsf{AP}$ is unknown, i.e., $\mathcal{S}(p) = \natural$. However, this scenario contradicts our assumption on *event coverage* (see Section III) in our computation model.
  - The second scenario is that q_1 and q_2 are possible LTL_3 monitor state, because s_j is opaque to some outgoing transitions from $q_{current}$ in the LTL_3 monitor. This case contradicts with our construction of extended LTL_3 monitor in Algorithm 4.
- **Step 2.** We prove this step by induction on the length of the finite input trace. The base case is that the monitors are evaluating event s_0 and $q_{current} = q_0$. From Step 1 of the proof, we know that $|LS_i^1| = 1$. We also know that $|LS_i^r| = 1$ (for all $r \in [1, t+1]$) and LS_i^r contains the same content as LS_i^1. Let this content be an LTL_3 monitor state q. Our goal is to show that:

  $$\lambda(q) = [s_0 \models \varphi].$$

  The proof, again, is by contradiction. This scenario can happen if the intersection of all possible monitor states q, where $q = \delta(q_0, s_0)$ and $\lambda(q) \neq [s_0 \models \varphi]$. This can happen only if due to opacity, a wrong monitor state comes out of the intersection. This case contradicts with out construction of extended LTL_3 monitor in Algorithm 4. Hence, q would be the monitor state that a centralizes monitor would compute. The induction step is now trivial: it is straightforward to show that for any valid $q_{current}$ and any s_j, the next monitor state is the same as what a centralized monitor would compute.
- **Step 3.** From Steps 1 and 2, we know that if no faults occur, in one round all monitors compute one and only one LTL_3 monitor state q, where $\lambda(q) = [\alpha \models \varphi]$. Now, we show in a fault-prone scenario, in some round $1 \leq r \leq t+1$, any two active monitors M_i and M_j compute the same single monitor state $LS_i^r = \{q\}$, where $\lambda(q) = [\alpha \models \varphi]$. Since there are at most t crash failures, there has to be some round r, where no failures occur. Recall that in Section III, we assume that if a monitor crashes and this monitor is the only one that is aware of some proposition $p \in \mathsf{AP}$, this monitor sends a message containing its set of possible monitor states before crashing. This assumption ensures event coverage. This means that in any round $r \leq r' \leq t+1$, the value of all propositions are read. This in turn implies that all rounds r' are now identical to a fault-free setting and, hence, Steps 1 and 2 hold.

These three steps prove the soundness of Algorithm 2 when augmented by an extended LTL_3 monitor as constructed by Algorithm 4.

■

B. Proof of Theorem 2

We analyze the complexity of each part of Algorithm 2:

- The algorithm has a nested loop. The outer loop iterates exactly $|\alpha|$ times.
- The inner loop iterates exactly $t+1$ times.
- In the inner loop each monitor process sends n messages to all other monitrs and receives n messages from all other monitors. That is, n^2 messages.

This makes it a total of $|\alpha|(t+1)n^2$ messages throughout the algorithm.

We now focus on the size of each message. Let $\mathcal{M}_3^\varphi = \langle \Sigma, Q, q_0, \delta, \lambda \rangle$ be an LTL_3 monitor and $\mathcal{M}_e^\varphi = \langle \Sigma, Q_e, q_0, \delta_e, \lambda_e \rangle$ be its extended monitor constructed by Algorithm 4. The algorithm may split a transition at most $|\mathsf{AP}|$ number of times. Hence, we have

$$|Q_e| \leq 2|Q| \cdot |\mathsf{AP}|.$$

Recall that each message contains the possible states of the extended LTL_3 monitor. This means each message in Algorithm 2 needs

$$O\left(\log\left(|Q| \cdot |\mathsf{AP}|\right)\right)$$

bits for each message. Recall that the size of an LTL_3 monitor is the number of its state, i.e., $|\mathcal{M}_3^\varphi| = |Q|$. Hence, the message complexity is

$$O\left(\log\left(|\mathcal{M}_3^\varphi| \cdot |\mathsf{AP}|\right)|\alpha|(t+1)n^2\right).$$

■