Crash-Resilient Decentralized Synchronous Runtime Verification

Ritam Ganguly, Shokufeh Kazemloo, and Borzoo Bonakdarpour

Abstract—Runtime verification is a technique, where a monitor process extracts information from a running system in order to evaluate whether system executions violate or satisfy a given correctness specification. In this paper, we consider runtime verification of synchronous distributed systems, where a set of decentralized 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. In our proposed approach, 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. We evaluate our algorithm on a wide range of formulas and observe an average of 2.5 times increase in the number of states of the monitor automaton.

Index Terms—Runtime Verification, Fault-tolerant Monitoring, Distributed Systems

1 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 written 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], [4], [5], [6], [7], [8], [9], [10], [11], [12], [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) we have every reason to believe that distributed monitors like any other process are not perfect and may be subject to faults. While the first difficulty has been studied in various forms [3], [4], [9], [10], [11], [12], [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. We consider that the clock synchronization algorithm keeps the clock skew between any two processes to the minimum such that we can consider the different processes to be synchronous, while we attempt to solve the later two challenges.

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 features:

- 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.

This work is sponsored by the United States National Science Foundation CCF (FMIf) Award 2102106 and CCF (SHF) Award 2118356.
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 reads only an arbitrary proper subset of AP.

The formal specification language is the popular linear temporal logic (LTL) [16], where formulas are inductively constructed using the propositions in AP and operators that describe the temporal order of events.

Our goal is to design a distributed monitoring algorithm with the following properties:

- **Soundness.** Upon termination, all local monitors compute the same monitoring verdict as a central-ized 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 \(O(|AP|)\), 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)\(^1\). 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 \(O(\log(|M^\pi_{AP}|) \cdot |AP|)\), where \(M^\pi_{AP}\) denotes the finite state automaton for monitoring an LTL formula \(\phi\) in the 3-valued semantics\(^2\) as constructed in [17]. In short, our RV framework has message complexity

\[
O\left( \log \left( |M^\pi_{AP}| \right) |AP| n^2 (t + 1) \right)
\]

for evaluating each global state, where \(n\) is the number of distributed monitors and \(t\) is the bound on the number of crash failures. An important implication of our results is that unlike the asynchronous fault-prone setting, 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 for sound distributed monitoring.

To enhance the efficiency further, we limit the number of rounds to the maximum number of crashes that is possible in the system at any given state and not be constant at \(t\). Thus reducing the average number of rounds. Also, to limit the total number of messages sent between monitors we let the communication happen after every \(l\) states. The partial observation of all previous \(l\) states are preserved for communication. This considerably decreases the number of messages being sent for inter-monitor communication, at the cost of increase in the average size of the message because of the higher number of possible states that the monitor automaton can be in.

We have implemented and evaluated our approach on a variety of LTL formulas for traces being generated using different random distributions as well as an IoT dataset, Orange4Home [18]. We analyze the average number of rounds and total messages being sent in the system for different values of \(t\) and \(l\). We also analyze the change in the average number of rounds, total number of messages, average size of a message along with total monitor crashing in the system for different length of execution traces.

**Comparison to the conference version:** This paper is an extended version of the short paper appeared in the 37th IEEE International Symposium on Reliable Distributed Systems (SRDS’18). Section 5 on the minimization and monitor transformation techniques as well as the the experimental results in Section 6 are entirely new material compared to the conference version.

**Organization:** The rest of the paper is organized as follows. We introduce the preliminary concepts in Section 2. Section 3 presents our model of computation for decentralized crash-resilient synchronous RV. We present the general idea behind our RV algorithm in Section 4 and subsequently elaborate on the details in Section 5. Experimental Results for various formulas are analyzed in Section 6. Related work is discussed in Section 7. Finally, we make concluding remarks and discuss future work in Section 8.

### 2 Preliminaries

In this section, we review the preliminary concepts.

---

1. 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.

2. LTL\(_3\) introduces a three-valued semantics with \(\top, \bot\) and \(?\) as the interpretation of a finite trace, where the interpretations correspond to satisfaction, violation and inconclusive observations respectively.
2.1 Linear Temporal Logic

Let $AP$ be a set of atomic propositions and $\Sigma = 2^{AP}$ be the alphabet. We call each element of $\Sigma$ an event. For example, for $AP = \{a, b\}$, 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 $s_0s_1s_2 \cdots$, where $s_i \in \Sigma$, for every $i \geq 0$. The set of all finite (respectively, infinite) traces over $\Sigma$ is denoted by $\Sigma^*$ (respectively, $\Sigma^\omega$). Throughout the paper, we denote finite traces by the letter $\alpha$, and infinite traces by the letter $\sigma$.

For a finite trace $\alpha = s_0s_1 \cdots s_n$, by $\alpha^i$, we mean trace suffix $s_is_{i+1} \cdots s_n$ of $\alpha$.

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

$$\varphi ::= p \mid \neg \varphi \mid \varphi \lor \varphi \mid \Box \varphi \mid \varphi \land \varphi$$

where $p \in AP$ is an atomic proposition, $\land$ is the “until” operator, and $\Box$ is the “next” operator. Additionally, we allow the following operators as syntactic sugar, each of which is defined in terms of the above ones:

- $\true = p \lor \neg p$, $\false = \neg \true$
- $\varphi_1 \lor \varphi_2$ is denoted by $\varphi_1 \lor \neg \varphi_2$
- $\varphi \land \varphi_1 \lor \varphi_2$ is denoted by $\varphi \land \neg (\varphi_1 \lor \varphi_2)$
- $\Box \varphi$ is true $\varphi$, and $\Diamond \varphi$ is $\neg \Box (\neg \varphi)$, 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 $\sigma = s_0s_1 \cdots$ be an infinite trace in $\Sigma^\omega$, $i \geq 0$ be a non-negative integer, and $\models$ denote the satisfaction relation. The semantics of LTL is defined as follows:

$$\sigma, i \models p \iff p \in s_i$$

$$\sigma, i \models \neg \varphi \iff \sigma, i \not\models \varphi$$

$$\sigma, i \models \varphi_1 \lor \varphi_2 \iff \sigma, i \models \varphi_1 \lor \sigma, i \not\models \varphi_2$$

$$\sigma, i \models \Box \varphi \iff \sigma, i + 1 \models \varphi$$

$$\sigma, i \models \Box \varphi_1 \land \varphi_2 \iff \exists k \geq i : \sigma, k \models \varphi_2 \land \forall j \in [i, k) : \sigma, j \models \varphi_1.$$ 

Also, $\sigma \models \varphi$ holds iff $\sigma, 0 \models \varphi$ holds. For example, consider the following request/acknowledgment LTL formula $\varphi_{ra} = \Box (\neg a \land \neg r) \lor ((\neg a \land r) \land \Diamond a)$ 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.

2.2 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 LTL3) [17]) allows us to reason about finite traces for verifying properties at run time with an eye on possible future extensions. The syntax of LTL3 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 $\varphi$ with respect to $\alpha$ in the 3-valued semantics, denoted by $[\alpha \models_3 \varphi]$, is defined as follows:

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

For example, consider formula $\varphi = a \land b$ and the following three finite traces:

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

Here, we have $[u_1 \models_3 \varphi] = \top$, as this finite trace can be extended to traces that result in violation or satisfaction of $\varphi$. 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 LTL3 monitor of a formula $\varphi$ is the unique deterministic finite state machine $M_{\varphi} = (\Sigma, Q, q_0, \delta, \lambda)$, where $Q$ is a set of states, $q_0 \in Q$ is the initial state, $\delta : Q \times \Sigma \rightarrow Q$ is the transition function, and $\lambda : Q \rightarrow \mathbb{B}_3$, is a function such that: $\lambda(\delta(q_0, \alpha)) = [\alpha \models_3 \varphi]$, for every finite trace $\alpha \in \Sigma^*$. \hfill \square

In [17], the authors introduce an algorithm that takes as input an LTL formula and constructs as output an LTL3 monitor. For example, Fig. 1 shows the LTL3 monitor for formula $\varphi = a \land b$, where $\lambda(q_0) = \top$, $\lambda(q_L) = \bot$, and $\lambda(q_T) = \top$. It is easy to observe that finite traces $u_1$, $u_2$, and $u_3$ terminate at monitor states $q_0$, $q_L$, and $q_T$, respectively.

3 Model of Computation

An LTL3 monitor as defined in Definition 1 can evaluate an LTL formula $\varphi$ 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.

3.1 Overall Picture

We consider a distributed monitoring system comprising of a fixed number $n$ of monitor processes $M = \{M_1, M_2, \ldots, M_n\}$ that communicate with each other by sending and receiving messages through point-to-point
bidirectional communication links. 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 [19]. 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 \( \alpha = s_0s_1 \cdots s_k \), and is inspected with respect to an LTL formula \( \varphi \) 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 \( i \in [1, n] \) (we will generalize this event-by-event approach in Section 5):

1. reads the value of propositions in \( s_j \) (visible to \( M_i \)), 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 \( \varphi \) and emits a truth value from \( \mathbb{B}_3 \), where \( t \) is the upper bound on the number of monitor process crash failures.

3.2 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 \( \alpha = s_0s_1 \cdots s_k \), 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 \( S_i^j \) defined next.

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

Notice that the unknown value ‘\( \sharp \)’ for a proposition is different from the unknown truth value ‘?’ in the LTL\(_3\) semantics.

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

\[
(S(p) = \text{true} \iff p \in s) \land (S(p) = \text{false} \iff p \notin s).
\]

Hence, a partial view \( S \) 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 \( S_i^s \) and assume that \( S_i^s \subseteq s \). This implies that two monitors \( M_i \) and \( M_j \) cannot have inconsistent partial views of the same global event. That is, for any event \( s \) and partial views \( S_i^s, S_j^s \), and for every \( p \in \text{AP} \), we have:

\[
(S_i^s(p) \neq S_j^s(p)) \implies (S_i^s(p) = \sharp \lor S_j^s(p) = \sharp).
\]

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 \(|\text{AP}|\) bits. Our goal is to design a technique, where monitor processes can communicate their observations without sending and receiving their partial views of 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 \( S_i^s \) with respect to an LTL formula \( \varphi \) by \( LS_i^s = \mu(S_i^s, \varphi) \) (see Line 3 in Algorithm 1). In Section 4, we will present a concrete way of computing \( \mu \).

Let \( LS_i^s \) 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 \( \Pi_i^r = \{LS_j^s\}_{j \in [1, n]} \) be the set of all messages received 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 4). 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.

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.
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 \( p \in AP \), there exists \( M_i \in \mathcal{M} \) such that \( S_3^i(p) \neq \top \).

### 3.3 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 \( p \in AP \), such that at round \( r \) monitor process \( M_i \) is the only monitor aware of \( p \), the message sent to \( M_i \) 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\(^5\)). To account for the maximum number of \( t \) monitor crashes that can take place, we have \( t + 1 \) communication rounds. This guarantees the symbolic view of all monitors reaches all active monitors at the end of each iteration of the outer loop.

### 3.4 Problem Statement

Our formal problem statement is the termination requirement for Algorithm 1. We require that when a non-faulty monitor process runs Algorithm 1 to the end, it emits a verdict that a centralized monitor that has global view of the system would compute:

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

where \( \alpha \in \Sigma^* \), \( \varphi \) is an LTL formula, and \( \nu_i \) is the truth value emitted by monitor \( M_i \) at the end of 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 [19]. The main difference is that in consensus, processes need to agree on ones values that they own. In our problem, they should agree on the value \( [\alpha \models_3 \varphi] \), while none of the monitors necessarily has this value before the inner for-loop. In Section 5, 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.

## 4 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 functions \( \mu \) and \( LC \). Our general idea is described in the sequel and is reflected in Algorithm 2, which refines Algorithm 1.

5. Through Triple Modular Redundancy (TMR) we make sure that in a case with three systems, even if any one of the systems fail, the other two will make sure that the overall output is not effected.

![Fig. 2: LTL3 monitor for \( \varphi = \Diamond(a \land b) \).](image)

### 4.1 Symbolic View \( \mu \)

As mentioned in Section 3, sharing explicit partial views is not space efficient, as each message will need at least \( |AP| \) 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 \( S \) be the partial view of the monitor process. The set of possible next LTL\(_3\) monitor states can be computed as follows:

\[
\mu(S, q) = \left\{ q' \mid \exists s \in \Sigma . (S \subseteq s \land \delta(s, q, s) = q') \right\}
\]

Recall that \( \delta \) denotes the transition function in LTL\(_3\) monitors. For example, consider the following LTL\(_3\) formula \( \varphi = \Box(a \land b) \). The LTL\(_3\) monitor of this formula is shown in Fig. 2, where \( \lambda(q_0) = \bot \) and \( \lambda(q_T) = \top \). 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 \( S_1^i(a) = \top \) and \( S_1^i(b) = \top \). This implies that monitor \( M_1 \) considers \( q_T \) as the only possible next LTL\(_3\) monitor state, i.e., \( \mu(S_1^i, q_0) = \{ q_T \} \). However, considering another partial view \( S_1^i(a) = \top \) and \( S_1^i(b) = \bot \), monitor process \( M_1 \) will have to consider \( \{ q_0, q_T \} \) as possible next LTL\(_3\) monitor states. This is because it has to consider two possibilities for proposition \( b \). That is, \( \mu(S_1^i, q_0) = \{ q_0, q_T \} \).

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

### 4.2 Computing \( LC \)

Given a set of possible LTL\(_3\) monitor states computed by \( \mu \), in Line 7 of Algorithm 2, each monitor process receives a set of possible states from all other monitors, denoted by \( LS_i^\tau \) for each monitor process \( M_i \), where \( i \in [1, n] \) and each communication round \( \tau \). 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:

\[
LC(\Pi_i^\tau) = \bigcap_{i \in [1, n]} LS_i^\tau.
\]

### 4.3 Motivating Example

The above general ideas for computing \( \mu \) 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 \( \Diamond(a \land b) \) 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,

\[ S^*_1(a) = \text{true} \quad S^*_1(b) = \top \]
\[ S^*_2(a) = \top \quad S^*_2(b) = \text{true} \]

These monitors will compute \( \mu \) as follows:

\[ \mu(S^*_1, q_0) = \mu(S^*_2, q_0) = \{q_0, q_\top\} \]

Applying intersection on \( \mu(S^*_1, q_0) \) and \( \mu(S^*_2, q_0) \) 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, \( q_{\text{current}} \) 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 \( \mu \) and \( LC \). It is mainly due to the structure of the LT3 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 5, we present a technique to transform an LT3 monitor into an equivalent one capable of encoding enough information for monitor processes with partial views.

5 Monitor Transformation Algorithm

The discussion in Section 4 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 4.3. More specifically, consider the LT3 monitor shown in Fig. 3 for formula \( \varphi = \Box(a \land b) \), 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 4.3:

\[ S^*_1(a) = \text{true} \quad S^*_1(b) = \top \]
\[ S^*_2(a) = \top \quad S^*_2(b) = \text{true} \]

These monitors will compute \( \mu \) as follows:

\[ \mu(S^*_1, q_0) = \{q_{02}, q_\top\} \]
\[ \mu(S^*_2, q_0) = \{q_{01}, q_\top\} \]

Applying intersection on \( \mu(S^*_1, q_0) \) and \( \mu(S^*_2, q_0) \) 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 LT3 monitor.

In this section, we present an algorithm that takes as input an LT3 monitor and generates as output an extended LT3 monitor. We prove that by plugging an extended LT3 monitor in the distributed RV Algorithm 2, it will produce a verdict identical to that of a centralized LT3 monitor.

5.1 The Challenge of Constructing Extended Monitors

Let \( M^\varphi_i = (\Sigma, Q, q_0, \delta, \lambda) \) be the LT3 monitor of an LT3 formula \( \varphi \). To simplify our notation, we denote transitions of \( \delta \) by:

\[ q \xrightarrow{\varphi} q' \]

where the set \( L(q, q') \) of labels is formally defined as follows:

\[ L(q, q') = \{s \in \Sigma \mid \delta(q, s) = q'\} \]

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

Now, suppose that AP = \{a, b, c\}, an LT3 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 = \{a, b, c\}, the monitor state q_1 appears in the symbolic view of every monitor process M_i, i.e., q_1 \in \mu(S^*_i, 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 \( q_0 \xrightarrow{\{b,c\}} q_{12} \)

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 \( \text{LTL}_3 \) monitor. In order to minimize the number of new states that are added to the extended \( \text{LTL}_3 \) 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 problems [20]. In the next subsection, we present an SMT-based technique to obtain the minimum-size transition split.

### 5.2 Identifying the Minimum-size Split

**Definition 5.** We say that a transition \( q \xrightarrow{e} q’ \) covers an event \( s \in \Sigma \) if and only if

\[
\forall p \in \text{AP} : \exists s' \in L : (p \in s \iff p \in s').
\]

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 \( q \xrightarrow{e} q’ \), if (1) \( s \notin \text{L} \), and (2) \( q \xrightarrow{e} q’ \) covers \( s \).

For example, event \( \{a, b\} \) is opaque to transition \( q_0 \xrightarrow{\{a\},\{b\}} q_0 \) in the \( \text{LTL}_3 \) 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 \( \text{LTL}_3 \) monitor states. When one monitor only reads \( a \) and another monitor reads only \( b \), then the resulting set of possible states (i.e., \( \{q_0, q_I\} \)) 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 \( \text{SPLIT} \) (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 \( CV \) for transition \( q_0 \xrightarrow{\{a\},\{b\}} q_0 \) is 4. The intuition behind \( CV > |L| \) is that the transition \( L \) should not correspond to more events than the number of labels in the transition. \( CV \) counts the number of atomic proposition that is true in one label and false in the other. Thus, \( CV \) gives us the total number of events that it corresponds to. If it is more than the number of labels in the transition, then it induces ambiguity about the automata state of the monitor, therefore suggesting the need of splitting the transition.

Our goal is to minimize the number of splits for a transition, as the number of splits determines the final size of the extended \( \text{LTL}_3 \) monitor. Formally, given an event \( s \in \Sigma \) opaque to a transition \( q \xrightarrow{e} q’ \), we aim at splitting the transition to transitions \( q \xrightarrow{L_1} q_1 \) to \( q \xrightarrow{L_n} q_n \) such that \( 1 \bigcup_{i \in [1, n]} L_i = L \), \( n \) 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 \( L \) to find the best choice for \( L_1 \) to \( L_n \), 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 \( q \xrightarrow{e} q’ \) and the output are two transitions \( q \xrightarrow{L_1} q_1 \) and \( q \xrightarrow{L_2} q_2 \) such that minimum number of global events are opaque to the transition.

**Constants.** For every atomic proposition \( p \in \text{AP} \) and every global event \( s \in L \), we employ a Boolean constant \( a_p^s \) defined as follows:

\[
a_p^s = \begin{cases} 
\text{true} & \text{if } p \in s \\
\text{false} & \text{if } p \notin s
\end{cases}
\]

**Variables and functions.** For every global event \( s \in L \), we define two Boolean variables \( x_{L_1}^s \) and \( x_{L_2}^s \), meaning that \( x_{L_1}^s = \text{true} \) if \( s \in L_1 \), otherwise \( x_{L_1}^s = \text{false} \). Likewise, \( x_{L_2}^s = \text{true} \) if \( s \in L_2 \), otherwise \( x_{L_2}^s = \text{false} \). We define an operator \( \circ \) between a Boolean variable \( x \) and a constant \( a \) as follows:

\[
x \circ a = \begin{cases} 
a & \text{if } x = \text{true} \\
\text{true} & \text{if } x = \text{false}
\end{cases}
\]

For each atomic proposition \( p \in \text{AP} \), we introduce two Boolean variables \( y_{L_1}^p \) and \( y_{L_2}^p \) with the following meaning:

\[
y_{L_1}^p = \begin{cases} 
\text{true} & \text{if } \forall s \in L_1 : p \in s \\
\text{false} & \text{otherwise}
\end{cases}
\]

\[
y_{L_2}^p = \begin{cases} 
\text{true} & \text{if } \forall s \in L_2 : p \notin s \\
\text{false} & \text{otherwise}
\end{cases}
\]

Analogously, for each atomic proposition \( p \in \text{AP} \), we introduce two binary integer variables \( w_{L_1}^p \) and \( w_{L_2}^p \). We also include two Booleans \( v_{L_1}^p \) and \( v_{L_2}^p \), whose meaning is explained later in the set of SMT constraints. For each atomic proposition \( p \in \text{AP} \), we define two binary integer variables \( w_{L_1}^p \) and \( w_{L_2}^p \) (for the purpose of counting and optimization) as follows:

\[
w_{L_1}^p = \begin{cases} 
0 & \text{if } v_{L_1}^p = \text{true} \\
1 & \text{otherwise}
\end{cases}
\]
\[ w_{\mathcal{L}_2}^p = \begin{cases} 0 & \text{if } v_{\mathcal{L}_2}^p = \text{true} \\ 1 & \text{otherwise} \end{cases} \]

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

\[ x_s = \neg x_s^{\mathcal{L}_1} \]

The constraints to encode the meaning of variables \( y_{\mathcal{L}_1}^p \) and \( y_{\mathcal{L}_2}^p \) are as follows:

\[ y_{\mathcal{L}_1}^p = \bigwedge_{s \in \mathcal{L}} (x_s^{\mathcal{L}_1} \circ a_s^p) \]
\[ y_{\mathcal{L}_2}^p = \bigwedge_{s \in \mathcal{L}} (x_s^{\mathcal{L}_2} \circ a_s^p) \]

It is easy to verify that \( y_{\mathcal{L}_1}^p \) evaluates true if and only if for every event \( s \in \mathcal{L}_1 \), we have \( p \in s \), and \( y_{\mathcal{L}_2}^p \) evaluates true if and only if for every event \( s \in \mathcal{L}_2 \) we have \( p \notin s \). Here, the variables \( y_{\mathcal{L}_1}^p \) and \( y_{\mathcal{L}_2}^p \) for all \( p \in \text{AP} \) correspond to the event \( s \) being opaque to \( y_{\mathcal{L}_1}^p \) and \( y_{\mathcal{L}_2}^p \). Likewise, for variables \( y_{\mathcal{L}_2}^p \) and \( y_{\mathcal{L}_2}^p \), we add the following constraints:

\[ y_{\mathcal{L}_2}^p = \bigwedge_{s \in \mathcal{L}} (x_s^{\mathcal{L}_2} \circ a_s^p) \]
\[ y_{\mathcal{L}_2}^p = \bigwedge_{s \in \mathcal{L}} (x_s^{\mathcal{L}_2} \circ a_s^p) \]

Finally, we need to count the number of opaque events in \( y_{\mathcal{L}_1}^p \) and \( y_{\mathcal{L}_2}^p \) (respectively, \( y_{\mathcal{L}_1}^p \) and \( y_{\mathcal{L}_2}^p \)). Although we have the concept of opacity defined for an event \( s \), here we have a variable \( v \) for each \( p \in \text{AP} \), thus tying it with the definition. Hence, we add the following assertions:

\[ v_{\mathcal{L}_1}^p = y_{\mathcal{L}_1}^p \lor y_{\mathcal{L}_2}^p \]
\[ v_{\mathcal{L}_2}^p = y_{\mathcal{L}_2}^p \lor y_{\mathcal{L}_2}^p \]

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

\[ \min \sum_{p \in \text{AP}} (w_{\mathcal{L}_1}^p + w_{\mathcal{L}_2}^p) \]

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.

### 5.3 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 \( M_3^q = (\Sigma, Q, q_0, \delta, \lambda) \) and transforms it into an extended monitor \( M_4^q = (\Sigma, Q, q_0', \delta_e, \lambda_e) \) as output using the above SMT-based optimization technique. We now describe the details of this transformation in Algorithm 4:

- If a transition does not need to be split, we simply add the original transition to the extended monitor (Lines 28 and 29).
- For each transition that should be split, we apply the above SMT-based optimization technique described in Section 5.2. 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 \( q \xrightarrow{e} q' \) is not a self-loop (Lines 10 – 13), then two transitions \( q \xrightarrow{e} q_1 \) and \( q \xrightarrow{e} q_2 \) 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 \( q \xrightarrow{e} q \) (Lines 15 – 20), then two transitions \( q \xrightarrow{e} q_1 \) and \( q \xrightarrow{e} q_2 \) 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 \( q_1 \) to \( q_2 \) (Line 20) for the events not in the original self-loop.
- Finally, we include the incoming transitions to each state (Line 28) and remove labels that are have no opacity issues (Line 31).

We repeat the loop until no transition needs to be...
The reader can test that running Algorithm 4 on the LTL₀ monitor in Fig 2, will result in the extended LTL₀ monitor in Fig. 3.

We now show the soundness of Algorithm 2 (as defined in the problem statement in Section 3.4) when augmented by an extended LTL₀ monitor as constructed by Algorithm 4.

**Lemma 1.** For \( \alpha \in \Sigma^* \) be a finite trace and \( \varphi \) be an LTL formula with \( M^\varphi = (\Sigma, Q, q_0, \delta, \lambda) \) as the LTL₀ monitor. Using Algorithm 4 we get \( M^\varphi_3 = (\Sigma, Q_e, q'_{0}, \delta_e, \lambda_e) \) such that \( \lambda(\delta(q_0, \alpha)) = \lambda_e(\delta_e(q'_{0}, \alpha)) \).

**Proof.** Let \( \alpha = s_0s_1 \ldots s_n \). We prove that for some \( i \in [0, n], q \xrightarrow{s_i} q_1 \in \delta \Rightarrow q \xrightarrow{s_i} q_{01} \in \delta_e \) such that \( \lambda(q_1) = \lambda(q_{01}) \).

Case 1: \( q = q_1 \)

\( (\Rightarrow) \) This means that \( q_1 \) was split into multiple state which includes \( q_{01} \). As can be seen in Algorithm 4, lines 16-20, the states \( q' \) is split into \( q_1 \) and \( q_2 \), and the self-loop is preserved by having a loop within the states it was split into. Also in lines 22-24, all outgoing and incoming edges of \( q' \) is preserved with the label of \( q' \) being transferred to both \( q_1 \) and \( q_2 \) in line 29. Thus, \( \lambda(q_1) = \lambda(q_{01}) \).

\( (\Leftarrow) \) Trivial

Case 2: \( q \neq q_1 \)

\( (\Rightarrow) \) This means that \( q_1 \) was split into multiple state which includes \( q_{01} \). As can be seen in Algorithm 4, lines 10-13, the states \( q' \) is split into \( q_1 \) and \( q_2 \), and the transitions are preserved by having a a transition from \( q \) to both \( q_1 \) and \( q_2 \) respectively. Also in lines 22-24, all outgoing and incoming edges of \( q' \) is preserved with the label of \( q' \) being transferred to both \( q_1 \) and \( q_2 \) in line 29. Thus, \( \lambda(q_1) = \lambda(q_{01}) \).

\( (\Leftarrow) \) Trivial

Thus, \( \lambda(\delta(q_0, \alpha)) = \lambda_e(\delta_e(q'_{0}, \alpha)) \) \( \square \)

**Lemma 2.** Let \( \alpha \in \Sigma^* \) be a finite trace and \( \varphi \) be an LTL formula. The return value of Algorithm 2 augmented with an extended LTL₀ monitor as constructed in Algorithm 4 is \([\alpha \models \varphi_3]\) by every monitor process in the presence of up to \( t \) crash failures.

**Proof.** We prove Lemma 2 in three steps, similar to the proof technique for consensus in synchronous networks (e.g., the FloodSet algorithm) [19]. First, we prove that at the end of the inner for-loop, \( LS_i \) 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 \( M \) 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 \( |LS_i| = 1 \), 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 \( |LS_i| = 2 \) (the case for \( i > 2 \) can be trivially generalized). This means that at least two monitor processes sent a message containing two possible LTL₀ 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 LTL₀ monitor states, because the value of some atomic proposition \( p \in AP \) is unknown, i.e., \( S(p) = \varnothing \). However, this scenario contradicts our assumption on event coverage (see Section 3) in our computation model.
  - The second scenario is that \( q_1 \) and \( q_2 \) are possible LTL₀ monitor state, because \( s_j \) is opaque to some outgoing transitions from \( q_{current} \) in the LTL₀ monitor. This case contradicts with our construction of extended LTL₀ 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 \( |LS_i| = 1 \). We also know that \( |LS_i| = 1 \) (for all \( r \in [1, t + 1] \)) and \( LS_i \) contains the same content as \( LS_i \). Let this content be an LTL₀ 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 our construction of extended LTL₀ monitor in Algorithm 4. Hence, \( q \) would be the monitor state that a centralized monitor would compute. The induction step is now trivial: it is straightforward to show that for any valid \( q_{current} \) and any \( s_{jr} \), 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 LTL₃ monitor state $q$, where $\lambda(q) = [a \models \varphi]$.

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 $\lambda(q) = [a \models \varphi]$. Since there are at most $t$ crash failures, there has to be some round $r$, where no failures occur. Recall that in Section 3, we assume that if a monitor crashes and this monitor is the only one that is aware of some proposition $p \in AP$, 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 LTL₃ monitor as constructed by Algorithm 4.

We now extend our technique by monitors that evaluate a formula every $l \geq 1$ global states rather than after every global state. That is, the for-loop in Algorithm 2 iterates every $\lfloor k/l \rfloor$ and instead of a partial view $S_i^{t_0}$ it evaluates a sequence of partial views $S_i^{t_0}S_i^{t_1}...S_i^{t_{l-1}}$ and so forth and, hence, the monitors communicate every $l$ state (rather than every single state). To this end, let us recursively extend $\mu$ from a single partial view and a monitor state transition (i.e., $\mu(S, q)$ as defined in Section 4) to a sequence of partial views $S_i^{t_0}S_i^{t_1}...S_i^{t_{l-1}}$ and a set of monitor states $Q' \subseteq Q$ as follows (denoted $\mu_l$):

$$\mu_l(S_i^{t_0}S_i^{t_1}...S_i^{t_{l-1}}, Q') = \mu_1(S_i^{t_{l-1}}, \mu_{l-1}(S_i^{t_0}S_i^{t_1}...S_i^{t_{l-2}}, Q'))$$

The base case of the recursion, $\mu_1(S_i^{t_0})$ is defined as Equation 1.

**Theorem 1.** Let $\varphi$ be an LTL formula, $\alpha \in \Sigma^*$ with $|\alpha| = k$ and $l$ a natural number, where $l \leq k$. Given the generalization of $\mu$ to $\mu_l$, the output of Algorithm 2 for $\mu_l$ is $[\alpha \models_3 \varphi]$.

**Proof.** We prove the theorem by induction over $l$. The base case, (i.e., $l = 1$), trivially holds by Lemma 2. For the inductive step, let the statement of the theorem be true for $l$, meaning that the verdict of the algorithm is indeed for length $l$, same as the verdict of an LTL₃ monitor. We have to show that it also holds for $l + 1$. This case is also discharged by Lemma 2, since state by state evaluation results in the correct LTL₃ evaluation.

**Theorem 2.** Let $\varphi$ be an LTL formula and $\alpha \in \Sigma^*$ be a finite trace. The message complexity (i.e., space complexity of LS) of Algorithm 2 using an extended LTL₃ monitor is

$$O\left(\log(|M_3^\varphi|) \cdot |AP| \cdot n^2(t + 1)|\alpha|\right)$$

where $n$ is the number of distributed monitors.

**Proof.** We analyze the complexity of each part of Algorithm 2:

- The algorithm has a nested loop. The outer loop iterates exactly $|\alpha|l$ times.
- The inner loop iterates exactly $t + 1$ times.
- In the inner loop each monitor process sends $n$ messages to all other monitors 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 $M_3^\varphi = (\Sigma, Q_0, q_0, \delta, \lambda)$ be an LTL₃ monitor and $M_3^\varphi = (\Sigma, Q_e, q_0^e, \delta_e, \lambda_e)$ be its extended monitor constructed by Algorithm 4. The algorithm may split a transition at most $|AP|$ number of times. Hence, we have

$$|Q_e| \leq 2|Q| \cdot |AP|.$$ 

Recall that each message contains the possible states of the extended LTL₃ monitor. This means each message in Algorithm 2 needs

$$O\left(\log(|M_3^\varphi|) \cdot |AP|\right)$$

bits for each message. Recall that the size of an LTL₃ monitor is the number of its state, i.e., $|M_3^\varphi| = |Q|$ and $|AP|$ corresponds to the number of bits required for representing all the atomic proposition. Hence, the message complexity is

$$O\left(\log(|M_3^\varphi|) \cdot |AP|\right).$$

We note that if the distributed monitors verify the finite computation $\alpha$ every $k$ state (see Theorem 1), then the $|\alpha|$ factor reduces to $\lceil|\alpha|/k\rceil$.

**Theorem 3.** Rather than going through $t + 1$ rounds of communication with peer monitors, each monitor can only go through $k + 1$ rounds where $k$ denotes the maximum number of monitor crashes that are possible in a particular state and $t$ is the maximum number of monitor crashes that were possible in the initial state without loss of any information or correctness.

**Proof.** We first take a look into why we need $t + 1$ rounds to come to a common conclusion at the first place. It is to accommodate for any monitor crashes during communication such that no information is lost. We need $t + 1$ rounds, since the system can only have a maximum of $t$ number of monitor crashes.

Here, we consider a synchronous system, i.e., all the monitors share the same global clock, thus whenever a monitor does not receive from another monitor the former considers the later has crashed. This holds with our assumption that once a monitor has crashed it cannot revive itself and the network we are using is clean, i.e., all the messages sent are received by the receiver and none gets lost in transmission.

For the first state, the maximum number of possible crashes are $t$. But for any subsequent states, the maximum number of possible monitor crashes depends upon the number of monitor crashes that has already taken place in the states leading up to it. For example, for the $i$-th state, the maximum number of possible monitor crashes is $k = t - c$, where $c$ denotes the number of already crashed monitors in the system in the previous $i - 1$ states. Thus, we can only go through $k + 1$ rounds accounting for the maximum $k$ crashes that is possible in the present state.
6 EXPERIMENTAL RESULTS

In this section, we present the results of our experiments on monitoring formulas with respect to a synthetic model of the system and monitoring correctness and behavioral specifications on the Orange4Home [18] dataset for IoT.

6.1 Synthetic Experiments

6.1.1 Setup

We evaluate our decentralized system using different LTL formulae generated from specification patterns mentioned in [21]. The corresponding monitor is generated using LTL3 tools [22]. Each of the following experiments were conducted on the following combinations of total number of monitors in the system and the maximum number of crashes (t) that the system is prone to have:

- # of Monitors = 10; t = 4, 5, 6, 7, 8
- # of Monitors = 20; t = 10, 12, 14, 16, 18
- # of Monitors = 30; t = 10, 15, 20, 25, 28

We also extend our setting of the system under observation by considering different probability distributions (uniform, Bernoulli (0.1), and Bernoulli (0.9)) for different aspects of the system, namely: read distribution of an atomic proposition given the set of all monitors and crash distribution of a monitor given the execution state. The number of crashes per state is controlled by a right skewed normal distribution $N(\mu = 0, \sigma = 1.5)$ where all numbers are positive rounded to the nearest decimal.

A monitor may crash at two different points during its execution. The first being immediately after having read state of the system and the next being while communicating. If a monitor crashes immediately after reading the state of the system, i.e., before communicating with the rest of the monitors, we assume that there exists at least one other monitor who read the same atomic propositions. This is done to make sure, the value of an atomic proposition is not lost with the monitoring which crashed. On the other hand, if a monitor crashes while communicating, we assume that it was able to send its partial observation to at least one other monitor in the system which did not crash in the same round. This is also done, to ensure that the information of the state of the execution is not lost with a monitor crashing.

As can be seen in Fig. 6, the distribution of monitor crashes for Bernoulli (0.9) is more left skewed when compared with uniform distribution. This is because in Bernoulli (0.9), the likelihood of a monitor crashing is higher compared to uniform where it is 0.5. Higher crash likelihood makes the monitor in the system crash earlier till the system reaches the maximum number of crashes allowed. We also notice that the likelihood of a monitor crashing is dependent on the read distribution of the atomic proposition over the monitors. More number of monitors read a atomic proposition when distributed uniformly compared to Bernoulli (0.1). As mentioned earlier, a monitor only crashes if there exists another monitor who has read the same atomic propositions. Thus, the likelihood of a monitor crashing is more for a read distribution of uniform compared to Bernoulli (0.1). The implementation is publicly available on GitHub.

6.1.2 Analysis of Results

As mentioned earlier, we have put our system to the test with respect to all the LTL formulas mentioned in [21] (for specification patterns) under all the different scenarios explained above but, for both space and redundancy of similar observations, below we only mention results for the

Fig. 6: Crash distribution over a trace of length 100 for different combination of crash and read distribution and all the experiments were performed on a 2017 MacBook Pro with 3.5 GHz Dual-Core Intel i7 processor and 16GB DDR3 RAM.

The partial view of a monitor should be such that the global observation is equal to the partial views of all the monitors taking together. If the global observation is denoted by $G_{S_j}$, then the partial observation, $S_i^j$, for monitor $i$ should be such that:

$$G_{S_j} = \bigcup_{i=1}^{n} S_i^j$$

This condition is necessary as this guarantees that the entire global observation is observed by the list of all monitors taken together.

Similar to the tool, DECENT-MON [23], we test out each system configuration on three different traces where the probability of occurrence of an atomic proposition given a state is controlled by uniform distribution and Bernoulli distribution with 0.1 and 0.9 as a parameter. In our experiments, we study and report on the following metrics:

- The average number of rounds needed to traverse through the entire trace sequence.
- The number of messages, #msg., exchanged between monitors.
- The average size of a message, size (msg.), exchanged between the monitors.
- The number of monitor crashes the system was a victim of.

All of our experiments are run sufficiently enough (20 times) and each result is accompanied by the corresponding 95% confidence interval.

6.1.2 Analysis of Results

As mentioned earlier, we have put our system to the test with respect to all the LTL formulas mentioned in [21] (for specification patterns) under all the different scenarios explained above but, for both space and redundancy of similar observations, below we only mention results for the
Impact of monitor crashes. As expected a higher number of monitor crashes results in an increase in the average number of rounds when monitoring. In Fig. 7a, for LTL formula $\varphi_4$, we observe that the average number of rounds is significantly improved when accounting for only the number of crashes that are possible given a state of the execution. In other words, as more and more monitor crashes, the less number of further monitor crashes can there be, thereby reducing the total number of communication rounds that is required. For example, in a system with $t = 8$ and 5 monitors have already crashed, there can be only 3 more monitor crashes taking place. Thus, we update the number of communication rounds that is required to be $3 + 1 = 4$. This reduces the net number of rounds that was required.

In Fig. 7b, we see for $\varphi_4$ that with increase in the number of monitor crashes, the number of messages exchanged among the monitors increase as well. This is due to the fact that with more number of monitor crashes, the value of $t$ is more and there-by increasing the number of communication rounds required. During each round a monitor in the system communicates its observation with all other monitors in the system, thereby making the total number of messages directly proportional to the number of active monitors and also the number of rounds.

Following our setup described in Fig. 6, the distribution of crashes also have an effect on the average number of rounds and the number of messages being passed in the system. The more left skewed will be the distribution of the monitor crashes, the less average number of rounds are required to come to a consensus among the monitors. This is because a left skewed monitor crash distribution equates to the mean number of monitors present in the system being low and there-by lower number of rounds as well as number of messages.

Communication after $l$ states: We test our algorithm on different values of $l$, starting from 1 when the communication between monitors take place after every state and going all the way to 50 when the monitors communicate only twice for a trace length of 100. As stated in Theorem 1, the correctness of the protocol is not affected by changing the values of $l$ however as seen in Fig. 8 for different LTL specifications, the average number of rounds and average number of messages decreases with increasing values of $l$. For lower $l$, the communication takes place more often than higher values of $l$ and thus accounting for higher values of number of rounds and number of messages.

The average size of messages increases with an increase in the value of $l$. This is because the size of a message depends on the number of states present in the local observation of a monitor. With communication happening after every $l$ states, the local observation constitutes of a larger number of states than when it was happening after every state. This can be seen when comparing the results of Figure 8c for different LTL formula. The size of messages for $\varphi_{38}$ is substantially larger when compared to that of others due to the more number of states in its updated $\text{LTL}_3$ monitor automata along with higher number of atomic proposition. Comparing the message size in Fig. 8c when sharing the actual observation and the symbolic representation consisting of the states of the extended automata, we see that for $l > 15$, the symbolic representation will have an advantage over the actual representation. Moreover, with increasing value of $l$, the size of the message increases linearly where as the symbolic view gets bounded by the size of the monitor, suggesting sharing the symbolic view is better. The network throughput can be easily calculated taking the average number of messages and average size of the messages into consideration.

We also see that with increasing the value of $l$, the number of monitor crashes decreases. This is because, with increasing the value of $l$, communication is limited to only after every $l$ states and there-by decreasing the number of communicating rounds (keeping the length of the trace constant) and there-by decreasing the number of monitor crashes. However, if we did not have constant length of the trace, the number of communication round would increase given the more number of states to be verified, and there-by
making more monitors vulnerable to crash. Taking all the plots into consideration, we observe that the benefit from the lower number of rounds and messages out-weights the drawback from the increase in the size of messages for any value of \( l \geq 5 \).

### 6.2 Orange4Home Dataset

Orange4Home [18] is a dataset capturing routines of daily living in Amiqual4Home’s smart home environment. It is a result of a joint work between Orange Labs and Inria. The dataset consists of around 180 hours of recording of activities of daily living of a single occupant, spanning 4 consecutive weeks of work days. The dataset contains recordings of a total of 236 sensors scattered throughout the apartment and for 20 different classes of activities. We divide all specifications into two categories: (1) Behavioral correctness: monitor the correctness of the different sensors (2) Activity of Daily Living (ADL): monitoring the activity that the occupant is up to using the values of different sensors. In Fig. 9, we show the results for various values of \( l \) keeping the read and crash distribution to be uniform and Bernoulli \((0.1)\) respectively and report on the number of rounds, number of messages, size of the message and actual number of monitor crashing for a system with 30 monitors and \( t = 20 \) monitoring the following specifications:

\[
\begin{align*}
\varphi_{oh,1} &= [\text{switch} \rightarrow \varnothing(\text{light} \ U \ 
eg\text{switch})] \\
\varphi_{oh,2} &= [\varnothing \leq 5 (\text{cooktop} \ \text{V} \ \text{oven})] \\
\varphi_{oh,3} &= [\varnothing \leq 5 (\text{kitchen}_\text{sink} \ \text{V} \ \text{kitchen}_\text{fridge} \ \text{V} \\
& \ \ \ \ \text{kitchen}_\text{cupboard})] \\
\varphi_{oh,4} &= [\varnothing \leq 5 (\text{cooktop} \ \text{V} \ \text{oven} \ \text{V} \ \text{kitchen}_\text{sink} \ \text{V} \\
& \ \ \ \ \text{kitchen}_\text{fridge} \ \text{V} \ \text{kitchen}_\text{cupboard} \ \text{V} \\
& \ \ \ \ \text{kitchen}_\text{dishwasher})]
\end{align*}
\]

First we construct the equivalent LTL3 monitors using Algorithm 4. The change in the number of states of the final automata can be observed in Table 1. Monitoring ADL specifications involve the system keeping a track of the passage of time, essential in monitoring a time bounded specification as is the case with \( \varphi_{oh,2} \) through \( \varphi_{oh,4} \). Apart from a similar observation to the synthetic data for increasing values of \( l \), we observe in Fig. 9 that monitoring specifications involving more number of atomic propositions have higher message size. The higher message size can be explained by Theorem 2 which shows the message complexity when using an extended LTL3 monitor is directly proportional to |AP|. Additionally, higher value of \( l \) decreases the number of communicating rounds and thus accounting for lower number of monitor crashes. Subsequently, lower number of monitor crashes equates to higher number of active monitors in the system and therefore higher number of rounds and higher number of messages.

### 7 Related Work

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

#### 7.1 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 submonitors 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 the following fundamental ways: (1) the framework in [6], [7] is fault-free, and (2) we do not assume that components have disjoint alphabet. The monitors in our case observes a shared set of propositions and this creates ambiguity among monitors and brings inconsistency issues to the problem, which are absent in [6], [7].

### 7.2 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], [24], the authors in [5] show that employing the four-valued LTL [25] 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 ≥ 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.

### 7.3 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.

### 7.4 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 [26]. Computation slicing [27] 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 [27], 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]. A similar partially synchronous runtime verification technique for a more broader LTL specification is introduced in [28].

### 8 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 (LTL). 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 monitors. Since LTL 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 monitors. For an LTL formula φ, our SMT-based algorithm only increases the size of an LTL monitor Mφ only by a factor of $O(\log |M_φ| \cdot |\text{AP}|)$ (communicating explicit observations would require $O(|\text{AP}|)$ bits), where AP is the set of atomic propositions that describe the global state of the underlying system. We put our approach through...
an extensive number of experiments with varying distribution responsible for modeling monitor crashes, atomic prepositions distributed over the states and also the partial observation of each monitor. Through this extensive experimentation we learn that limiting the number of rounds to not go till \( t \) and communication between monitors now happening after every \( k \) states reduces the average number of rounds, number of messages sent considerably with only the average size of the message going up by a small quantity.

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. Also, in order to transition our approach to a more practical setting, we plan to incorporate timed temporal logics where monitors evaluate formulas within certain time intervals.

**References**


Ritam Ganguly received his bachelor's degree in Computer Science and Engineering from Maulana Abul Kalam Azad University of Technology, India, in 2017 and is currently pursuing his Ph.D. in Computer Science from Michigan State University. His research interests include distributed computing, formal methods, and computer security.

Shokufeh Kazemloo is a clinical data science engineer at Rad AI. She received her Master's degree in Computer Science from MacMaster University, Canada, in 2018 and her Bachelor's in Chemical Engineering in 2009 from Sharif University of Technology, Iran.

Borozzo Bonakdarpour received his bachelor's degree in Computer Engineering – Software from The University of Esfahan, Iran, in 1999 and his Ph.D. in Computer Science from Michigan State University in 2009. He is currently an Associate Professor of Computer Science with the Department of Computer Science and Engineering at Michigan State University. His research interests include distributed computing, formal methods, cyber-physical systems, and computer security.
**APPENDIX**

**Example**

In this example, we go over our entire monitoring approach including the monitor transformation algorithm. We consider $\varphi = \Diamond (a \land b)$ with 3 monitors and $k = 1$. First, we motivate the need of the monitor transformation algorithm. Let’s consider the state, $s = a \land b$ with

$S_1 = \{ a : \text{true}, b : \top \}$,
$S_2 = \{ a : \top, b : \text{true} \}$
and
$S_3 = \{ a : \top, b : \text{false} \}$. Therefore, $LS$ computed by the monitors are

$LS_1 = \{ q_0, q_T \}$,
$LS_2 = \{ q_0, q_T \}$
and
$LS_3 = \{ q_0, q_T \}$. Communicating within themselves they are unable to come to a common conclusion which is the same as it would have been with a centralized monitor. Thus, we need the monitor transformation algorithm, so specifically make the self-loop on $q_0$ more discrete.

Our monitor transformation algorithm identifies the transition $L = \{ \{ a \}, \{ b \}, \emptyset \}$ to be split into $L_1 = \{ \{ a \}, \emptyset \}$ and $L_2 = \{ \{ b \} \}$. To begin, we split the state $q_0$ into $q_{01}$ and $q_{02}$ and attach the $L_1$ and $L_2$ to the respective states. To conserve the self-loop, we add the transitions $q_{03} \xrightarrow{a} q_{01}$ and $q_{01} \xrightarrow{a} q_{03}$. Next, we keep all the rest of the transitions in the automata by attaching all the outward and incoming transitions of $q_0$ to both $q_{01}$ and $q_{02}$. Since, no other transitions needed to be split, we get our final modified automata.

Here, we consider a trace, $\alpha$ with three states as shown in Table 2 with each monitor having the corresponding partial view. Computing the local states after reading the first state, they are

$LS_1 = \{ q_{01}, q_T \}$,
$LS_2 = \{ q_{01}, q_{02}, q_T \}$
and
$LS_3 = \{ q_{01} \}$. Therefore, after communication the union comes out to be $\{ q_{01} \}$. Reading the second state, the local computations are

$LS_1 = \{ q_{01}, q_{02}, q_T \}$,
$LS_2 = \{ q_{01}, q_{02} \}$
and
$LS_3 = \{ q_{01} \}$. Now we assume that monitor 3, crashes after sending its data to monitor 1. So after round 1 of communication, $LS_1 = \{ q_{01} \}$ and $LS_2 = \{ q_{01}, q_{02} \}$. After another round of communication, we have $LS_1^2 = \{ q_{01} \}$ and $LS_2^2 = \{ q_{01} \}$. Since there was only 1 crash that was allowed, we go with 2 communication rounds. The local states after every round is shown in Table 3.

Next, the two monitors compute the local states as

$LS_1 = \{ q_{01}, q_T \}$
and
$LS_2 = \{ q_{02}, q_T \}$
which after communication we have $\{ q_T \}$ on both the monitors. We observe that throughout the computation after every state, the $q_{\text{current}}$ always used to one automata state which was consistent with what it would have been in case we had a centralized monitor. There-by showing the correctness and soundness of our approach.

**Table of LTL Formulas**

Below mentioned are the list of LTL Formulas that were used to experiment on our algorithm. Each Formula is categorized as Absence, Existence, Bounded Existence, Universality, Precedence, Response, Precedence Chain, Response Chain and Constrained Chain. The size(Before) mentioned for each formula denotes the number of states present in the original LTL3 automaton. Size(After) denotes the number of states in the LTL3 automaton after the former was passed through Algorithm 4. The percentage change in the number of states is also mentioned in Table 4 below.

<table>
<thead>
<tr>
<th>$i$</th>
<th>$s_1$</th>
<th>$S_1$</th>
<th>$S_2$</th>
<th>$S_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>${ a }$</td>
<td>${ a : \text{true}, b : \top }$</td>
<td>${ a : \top, b : \text{false} }$</td>
<td>${ a : \top, b : \text{false} }$</td>
</tr>
<tr>
<td>2</td>
<td>$\emptyset$</td>
<td>${ a : \top, b : \text{true} }$</td>
<td>${ a : \top, b : \text{false} }$</td>
<td>${ a : \top, b : \text{false} }$</td>
</tr>
<tr>
<td>3</td>
<td>${ a, b }$</td>
<td>${ a : \text{true}, b : \top }$</td>
<td>${ a : \top, b : \text{true} }$</td>
<td>crash</td>
</tr>
</tbody>
</table>

**TABLE 2**

<table>
<thead>
<tr>
<th>$r$</th>
<th>$LS_1^r$</th>
<th>$LS_2^r$</th>
<th>$LS_3^r$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>${ q_{01}, q_{02}, q_T }$</td>
<td>${ q_{01}, q_{02} }$</td>
<td>${ q_{01} }$</td>
</tr>
<tr>
<td>1</td>
<td>${ q_{01} }$</td>
<td>${ q_{01}, q_{02} }$</td>
<td>crash</td>
</tr>
<tr>
<td>2</td>
<td>${ q_{01} }$</td>
<td>${ q_{01} }$</td>
<td>crash</td>
</tr>
</tbody>
</table>

**TABLE 3**
<table>
<thead>
<tr>
<th>No.</th>
<th>Type of formula</th>
<th>Formula</th>
<th>Size (Before)</th>
<th>Size (After)</th>
<th>Change (Times)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Absence</td>
<td><img src="1" alt="formulas" /></td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>Existence</td>
<td><img src="2" alt="formulas" /></td>
<td>4</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>Bounded Existence</td>
<td><img src="3" alt="formulas" /></td>
<td>3</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>Precedence</td>
<td><img src="4" alt="formulas" /></td>
<td>2</td>
<td>3</td>
<td>1.5</td>
</tr>
<tr>
<td>6</td>
<td>Response</td>
<td><img src="5" alt="formulas" /></td>
<td>3</td>
<td>4</td>
<td>1.3</td>
</tr>
<tr>
<td>7</td>
<td>Precedence Chain</td>
<td><img src="6" alt="formulas" /></td>
<td>4</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>Response Chain</td>
<td><img src="7" alt="formulas" /></td>
<td>4</td>
<td>5</td>
<td>1.25</td>
</tr>
<tr>
<td>9</td>
<td>Constrained Chain</td>
<td><img src="8" alt="formulas" /></td>
<td>2</td>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>

TABLE 4: List of formulas used to check our algorithm