Decentralized Runtime Verification of LTL Specifications in Distributed Systems

by

Mennatallah Hasabelnaby

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Science
in
Computer Science

Waterloo, Ontario, Canada, 2016

Abstract

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

Acknowledgements

I would like to thank my supervisor *Borzoo Bonakdarpour* for his continuous technical and financial support, encouragement, hard work and dedication. Special thanks to my husband who has always been my sounding board and who has helped monumentally in making this happen. Special thanks to my parents for always reminding me of my priorities.

Dedication

This is dedicated to my small family that is about to get a bit bigger.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Runtime Verification

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

Runtime verification is a lightweight technique where a *monitor* checks at run-time whether or not the execution of a system under inspection satisfies a given correctness property. Runtime verification complements exhaustive verification methods such as model checking and theorem proving, as well as incomplete solutions such as testing and debugging. A correctness property can be expressed in any specification language that is expressive enough for the property. Usually, the correctness properties are derived from the software requirements and represent desired safety or liveness properties. For example, a safety property for a traffic light system would be ”If the current signal becomes red, then the green signal can not be the next signal”. While an example for a liveness property for the same traffic light system would be ”Green signal should occur infinitely often” meaning that it should never be the case that the green signal never appears in the execution trace. Given such a correctness property, a monitor program is synthesized that reads the events that are added to the program’s execution trace in runtime, and decides wether the execution adheres to the correctness property or not.

Properties that deal with events’ chronological order or eventuality require an expressive language that can reason in the temporal domain, therefore first order logic is not used to express such properties. For example, there is no way to express the property ”Green signal should occur infinitely often” in first order logic. *Linear Temporal Logic* (LTL) [25] is a specification language that reasons in the temporal domain and is heavily used in the runtime verification community due to its simplicity and lightweight monitor synthesizes algorithms [1].

1.2 Distributed Programs Verification

The need for verification of distributed programs rises as our dependency on distributed system nowadays rises. The abundance of low commodity hardware in cloud computing data centers allows programmers to scale their distributed programs easily. Also, systems for coordination of agents such as robots or drones in swarms are considered distributed programs and are becoming very popular in many domains such as:

- Search and Rescue
- Traffic Monitoring
- Agriculture
- Inspection and Identification

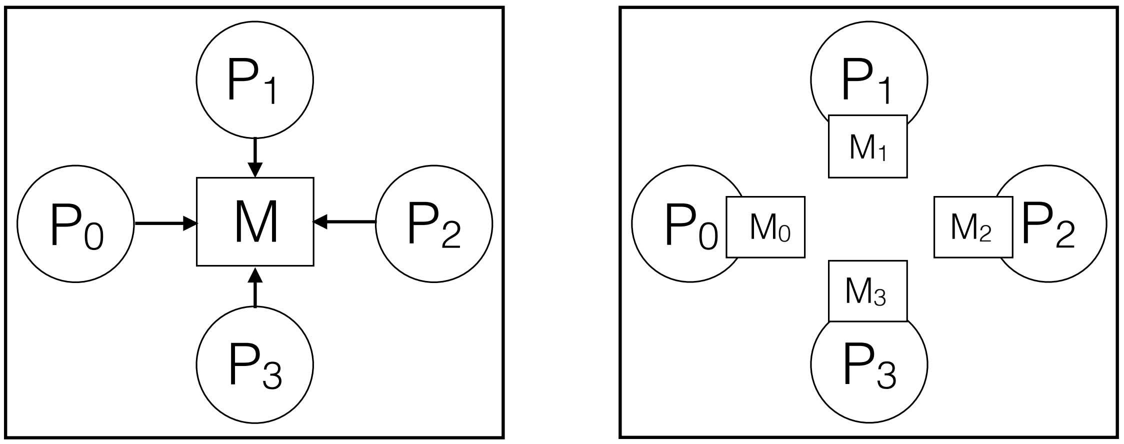
However, distributed programs are more error-prone and are notoriously harder to debug. Also, some bugs present themselves under specific conditions and corner cases that might be missed by the system programmer. Therefore, the need for formal runtime verification rises. Given some global correctness properties over the execution of all processes, runtime verification tests that the distributed system does not violate these properties.

1.2.1 Distributed Programs Monitoring Challenges

Runtime verification for distributed programs is more challenging than sequential programs since the monitor needs to construct the events trace from all the processes at run-time and then reason about their correctness. Moreover, designing a runtime monitor for an asynchronous distributed system is an especially difficult task. This is because asynchronous programs do not share a global clock and, hence, processes can have different processing speeds and can suffer from clock drifts. Therefore, events cannot be ordered based on time. Thus, *lattice theory* [12], *partial orders* [15] and *global snapshot detection* [6] are used to find partial orders between events and construct the set of possible paths for the execution trace. However, it might be impossible to determine the actual execution path that occurred in physical time. This implies that one has to monitor all possible execution paths based on possible partial orders. We note that runtime verification of all possible execution paths should not be confused with model checking where all possible executions are checked. In the former, there is an execution trace that can yield multiple possible execution paths. While in the latter, there is no specific execution trace, rather all the possible execution traces are checked. In this work, we attempt to design an algorithm that monitors the correctness of asynchronous distributed programs.

1.2.2 Distributed Programs Monitoring Configurations

There are many possible configurations with respect to the monitoring a distributed system. e.g., centralized monitor, decentralized monitors and migrating monitor [2], monitor choreography [8]. Fig 1.1 shows the centralized monitor configuration and the decentralized monitors configurations. In the centralized monitoring configuration, there is only one monitor process that can reside on one of the program nodes or on an additional node dedicated for monitoring. The central monitor receives every event that occurs at each program node and attempts to construct the possible execution traces. In decentralized monitoring, each program node P_i has a monitor process M_i that communicates with it and can read the local events that occur at P_i, and communicates with other monitor processes via peer-to-peer messages as needed to construct the possible execution traces.



(a) CentralizedMonitoring (b) Decentralized Monitoring

Figure 1.1: Monitor Design

While centralized monitoring suffers from many disadvanatges such as single point of attack/failure and centralized computation and communication overhead, its design is fairly simple compared to decentralized monitoring. Simply all processes send their events to the central monitor which is then responsible for ordering the events and constructing the possible execution paths.

On the other hand, decentralized monitoring does not expose a single point of failure or attack, since monitoring processes are independent and request only information required to verify the correctness properties from other processes. Also, program processes in decentralized monitoring are notified about violations or satisfactions faster than in centralized monitoring since the monitor process resides on the same node as the program process. However, its design can be complicated since each monitoring process is responsible for evaluating some or all possible execution paths, evaluating missing information and requesting them from other monitor processes.

Also, decentralized monitoring could possibly suffer from security threats if the distributed program is running on untrusted nodes that can abuse the information acquired from other program nodes, therefore it is best suited for closed systems where all nodes are trusted.

We discuss the different approaches to monitoring in more details in Chapter 6.

In this thesis, we use the decentralized monitoring approach since our proposed decentralized monitoring algorithm optimizes the monitoring according to the process location (i.e. a monitoring process M_i residing at a program process P_i is optimized to monitor possible execution paths in which P_i participates), thus contributing to decreasing communication and computation overhead.

1.3 Thesis Statement

In this thesis, we claim that, although the problem of runtime monitoring of asynchronous distributed systems suffers from state-space explosion due to events concurrency and execution nondeterminism, there can be a decentralized distributed monitoring algorithm that can monitor a distributed program for LTL properties in a sound and complete fashion while avoiding exploring all states. The proposed algorithm achieves this end by (1) generating a deterministic finite state machine (FSM) given the LTL property [1], (2) each monitoring process explores the global states that can change the state of the FSM instead of exploring all possible global states.

1.4 Contributions

The main contribution of this work is a novel decentralized algorithm for runtime verification of distributed programs. In our setting, a distributed program consists of a set of asynchronous processes that communicate using message-passing primitives over reliable channels. Our algorithm conducts runtime verification for the 3-valued semantics of the linear temporal logic (LTL_3) [1], designed for reasoning about LTL properties for finite executions. It indeed addresses the shortcomings of the related work discussed in Chapter 6. In particular:

- it does not assume a global clock.
- it is able to verify temporal properties and not just safety predicates at run-time.
- it is sound and complete.

Intuitively, our technique works as follows:

- Each process in the program is composed with a *monitor process*. Each monitor process is augmented with a *monitor automaton* for each LTL_3 property under inspection. The monitor automaton is a deterministic finite state Moore machine that defines how the monitor process should evaluate a property. The states of the automaton are labeled by evaluation verdicts while the transitions are labeled by global-state predicates (see Fig. 2.3 for an example). Thus, each monitor process should be able to evaluate these predicates. To this end, we adapt the lattice-theoretic technique proposed in [7] for detecting global-state predicates at run-time. However, due to the existence of concurrent events, a monitor process may construct different finite executions of consistent global states.
- Consequently, the monitor process maintains a *set* of possible evaluation verdicts. This set evolves over time, meaning that monitoring verdicts may be added or removed depending upon the truthfulness of predicates and the structure of the monitor automaton. Adding more than one monitoring verdict only happens due to the existence of concurrent events. We argue that maintaining a set of possible verification verdicts due to unresolvable nondeterminism is indeed what a decentralized monitor should propose for verification of distributed programs since it may uncover intricate existing bugs.

  We note that collecting a set of possible verdicts does not contradict soundness since the actual sound verdict can not possibly be verified in the presence of nondeterministic execution, and also since each verdict in the set of possible verdict could have actually occurred. We emphasize that the final size of this set is generally very small (usually less than five even for very sophisticated properties), even if there is a high degree of concurrency.
- Finally, merging takes place, if over time, two execution paths have the same global state and verdict. We note that for the final global state (composed of the final events at each process) the maximum number of possible verdicts is the size of the set of automaton states. Also, our algorithm attempts to intelligently merge execution paths that are bound to join.

Our algorithm is *sound*, meaning that if the actual total order of events can be somehow constructed, then our algorithm identifies the verification verdict for the actual total order as one of the possible verdicts. It is also *complete*, meaning that among the set of verdicts computed by local monitors, at least one corresponds to the verdict for the actualtotal order of events.

Our algorithm is fully implemented and we report the results of sophisticated experiments on a case study on runtime monitoring of a network of iOS devices. Our experiments to measure the communication overhead of monitoring, as well as the delay in evaluating different properties on the network clearly shows that although our algorithm explores all possible execution paths, it does not result in an explosion in communication. In particular, due to the design of our algorithm that distributes the exploration among the nodes and avoids exploring states that does not change the automaton state, monitoring overhead grows only in the *linear order* of the number of processes and events that need to be monitored. These results clearly show that our algorithm elegantly works as a *lightweight* automated formal method for online reasoning about the correctness of a distributed application.

In our paper [21], we presented a version of this algorithm where the automatons used in the monitoring processes were not identical. Since the LTL property for a process included propositions from some specific processes (particularly the process's neighbouring processes). While in this work, the automatons are identical in every process, and every process participates in the property.

1.5 Organization of the Thesis

The rest of the thesis is organized as follows. In Chapter 2 we present our computation model for distributed programs and the specification language LTL$_3$. We introduce the formal statement of the problem in Chapter 3. Our runtime verification algorithm is presented in Chapter 4 along with the analysis and the proofs of correctness, while the results of our experiments are discussed in Chapter 5. In Chapter 6 we describe the current state of the art related work. Finally, we make concluding remarks and discuss future work in Chapter 7.

Chapter 2

Background

2.1 Distributed Programs

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

A *distributed program* $\mathcal{D} = \{P_1, P_2, \ldots, P_n\}$ is a set of n reliable processes that do not lie nor die. We assume that no two processes share a common variable. Processes communicate with each other over lossless FIFO channels using asynchronous messages whose time of arrival is unbounded. An *event* e in a process $P_i = \langle s_i^0, S_i, T_i \rangle$ in $\mathcal{D}$, where $1 \leq i \leq n$, is either:

- A local transition $(s_0, s_1) \in T$, called an internal event where the local state of P_i is changed.
- A message send, where the local state of P remains unchanged.
- A message receive, where the local state of P remains unchanged.

Fig. 2.1 shows our running example of a distributed program that consists of two communicating processes `P1` and `P2`, with local integer variables `x1` and `x2` (both initialized to 0), respectively. Each instruction on Lines 4-7 is an event denoted by $e_0^1 \cdots e_3^1$ (respectively, $e_0^2 \cdots e_3^2$) of process `P1` (respectively, `P2`).

Since send and receive events do not change the local state of a process $P = \langle s^0, S, T \rangle$, we represent their occurrence by a self-loop $(s, s) \in T$, where s is the state of occurrence. Thus, any event e can be associated with one and only state $\mathfrak{s}(e)$ reached by execution of e. An *event-trace*

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

```
{x2=0}
Process P2()
{
  recv(m1);
  x2=15;
  x2=20;
  send(P1,"world");
}
```

Figure 2.1: A distributed program.

of a process $P_i \in \mathcal{D}$ is a finite or infinite sequence of events $\eta = e_0^i e_1^i \cdots$ iff there exists a state trace $\sigma = s^0 \mathfrak{s}(e_0^i) \mathfrak{s}(e_1^i) \cdots$. We denote the set of all possible events in P_i by E_i and the set of all events of $\mathcal{D}$ by $E_{\mathcal{D}} = \cup_{i=1}^{n} E_i$. Throughout this work, we denote the set of all global states of program $\mathcal{D} = \{P_1, P_2, \ldots, P_n\}$ by $\Sigma = S_1 \times, \ldots, \times S_n$; i.e., the Cartesian product of local state space of all processes. A (global-state) *predicate* P is a subset of Σ defined using the following syntax:

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

where $1 \leq i, j \leq n$, $p_i \in AP_i$, and $p_j \in AP_j$. We denote the set of all predicates by Pred. We note that propositions p_i and p_j can belong to different processes or the same process.

Since processes in a distributed program do not share a global clock and do not necessarily execute at the same speed, in order to reason about the global state of the distributed program, we employ Lamport's logical clocks [15].

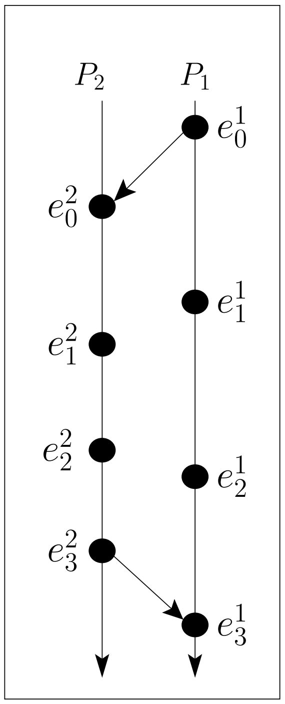
Definition 1 (Happened Before Relation) *A relation* $\rightsquigarrow \subseteq E_{\mathcal{D}} \times E_{\mathcal{D}}$ *is a* happened before *relation iff for any events* $a, b, c \in E_{\mathcal{D}}$*, the following conditions hold:*

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

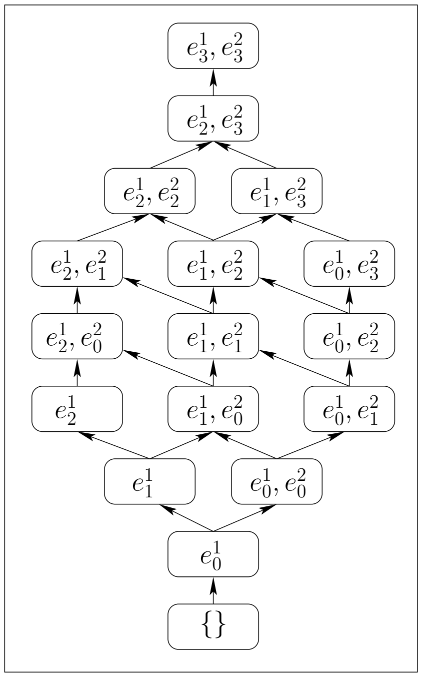
Definition 2 (Concurrent Events) *Two distinct events* $a, b \in E_{\mathcal{D}}$ *are called* concurrent events *(denoted* $a \parallel b$*) iff* $(a \not\rightsquigarrow b \wedge b \not\rightsquigarrow a)$*.* ■

For example, in the program in Fig. 2.1, we have $e_0^1 \rightsquigarrow e_2^2$ and $e_2^1 \parallel e_1^2$. A visual order of all the events in `P1` and `P2` is shown in Fig. 2.2a. Notice that one can trivially define the happened before relation for states as well.

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



(a) Visual order of events.



(b) Computation lattice.

Figure 2.2: Visual order of events and computation lattice for the program in Fig 2.1

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

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

Definition 4 (Consistent Cut [6]) *Let $\mathcal{D} = \{P_1, P_2, \ldots, P_n\}$ be a distributed program. A* cut *C is a subset of $E_{\mathcal{D}}$ and is represented by a tuple of* frontier events *$FE_C = \langle e^1_{last}, e^2_{last}, \ldots, e^n_{last} \rangle$, such that e^i_{last}, $1 \leq i \leq n$, is the last event occurred in process P_i in C. We say that C is a* consistent cut *iff* $\forall e \in C, e' \in E_{\mathcal{D}} : ((e' \rightsquigarrow e) \Rightarrow (e' \in C))$. ■

For example, in Fig. 2.2a, the cut with frontier $\langle e_1^1, e_0^2 \rangle$ is consistent, while the cut with frontier $\langle e_3^1, e_2^2 \rangle$ is not consistent. We denote the set of all consistent cuts in a computation by $\mathcal{CC}$.

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

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

Definition 6 (Computation Lattice [19]) *A* computation lattice $\mathcal{L} = (\mathcal{CC}, \rightsquigarrow)$ *is a directed graph where (1) the set of vertices is the set of all consistent cuts $\mathcal{CC}$, (2) the edge relation is the happened before relation, and (3) every pair of consistent cuts have a unique greatest common predecessor and a unique lowest common successor.* ■

Fig. 2.2b shows the computation lattice for the distributed program in Fig. 2.1. Note that each node in the lattice is the frontier of a consistent cut.

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

Definition 7 (Global-state Trace) *Given a computation lattice $\mathcal{L}$ and a (finite or infinite) path $\pi = FE_{C_0}FE_{C_1}\cdots$ in $\Pi_{\mathcal{L}}$, the global-state trace corresponding to π is the sequence $\gamma = \mathfrak{s}(\pi) = g_0g_1\cdots$, such that $\forall i \geq 0 : g_i = \mathfrak{s}(FE_{C_i})$.* ■

For example, in Fig. 2.1 if the global predicate $B = (x1 \geq 5 \wedge x2 \geq 15)$, then the sub-computation lattice $\mathcal{L}_B$ that satisfies B is a subset of the lattice in Fig. 2.2b starting from the global state (e_1^1, e_1^2) upwards.

Verifying the correctness of a distributed system with respect to a global conjunctive boolean predicate B (over the set of all variables V) and the computation lattice $\mathcal{L}$ is now reduced to checking the predicate against each consistent global state in every path $\pi \in \Pi_{\mathcal{L}}$. However, in an online setting, enumerating all global states poses a huge overhead on the distributed system.

So far, we have presented the background for global boolean predicate detection in distributed systems, next we explain the background required for detecting temporal properties such as *Linear Temporal Logic* in distributed systems.

2.2 Linear Temporal Logic (LTL) [25]

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

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

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

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

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

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

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

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

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

2.2.1 3-valued LTL

LTL semantics is defined over infinite traces and a running program can only deliver a finite trace at a monitoring point. To formalize satisfaction of LTL properties at run time, in [1], the authors propose semantics for LTL, where the evaluation of a formula ranges over three values $\{\top, \bot, ?\}$ (denoted LTL$_3$). The value '?' expresses the fact that it is not possible to decide on the satisfaction or violation of a property, given the current program finite trace.

Definition 11 (LTL$_3$ semantics) *Let $\alpha \in \Sigma^*$ be a finite trace. The valuation of an* LTL$_3$ *formula φ with respect to α, denoted by $[\alpha \models \varphi]$, is defined as follows:*

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

■

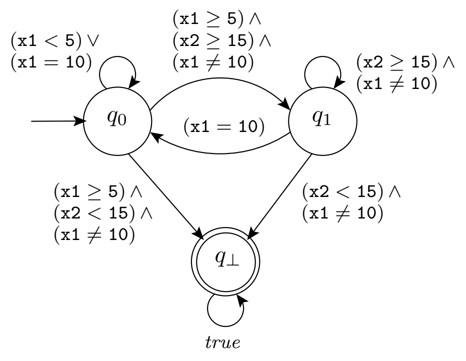


Figure 2.3: The monitor automaton for property $\psi = \Box((\texttt{x1} \geq 5) \Rightarrow ((\texttt{x2} \geq 15)\ \mathbf{U}\ (\texttt{x1} = 10)))$.

Note that the syntax '$[\alpha \models \varphi]$' for LTL$_3$ semantics is defined over finite words as opposed to '$\gamma \models \varphi$' for LTL semantics, which is defined over infinite words. For example, given a finite program trace $\alpha = g_0 g_1 \cdots g_n$, property $\Diamond p$ holds iff $g_i \models p$, for some i, $0 \leq i \leq n$. Otherwise, the property evaluates to ?.

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

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

■

For example, Fig. 2.3 shows the monitor automaton for property

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

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

There are two classes of properties often used in verification:

- Safety Properties: properties that state that something bad should never happen. Example: $\varphi = (\Box\neg p)$
- Liveness Properties: properties that state that something good should always or eventually happen. Example: $\varphi = (\Diamond p)$.

2.3 Monitoring Distributed Programs using Linear Temporal Logic

Verification of LTL$_3$ properties for distributed programs is now reduced to ensuring that none of the distributed program execution paths run into a violating state $q_\bot$ for safety properties or that all of the paths eventually run into satisfying states $q_\top$ for liveness properties. For example, in the distributed program presented in Fig.2.1 and its execution presented in Fig.2.2a, if the program is monitored for the LTL property in Fig.2.3, then the goal of a good monitoring technique is to monitor all paths in the lattice in Fig.2.2b that could lead to a final state in the automaton, namely state $q_\perp$.

Chapter 3

Formal Problem Description

Roughly speaking, given a distributed program $\mathcal{D}$ and an LTL$_3$ property φ, our goal is to construct a set $\mathcal{M}$ of *monitor processes*, such that their composition with $\mathcal{D}$ can evaluate φ at run time in a sound, complete, and decentralized fashion. However, since the occurrence of events in a distributed program does not form a total order, valuation of φ at each time instance cannot be a single value. I.e., since there are multiple possible execution paths in the computation lattice, multiple monitoring verdicts may be reached.

In order to formalize the problem, let us assume that at run-time, an *oracle* can construct the computation lattice $\mathcal{L}$, and compute the verdict for each path $\pi \in \Pi_{\mathcal{L}}$ by evaluating $\lambda(\delta(q^0, \mathfrak{s}(\pi)))$

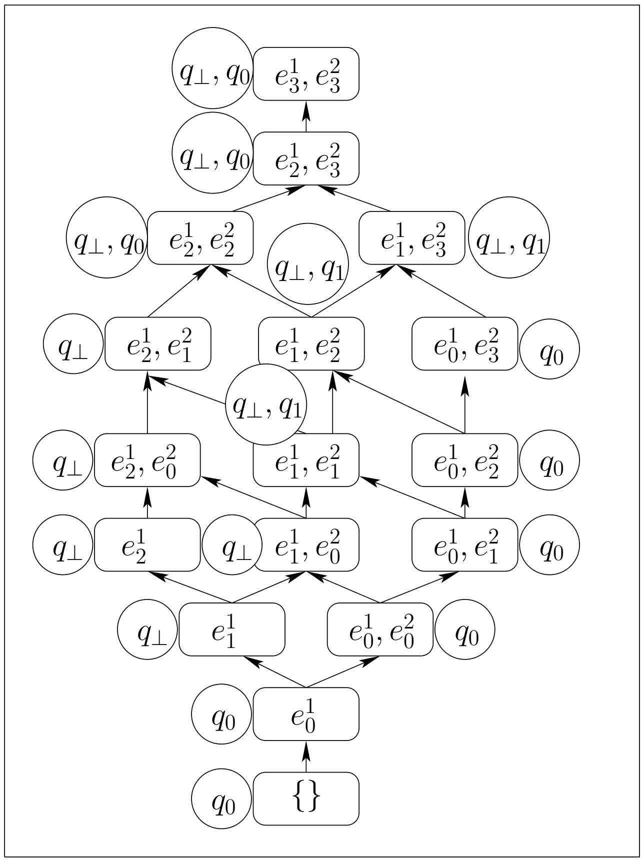


Figure 3.1: The computation lattice for the distributed program shown in Fig. 2.1 marked with the automaton state for the LTL property shown in Fig. 2.3

starting from the initial automaton state q^0. Intuitively, this means that each global state in the finite sequence of global state trace for path π is run through the automaton one by one given the automaton state for the previous global state. Fig. 3.1 shows the lattice for the distributed program shown in Fig. 2.1. Note that each global state in each path is marked with the automaton state for the LTL property

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

shown in Fig. 2.3. As can be seen, any path that goes through node $\langle e_1^1 \rangle$ will evaluate to $\bot$, while for path $\beta = \langle\{\}\rangle\langle e_0^1\rangle\langle e_0^1, e_0^2\rangle\langle e_0^1, e_1^2\rangle\langle e_0^1, e_2^2\rangle\langle e_1^1, e_2^2\rangle\langle e_1^1, e_3^2\rangle\langle e_2^1, e_3^2\rangle\langle e_3^1, e_3^2\rangle$, we have $[\beta \models \psi] = ?$. On the contrary, if we consider property

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

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

Following the above discussion, we argue that the goal of runtime verification for LTL$_3$ in a distributed system must be designing a decentralized runtime monitor, such that the set of verdicts identified by the oracle is equal to the union of the set of verdicts identified by the distributed monitors.

Decentralized LTL3 monitoring problem: Given a distributed program $\mathcal{D} = \{P_1, P_2, \ldots, P_n\}$ with an execution lattice $\mathcal{L}$, and an LTL$_3$ property φ, the goal is to identify a set of *monitor processes* $\mathcal{M} = \{M_1, M_2, \ldots, M_n\}$, such that

if each monitor process M_i

- can read the local state of P_i in one atomic step;
- can communicate with other monitor processes, and
- outputs set of path/verdict pairs Λ_i that contains a lattice path and its verdict for each lattice path $\pi \in \Pi_{\mathcal{L}}$ traversed by M_i

then

$$\forall\ \ \pi \in \Pi_{\mathcal{L}} : (\exists\ \ i, 1 \leq i \leq n : \langle \pi, [\pi \models \varphi]\rangle \in \Lambda_i) \tag{3.1}$$

and

$$\forall\ \ \langle \pi, \lambda \rangle \in \Lambda_i : (\exists\ \ \pi' \in \Pi_{\mathcal{L}} : ([\pi' \models \varphi] = \lambda) \wedge (\pi' = \pi)) \tag{3.2}$$

We note that for brevity, the notions of "communicating" and "reading the state" are not formalized, but their meaning complies with the common understanding of these concepts.

We also note that the above problem statement is for soundness and completeness. Equation 3.1 is for completeness and intuitively indicates that for each path in the lattice, there should exist at least one monitor process that can construct the trace and identify its true verdict. While Equation 3.2 is for soundness and indicates that each path/verdict pair in a monitor's verdict set should map to a path in the lattice and the oracle's verdict for that path.

Chapter 4

Monitoring Algorithm Design

4.1 Algorithm Sketch

First, we present the sketch of our algorithm that solves the problem of decentralized LTL_3 monitoring of distributed systems as stated in chapter 3.

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

1. *(Predicate detection)* Since each transition in a monitor automaton is labelled by a global-state predicate, a monitoring process has to be able to evaluate such predicates. For example, in the monitor automaton in Fig. 2.3, for the program in Fig. 2.1, if the current monitor automaton state is q_1, upon the occurrence of a local event in P_1, monitor process M_1 has to evaluate predicates that label outgoing transitions $q_1 \rightarrow q_\perp$ and $q_1 \rightarrow q_0$. For transition $q_1 \rightarrow q_0$, monitor process M_1 can verify proposition $(\texttt{x1} = \texttt{10})$ locally, hence, no need to communicate with monitor process M_2. On the contrary, for transition $q_1 \rightarrow q_\perp$, M_1 can only verify proposition $(\texttt{x1} \neq \texttt{10})$. Thus, if $(\texttt{x1} \neq \texttt{10})$ is true, M_1 has to evaluate $(\texttt{x2} < \texttt{15})$ as well, by consulting M_2.

   To address this problem we adapt the distributed computation *slicing* algorithm in [7] for distributed online detection of conjunctive predicates[1]. Computation slicing [20] is a technique to find abstract representations, called *slices*, that encode all concurrent and consecutive global states that satisfy a predicate without explicitly enumerating them. For example in Fig. 2.1, slices for predicate $(\texttt{x1} \geq \texttt{0} \wedge \texttt{x2} \neq \texttt{20})$ are represented by the set of events $\{e_1^1, e_1^2\}$ and $\{e_1^1, e_1^2, e_2^1\}$. If a slicer is distributed, then different processes may output different event-sets of the slice [7]. The technique applied in [7] detects the slice satisfying a conjunctive predicate by enumerating for the join-irreducible elements that form the sub-lattice that satisfies the predicate. We present the definitions for

   Definition 13 (Computation Slice) *[20] The slice of a computation with respect to a predicate is a sub-computation that satisfies the following properties: (a) it contains all*

[1]We emphasize that detecting only conjunctive predicates does not impose any restrictions, since transitions in LTL_3 monitor automata are only labeled by conjunctive predicates (i.e., monitor transition labeled by disjunctive predicates are handled by splitting them into multiple transitions, one per each disjunct).

global states of the computation for which the predicate evaluates to true, and (b) of all the sub-computations that satisfy condition (a), it has the least number of global states.

Definition 14 (Lattice Join/Meet) *[20] Given a computation lattice $\mathcal{L}$, and two consistent cut elements $C' \in \mathcal{L}$ and $C'' \in \mathcal{L}$, the join of C' and C'' is defined as: Join$(C', C'') = FE_{C'} \cup FE_{C''}$, while the meet of C' and C'' is defined as Meet$(C', C'') = FE_{C'} \cap FE_{C''}$.*

Definition 15 (Join-Irreducible Element) *[20] An element $C \in \mathcal{L}$ is join-irreducible if (1) C is not the smallest element of $\mathcal{L}$ (2) $\forall\, C', C'' \in \mathcal{L} : (FE_C = FE_{C'} \cup FE_{C''}) \rightarrow (C = C') \vee (C = C'')$.*

Intuitively, this means that a join-irreducible element is one that can not be represented by the join of any other two elements in the lattice. The algorithm presented in [20], finds the join-irreducible elements for a predicate thus finding the least consistent cut that satisfies that predicate. In the context of runtime monitoring using a monitor automaton, we need to detect different predicates at different times, according to the current state of the monitor automaton and the outgoing transitions from that state. Hence, after updating the current global state with a new event, each monitor attempts to find the join-irreducible elements at all processes that enables an outgoing transitions.

For example, in Fig. 2.3, if the current monitor automaton state is q_0, then both monitors M_1 and M_2 need to detect predicates $((\texttt{x1} \geq \texttt{5}) \wedge (\texttt{x2} \geq \texttt{15}) \wedge (\texttt{x1} \neq \texttt{10}))$ and $((\texttt{x1} \geq \texttt{5}) \wedge (\texttt{x2} < \texttt{15}) \wedge (\texttt{x1} \neq \texttt{10}))$. And, if the current state of the monitor is q_1, then M_1 needs to detect predicates $((\texttt{x1} \neq \texttt{10}) \wedge (\texttt{x2} < \texttt{15}))$ and $(\texttt{x1} = \texttt{10})$, while M_2 needs to detect the predicate $((\texttt{x1} \neq \texttt{10}) \wedge (\texttt{x2} < \texttt{15}))$ only. Unlike the framework of [7], in our setting, the distributed monitors might be detecting different predicate of different sizes with each event, so the communication headache differs according to the current global state. Thus, our algorithm is more sophisticated, as the set of predicates for detection changes over time.

2. *(Concurrent events)* As mentioned earlier, the existence of concurrent events results in the possibility of having multiple global-state traces or multiple lattice paths. In order to prevent exploring all possible interleavings, we only explore the global-state traces that can trigger a transition (excluding self-loops) in the monitor automaton. Each monitor process will attempt to explore all lattice paths that can enable an outgoing transition. We call this *forking*. For example, in the monitor automaton in Fig. 2.3, for the program in Fig. 2.1, events e_2^1 and e_1^2 are concurrent and as explained in Chapter 3, paths that go through these events result in different verdicts. Hence, for each such path, its corresponding verdict is stored. Note that the algorithm does not attempt to explore all possible orders of the concurrent events. It rather attempts to explore orders that enable outgoing transitions of the current state of monitor automaton. Also, note that we avoid exploring self-loops transitions since the monitor automaton state does not change when a self-loop is enabled. We emphasize that since the number of states LTL_3 monitors even for very complex properties is a handful (normally less than five), keeping all possible verdicts does not lead to any implementation issues.

3. *(Merging verdicts)* Obviously forking monitor automaton states can lead to degrading performance, since more predicates have to be evaluated. Thus, monitor processes attempt to merge monitor states, if starting from a set of possible monitor states, enabled transitions

reach the same monitor state. After merging, the maximum number of global-state traces should be bounded by the number of states in the monitor automaton. For example, in the monitor automaton in Fig. 2.3, if the current set of monitor states is $\{q_0, q_1\}$, and both outgoing transitions to $q_\perp$ are enabled, this yields to merging the two current monitor states into $\{q_\perp\}$. Also, the algorithm uses the slicing concept presented in [20] to merge global states that belong to the same slice into single global state. Thus minimizing the number of global states detected.

4.2 Algorithm Details

We assume the reader is familiar with the notion of *logical clocks* [15]. In a distributed program $\mathcal{D} = \{P_1, P_2, \cdots, P_n\}$, a vector $VC^i_k = \langle \upsilon_1, \upsilon_2, \ldots, \upsilon_n \rangle$ is the *vector clock* locally stored in process P_i, $1 \leq i \leq n$, at or after the k^{th} event occurs in P_i, and before the $(k+1)^{th}$ event occurs. υ_j is the value of the logical clock of process P_j, $1 \leq j \leq n$, as per the knowledge of process P_i. In other words, $VC^i_k[i] = k$, implies that VC^i_k is a vector clock that is aware of the occurrence of the k^{th} local event. When a process P_i sends a message to a process P_j, the vector clock of P_i is piggybacked with the message.

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

To allow a monitoring process to trace multiple execution paths, we use *Global Views*. Each global view represents a point in an execution path along with the LTL$_3$ verdict.

A monitoring process M_i has an initial global view gv^0_i that represents the initial view of the distributed system. A global view encapsulates the following data:

- Vector clock $gcut = \langle 0, 0, ..., 0 \rangle$
- Global state $g = \langle \mathfrak{s}(e^0_0), \mathfrak{s}(e^1_0), \cdots \mathfrak{s}(e^n_0) \rangle$ corresponding to the initial state of all the processes.
- Initial monitor automaton state q_0
- An initially empty pending events queue: p_events
- An object used by global views to collect information about other monitor processes, initially set to an empty object.
- An initial unblocked state
- A boolean variable: $KeepAfterFork$ to indicate whether the global view will be valid after forking another global view or not.

A global view has three possible states: **blocked, waiting and unblocked**. A blocked global view is waiting for events to happen at other monitoring processes. An unblocked global view is ready to receive new events from the monitoring process. A waiting global view is waiting for a specific event to happen at P_i.

Monitor processes communicate by exchanging global view's *tokens*, we use the term $token$ and messages interchangeably to indicate the monitor communication messages. A $token$ has the following data:

- $NextTargetProcess$: represents the process that the token should be sent to next.
- $NextTargetEvent$: represents the event sequence number that the token should evaluate when received at $M_{NextTargetProcess}$.
- $ParentProcess$: represents the process that created the token.
- $ParentEventVC$: represents the last event that was processed by the global view owning the token.
- $OutgoingTransitions$: a set of the possibly enabled outgoing transitions the global view is searching for.

A single entry in $OutgoingTransitions$: contains the following data:

- $TransitionId$ represents the id of the transition.
- $gcut$: represents the vector clock of the constructed global state thus far.
- $depend$: represents the vector clock used to check for inconsistencies in the constructed global state.
- $ConjunctsEvaluations$: a vector of n entries representing a mapping between the conjuncts in the transition's predicate and their boolean evaluation with respect to the constructed global state. Each evaluation entry can take values of $predtrue$ or $predfalse$ or $unset$. If a process has multiple conjuncts in a predicate, the conjuncts will occupy a single entry in the vector. For example: $[\langle\langle a \wedge b\rangle, predfalse\rangle, \langle\langle c\rangle, predfalse\rangle]$ is a $ConjunctsEvaluation$ for a distributed system with 2 processes, and a predicate $a \wedge b \wedge c$ where the first process has propositions a and b, while the second process has proposition c.
- $NextTargetProcess$: represents the process that the transition should be sent to next.
- $NextTargetEvent$: represents the event sequence number that the transition should be sent at next.
- $gstate$: represents the global state corresponding to $gcut$
- $eval$: represents the state of the transition (evaluated to true, false or not evaluated)

Next, we give a brief overview description of the main procedures that constitute this algorithm, and then we provide a fine-grained analysis for each procedure.

In subsubsection 4.2.0.1 the main monitor loop that receives events and messages from other processes is described.

In subsubsection 4.2.0.2 we show the monitor INIT procedure that is responsible for the monitor initialization process and defining the inputs and initial variables that the monitor process maintains.

In subsubsection 4.2.0.3 we show the RECEIVE EVENT procedure which is the main interface procedure between the program process and the monitor process. When a program generates an event (internal or receive), the program increments the i^{th} entry in its vector clock and sends the event details along with the vector clock to the monitor using the RECEIVE EVENT procedure.

In subsubsection 4.2.0.4, the procedure PROCESS EVENT is described. After receiving the event, RECEIVE EVENT invokes PROCESS EVENT for each *Global View* that is active and waiting for further events to advance its current monitor automaton state.

In subsubsection 4.2.0.5, CHECKOUTGOINGTRANSITIONS procedure is described. When PROCESSEVENT procedure decides that the global view it is processing needs to consult other monitor processes to advance their current monitor automaton state, it invokes CHECKOUTGOINGTRANSITIONS to prepare the message (token) that need to be sent to other monitor processes for all transitions that could possibly be enabled for the newly received event.

In subsubsection 4.2.0.6, procedure SENDTONEXTPROCESS is described. This procedure is responsible for routing the tokens between monitor processes. Since a monitoring message can contain multiple transitions to satisfy, the procedure attempts to route the message in an efficient manner to provide equal opportunities for visiting all monitor process required by all transitions.

In subsubsection 4.2.0.7, procedure RECEIVETOKEN is described. RECEIVETOKEN is responsible for receiving tokens (monitoring messages) from other monitor processes. If the received token's parent is the current monitor process (a monitoring message returning to its parent), then the procedure checks the transitions evaluation (enabled, disabled, inconsistent) and updates the token's parent global view. If the current monitor process is not the token's parent, then the event requested by the token is extracted from the events history and the transitions in the token that requested this event are updated with the requested event.

In subsubsection 4.2.0.8, procedure PROCESSTOKEN is described. PROCESSTOKEN is responsible for updating and routing tokens received from other monitor processes.

In subsubsection 4.2.0.9, procedure EVALUATETOKEN is described, which can be considered a helper procedure for PROCESSTOKEN, since its purpose is to evaluate a transition predicate in a token.

In subsubsection 4.2.0.10, procedure TERMINATE is described, which is responsible for preparing the monitor for termination as the program process has terminated.

4.2.0.1 ALGORITHM MAIN LOOP – Algorithm 1

The algorithm inputs are specified in the inputs for Algorithm 1, namely the following:

- Automaton $\mathcal{A}^{\varphi}$ that encapsulates all the states and transitions.
- Initial global state $init\_gstate$ that is simply the initial propositions valuation from all processes.
- Number of program processes participating in the distributed system: n.
- Current monitor process index: $i : 0 < i < n$.

In line 1 procedure INIT is invoked. Then in Lines 3–7, M_i receives token messages from other monitor processes (respectively, reads local events of P_i) in an infinite loop in a non-blocking fashion and dispatches them to RECEIVETOKEN (respectively, RECEIVEEVENT). We

Algorithm 1 Monitor Process M_i initialization

```
Input: Monitor automaton $\mathcal{A}^\varphi = (Pred, Q, q^0, \delta, \lambda)$ and initial global state $init\_gstate$, process index $i$ and number
    of processes $n$.
1:  INIT();
2:  while 1 do
3:      $m \leftarrow$recv() end if
4:      if $m$ is not empty then RECEIVETOKEN($m$); end if
5:      $e \leftarrow$ read()
6:      if $m$ is termination event then TERMINATE();end if
7:      if $e$ is not empty then RECEIVEEVENT($e$); end if
8:  end while
    ------------------------------------------------------------
9:  procedure INIT
10:     $history \leftarrow \langle\rangle$; $w\_tokens \leftarrow \{\}$;
11:     $gv_0.gstate \leftarrow init\_gstate$; $gv_0.q \leftarrow \delta(q^0, gv_0.gstate)$;
12:     $gv_0.token \leftarrow \{\}$; $GV \leftarrow \{gv_0\}$;
13:     let $e_0$ be an event representing the local initial state
14:     PROCESSEVENT($gv_0, e_0$)
15: end procedure
```

note that if the read event is a termination signal from the program to notify M_i that P_i has terminated execution, the monitor invokes procedure TERMINATE.

4.2.0.2 INIT

Procedure INIT is responsible for initializing the following data structures for M_i:

- $history$, a vector representing the sequence of local events that occurred in P_i, is initialized to the empty sequence.
- w_tokens, a set of the tokens waiting for processing at M_i initialized to the empty set. These tokens were received from other monitor processes and are targeting some future events at P_i to collect information about the local state of P_i.
- The initial global view gv_0 pointing to the initial global state and initial automaton state q_0.

Then in Line 11 the next state of the monitor automaton (if exists) is computed by applying the initial global view gv_0 on the transition relation δ from initial monitor automaton state q^0. In Line 12, $gv_0.token$ is initialized to an empty set, and gv_0 is added to the set of all global views GV.

In Lines 13 –14, the initial event is extracted from the initial input global state and passed to procedure PROCESS EVENT, so that if there is an enabled transition from the initial monitor automaton state q_0, the current monitor process M_i would be able to detect it.

4.2.0.3 RECEIVE EVENT **(Event e^i_k) – Algorithm 2**

The goal of this procedure is to process local events only for global views whose pending events queue is empty.

Algorithm 2 Local Event Handler in M_i

```
 1: procedure ReceiveEvent(event $e$)
 2:   MergeSimilarGlobalViews()
 3:   $history[e.sn] \leftarrow e$;
 4:   for each $t \in w\_tokens$ do
 5:     if $t.NextTargetEvent = e.sn$ then
 6:       ProcessToken($t, e$); $w\_tokens \leftarrow w\_tokens \setminus \{t\}$;
 7:     end if
 8:   end for
 9:   for each $gv \in GV$ do
10:     $gv.p\_events.enqueue(e)$;
11:     if $gv.state = unblocked \vee (gv.state = waiting \wedge gv.token.NextTargetEvent = gv.p\_events[0].sn)$
  then
12:       $e' \leftarrow gv.p\_events.dequeue()$;
13:       ProcessEvent($gv, e'$);
14:     end if
15:   end for
16: end procedure
```

```
17: procedure ProcessEvent($GV$ $gv$, event $e$)
18:   $gv.gcut[i] \leftarrow e.VC[i]$; $gv.gstate[i] \leftarrow \mathfrak{s}(e)$;
19:   $gv.keepAfterFork \leftarrow false$
20:   $isConsistent \leftarrow (\nexists j, 0 < j \leq n : gv.gcut[j] < e.VC[j])$
21:   if $isConsistent \wedge (\exists\ EnabledOutgoingTransition \vee \exists\ EnabledSelfLoopTransition)$ then
22:     ClearTokens($gv.token$)
23:     $q_n \leftarrow gv.q \leftarrow \delta(gv.q, gv.gstate)$;
24:     $gv.keepAfterFork \leftarrow true$
25:     if $\lambda(q_n) \in \{\perp, \top\}$ then Declare "satisfaction/violation"; end if
26:   end if
27:   if $gv.token \neq \{\}$ then
28:     UpdateToken($gv.token, e$)
29:   end if
30:   if $gv.token \neq \{\}$ then
31:     ProcessToken($gv.token, e$)
32:   else
33:     if $gv.keepAfterFork = true$ then
34:       $gv.keepAfterFork = false$
35:       $GV \leftarrow GV \cup \{$ CopyGlobalView($gv$)$\}$
36:     end if
37:     CheckOutgoingTransitions($gv, e$);
38:   end if
39: end procedure
```

In Line 2, M_i invokes procedure MergeSimilarGlobalViews that is responsible for merging global views that have the same monitor automaton state and belong to the same slice into one global view.

In Line 3, M_i adds the k^{th} event to its history. Then, in Lines 4 – 8, M_i invokes ProcessToken for each $token$ in w_tokens, where $token.NextTargetEvent = event.sn$.

Then each global view gv in the set of all global views GV enqueues the event to its pending events queue in Line 10. Then In Line 11 the monitoring process calls ProcessEvent on global views that are unblocked or waiting for the next local event.

4.2.0.4 PROCESS EVENT (**GlobalView gv, Event e^i_k) – Algorithm 2**

In Line 18, the monitoring process updates the global view gv to include the new event by updating the i^{th} index in $gv.gcut$ vector with the i^{th} index in $e^i_k.VC$ and updates the i^{th} state in $gv.gstate$ with the state of the received event $\mathfrak{s}(e^i_k)$.

Then in Line 20, if the $gv.gcut$ vector is consistent with the new event (i,e, $\nexists j : gv.gcut[j] < e^i_k.VC[j]$), the monitoring process will start examining possible transitions on $\mathcal{A}^\alpha$ given the current monitor automaton state $gv.q$. The following cases are possible:

- There exist an enabled outgoing transition.
- There exist an enabled self-loop only.
- There exist an enabled self-loop and at least one possibly enabled outgoing transition.

Self-loops transitions are transitions whose source and target automaton states are the same. Outgoing transitions are transitions whose source and target automaton states are different. Enabled transitions are transitions whose label predicate is satisfied by the current consistent global state. Possibly enabled transitions are transitions where at least the local state of the process P_i satisfies the conjuncts with P_i's propositions in the label predicate. Hence, there might be a consistent global state that satisfies the whole predicate.

Since $\mathcal{A}^\alpha$ is a deterministic automaton, a single global state can enable only one transition. However, due to concurrency, there might be more than one possible consistent global states. To avoid state space explosion, each monitor process attempts to only explore global states that can result in enabling an automaton transition. In Line 19, $gv.keepAfterFork$ is initialized to $false$.

In Line 20, the global view is evaluated for consistency by comparing the $gv.gcut$ and $e.VC$, if $e.VC$ has any entry bigger than $gv.gcut$ then the global view is inconsistent since the event knows more information about other processes that gv is unaware of.

In Lines 21–25 if $gv.gcut$ is consistent with the new event, and there exist an enabled outgoing or self-loop transitions, then $gv.q$ is advanced to the new automaton state. If the new automaton state is a final state, then the monitor process declares satisfaction/violation. $gv.keepAfterFork$ is set to $true$ to indicate that this global view is valid, and should be kept even if forked.

If the global view is inconsistent or consistent with no enabled transitions, then the global view would maintain $gv.keepAfterFork = false$ so that it would be deleted after forking to a new global state. Since gv is out of sync and does not represent a global state that can occur in realtime.

In Line 27, if the global view has a token, then it is in the waiting state (i.e. waiting for local events to update the token's outgoing transitions state). Procedure UPDATETOKEN is invoked to update the outgoing transitions in $gv.token$ with the new local event. UPDATETOKEN could result in no action if the new event's $gstate$ is identical to the event that created the token (i.e. same set of possibly enabled transitions), or could result in removing some transitions or adding new ones. In Line 30, since UPDATETOKEN could result in deleting all transitions without adding any new ones, $gv.token$ is checked again and if it is not empty then PROCESSTOKEN is invoked to update the remaining transitions with the new event and evaluate the state of the global view.

In Lines 33–36, if $gv.keepAfterFork$ was set to true, forking occurs by creating a new copy of gv and adding it to the set of all global views GV after setting $gv.keepAfterFork$ to false.

CHECKOUTGOINGTRANSITIONS is invoked in Line 37 to search for possible outgoing transitions.

4.2.0.5 CHECKOUTGOINGTRANSITIONS (GlobalView gv, Event e_k^i) – Algorithm 3

Procedure CHECKOUTGOINGTRANSITIONS is responsible for creating the $token$ object for a global view. $gv.token$ holds all the data needed to search for consistent global states that proceed gv. To achieve this purpose, the procedure searches for transitions whose label predicate can be satisfied by the current local state of event e_k^i. In other words, transitions which the state of process P_i is not forbidding the predicate from becoming true. Note that a forbidding process can either be a process whose known state does not satisfy the predicate or a process whose state is inconsistent with gv.

In Line 2, the $ParentProcess$ and $ParentEventVC$ variables are set to the current process and the vector clock of the event $e_k^i.VC$ respectively, the rest of the data in $gv.token$ will be set at the end of the procedure.

In Line 4, the procedure iterates over all transitions whose source is $gv.q$ and target is not $gv.q$ (not a self-loop).

Then in Line 6, the set of forbidding processes for $pred$ is computed and if P_i is not an element in this set (i.e. local state of P_i satisfies the transition partially), an empty $OutgoingTransition$ object is initialized with the transition id.

In Lines 8 –10, each index j in the $depend$ and $gcut$ vectors is updated with the maximum of $gv.gcut[j]$ and $e_k^i[j]$ to ensure that the token would search process P_j's local history for events that gv is not already aware of yet.

Then in Line 11, each conjunct in $pred$ is evaluated against $gv.gstate$ and then the conjuncts along with the evaluations are set into $OutgoingTransition.ConjunctsEvaluation$.

In Line 12, the next process which this transition should visit is set to the forbidding process with the smallest index. Note that this is the next target process for this $OutgoingTransition$ only, not for $gv.token$ and that procedure SENDTONEXTPROCESS has optimized logic that routes the token to processes in a manner that attempts to minimize visits between processes.

In Line 13, the event which the transition should check at the $NextTargetProcess$ is set to be the max of $gv.gcut[NextTargetProcess] + 1$ and $e_k^i.VC[NextTargetProcess]$, the reason for using the max is to make sure that the $OutgoingTransition$ will not target event that has already been consumed.

And then the $OutgoingTransition$ is added to $gv.token.OutgoingTransitions$. Finally, in Line 17 the procedure SENDTONEXTPROCESS is invoked to decide which process and which event the token as a whole should be sent to.

4.2.0.6 SENDTONEXTPROCESS (*token t*)

The purpose of SENDTONEXTPROCESS is to route the tokens between monitor processes with the objective of evaluating all $OutgoingTransition$ in t to either enabled or disabled. SENDTONEXTPROCESS is responsible for setting $t.NextTargetProcess$ and $t.NextTargetEvent$.

Algorithm 3 Token Handler in M_i (1)

```
procedure CheckOutgonigTransitions($GlobalView\ gv$, $event\ e^i_k$)
  $gv.token.ParentProcess \leftarrow i$
  $gv.token.ParentEventVC \leftarrow e^i_k.VC$
  for each $q'$, $pred$ st. $((gv.q \neq q')) \wedge (gv.q \xrightarrow{pred} q') \in \delta$ do
    $transition \leftarrow gv.q \xrightarrow{pred} q'$;
    $forbidding \leftarrow$getForbiddingProcesses$(gv, q')$
    if $P_i \notin forbidding$ then
      let $outgoingTransition$ be an empty transition token;
      $outgoingTransition.transition \leftarrow transition$
      $outgoingTransition.gcut \leftarrow outgoingTransition.depend \leftarrow max(gv.gcut, e^i_k.VC)$
      $outgoingTransition.ConjunctsEvaluation \leftarrow$ GetConjunctsWithEvaluation$(pred, gv.gstate)$
      $outgoingTransition.NextTargetProcess \leftarrow forbidding[0]$
      $outgoingTransition.NextTargetEvent \leftarrow max(gv.gcut[forbidding[0]] + 1, e^i_k.VC[forbidding[0]])$
      $gv.token.OutgoingTransitions \leftarrow gv.token.OutgoingTransitions \cup \{outgoingTransition\}$;
    end if
  end for
  SendToNextProcess$(gv.token)$; $gv.token.sent \leftarrow true$;
end procedure
```

```
procedure receiveToken($token\ t$)
  if $(t.ParentProcess) = i$ then
    let $gv$ be the owner global view for $t$
    for each $OutgoingTransition\ tran \in t.OutgoingTransitions$ do
      if $tran.eval = predtrue$ then
        spawnNewGlobalView$(tran, gv)$
        $t.OutgoingTransitions \leftarrow t.OutgoingTransitions \setminus \{tran\}$
      else if $tran.eval = predfalse$ then
        $t.OutgoingTransitions \leftarrow t.OutgoingTransitions \setminus \{tran\}$
      else if $\exists k, 1 < k \leq n : tran.gcut[k] < tran.depend[k]$ then
        $tran.NextTargetEvent \leftarrow tran.gcut[k] + 1$
        $tran.NextTargetProcess \leftarrow P_k$
      end if
    end for
    if $t.OutgoingTransitions = \{\}$ then
      if $gv.keepAfterFork = false$ then
        $GV \leftarrow GV \setminus \{gv\}$
      end if
    else // there still exist tokens with pending evaluation
      SendToNextProcess$(t)$
    end if
  else // $M_i$ is not the parent of the token
    $tokenReturned \leftarrow false$
    while $\exists f \in history : f.sn = t.NextTargetEvent$ do
      //required event has happened
      $tokenReturned \leftarrow$ProcessToken$(t, f)$
      if $tokenReturned$ then break; end if
    end while
    if $tokenReturned = false$ then
      $w\_tokens \leftarrow w\_tokens \cup t$
    end if
  end if
  MergeSimilarGlobalViews()
end procedure
```

The procedure will try to find a transition in $t.OutgoingTransitions$ that can satisfy any of the following ordered rules, if no transition is found that can satisfy the first three rules, the procedure falls back to the last rule and returns the token back to its parent.

1. A transition with all conjuncts evaluated to true (i.e. enabled).
2. A transition targeting the current process.
3. A transition targeting a process other than the parent of the $token$ and the current process.
4. If all of the previous rules are not satisfied, the token is sent back to the parent process.

After setting $t.NextTargetProcess$ and $t.NextTargetEvent$, the current monitoring process M_i sends t to $M_{t.NextTargetEvent}$. We note that SENDTONEXTPROCESS presents a heuristic algorithm for routing tokens, and while routing each $OutgoingTransition$ in t individually would guarantee faster inspecting for each outgoing transition, it adds a significant communication overhead between monitor processes. Therefore, we decided to route transitions in bulk to decrease communication overhead.

4.2.0.7 RECEIVETOKEN ($token\ t$) – Algorithm 3

When a monitoring process M_i receives a $token$ object from M_j, it first checks if P_i is the parent of the received token.

If P_i is the parent of t – Line 20:

t was created earlier by P_i to search for global states that can satisfy some transitions. When P_i receives back the token, in Line 22, it iterates over all the transitions with the following rules:

- In Line 23, if the transition was evaluated to $predtrue$, then a new global view is created (copied from the parent global view that created t) with the target monitor state of the transition, and then the global state and global state cut are updated with $tran.gstate$ and $tran.gcut$ respectively. If the new monitor state is a violation or satisfaction state, then the program is notified. Then the transition is deleted from the set of outgoing transitions in t.
- In Line 26, if the transition was evaluated to $predfalse$, then the transition is deleted from the set of outgoing transitions in t.
- In Line 28, if the transition is inconsistent which can occur when $tran$ has collected one or more recent events than the parent global view gv, then $tran.NextTargetEvent$ and $tran.NextTargetProcess$ are set to target the inconsistent process, which might be P_i itself.

In Line 35, if after applying the previous rules, the set of outgoing transitions in t is empty and gv was marked to be deleted after forking, then gv will be removed from the set of all global views. If the set of outgoing transitions in t is not empty, then SENDTONEXTPROCESS is invoked to attempt to send the token to the next process that can satisfy the remaining transitions. Note that SENDTONEXTPROCESS can determine that t should remain at P_i and then the gv will be in the waiting state.

If P_i is not the parent of t – Line 40:

Algorithm 4 Token Handler in M_i (2)

```
procedure AddEventToToken($OutgoingTransition\ tran$, $event\ e$)
    $tran.gstate[i] \leftarrow \mathfrak{s}(e)$
    $tran.gcut[i] \leftarrow e.sn$
    $tran.depend \leftarrow max(tran.depend, e.VC)$
end procedure
------------------------------------------------------------
procedure ProcessToken($token\ t$, $event\ e$)
    for each $OutgoingTransition\ tran \in t.OutgoingTransitions$ do
        if $tran.NextTargetProcess = i \wedge tran.NextTargetEvent = e.sn$ then
            AddEventToToken($tran, e$)
        end if
    end for
    EvaluateToken($t, e$)
    for each $OutgoingTransition\ tran \in t.OutgoingTransitions$ do
        if $tran.eval = predtrue \vee tran.eval = predfals$ then
            $tran.NextTargetProcess \leftarrow t.ParentProcess$
            if $t.ParentProcess = i$ then HandleCompletedTransition($tran$); end if
        else if $tran.gcut$ is inconsistent then
            /*find k: $tran.gcut[k] < tran.depend[k]$*/
            $tran.NextTargetEvent \leftarrow tran.gcut[k] + 1$
            $tran.NextTargetProcess \leftarrow P_k$
        else // transition evaluation incomplete, need to visit other processes
            /*find k: $tran.ConjunctsEvaluation[k] = unset$*/
            $tran.NextTargetEvent \leftarrow tran.gcut[k] + 1$
            $tran.NextTargetProcess \leftarrow P_k$
        end if
    end for
     return SendToNextProcess($t$)
end procedure
```

t was sent to M_i by $t.ParentProcess$ looking for the earliest event e^i_k that can satisfy at least one of the outgoing transitions in $t.OutgoingTransitions$. In Lines 42–46, the monitor loops through the history events invoking ProcessToken for each event starting with $t.NexrTargetEvent$ until the token is routed to another monitor process or the requested event has not yet occurred. ProcessToken returns $true$ if t was sent to another process. In Line 47, if the event has not yet occurred then t is added to the set of waiting tokens.

Finally before the procedure terminates, in Line 51 M_i invokes procedure MergeSimilarGlobalViews that is responsible for merging global views that has same monitor automaton state and belong to the same slice into one global view.

4.2.0.8 ProcessToken ($token\ t$, Event e^i_k) – Algorithm 4

ProcessToken purpose is applying and evaluating the local state of e^i_k to the transitions in t that have both (1) P_i as its $NextTargetProcess$ and (2) e^i_k as its $NextTargetEvent$.

In Line 9, M_i invokes AddEventToToken for the transitions that satisfy the above two rules.

In Lines 2 –3 ,AddEventToToken updates the transition's $gcut$ and $gstate$ with the new event. In Line 4 it also updates each index k in the transition's $depend$ vector with the maximum of $tran.depend[k]$ and $e.VC[k]$. The $depend$ vector helps identify inconsistent processes in the cut.

Afterwards in Line 12, PROCESSTOKEN invokes EVALUATETOKEN which updates the $ConjunctsEvaluations$ for each transition in t that has both (1) P_i as its $NextTargetProcess$ and (2) e_k^i as its $NextTargetEvent$.

Afterwards, in Line 16 if any transition was evaluated to either $predtrue$ or $predfalse$ then the transition's $NextTargetProcess$ will be set to $t.ParentProcess$. In Line 16, if the current process is the parent of t, then procedure HANDLECOMPLETEDTRANSITION is invoked to either spawn a new global view at the new discovered automaton state (if the transition is enabled) or to delete the transition (if the transition is disabled). Else if the transition's $eval$ is set to $unset$, then it is either because the transition is inconsistent and/or incomplete (i.e, some conjuncts have not been evaluated yet).

In Lines 17 –20 if the transitions is inconsistent, then transition's $NextTargetProcess$ will be set to the first inconsistent process found in the $gcut$ vector clock.

In Lines 22 –24, if the transition is incomplete, then the transition's $NextTargetProcess$ will be set to the first process that has an incomplete conjunct in the $ConjunctsEvaluations$ vector. After all transitions have been processed, M_i invokes SENDTONEXTPROCESS which examines the new $NextTargetProcess$ for each transition and attempts to find the next process to send t to in order to minimize messages overhead and detection latency. SENDTONEXTPROCESS returns true if the token was sent to another process and false otherwise. We note that the strategy used in SENDTONEXTPROCESS is best effort in the sense that it attempts first to return tokens that have at least one transition evaluated to $predtrue$ to avoid detection latency. However, in theory there might be an incomplete transition that would lead to a violation state and better be checked first. As an optimization, SENDTONEXTPROCESS can be tuned to prefer processing transitions with certain target monitor automaton states first.

4.2.0.9 EVALUATETOKEN($token\ t$, Event e) – Algorithm 5

The purpose of this procedure is to set the conjuncts evaluation in each transition in t against the global state constructed by the transition so far and also set the transition's $eval$ if possible.

In Line 4, M_i enumerates all transitions that satisfies all of the following:

- P_i participates in
- has P_i as its $NextTargetProcess$
- has e as its $NextTargetEvent$
- have not already been evaluated

For each enumerated transition, M_i checks if the conjunct(s) for P_i are satisfied by the transitions' $gstate$.

In Line 6, if the conjuncts(s) are satisfied then the i^{th} entry in $ConjunctsEvaluation$ will be set to $predtrue$. If not satisfied, in Line 9 the i^{th} entry in $ConjunctsEvaluation$ will be set to $predfalse$.

Algorithm 5 Token Handler in M_i (2)

```
procedure EvaluateToken(token $t$, event $e$)
    $flag \leftarrow false$
    for each transition $tran \in t.OutgoingTransitions$ do
        if ($P_i$ participates in $tran$) $\wedge$ ($tran.NextTargetProcess = P_i$) $\wedge$ ($tran.NextTargetEvent = e.sn$) $\wedge$ ($tran.eval = unset$) then
            if EVAL($tran.ConjunctsEvaluations[i], tran.gstate[i]$) then
                $tran.ConjunctsEvaluations[i] \leftarrow predtrue$
                $flag \leftarrow true$
            else
                $tran.ConjunctsEvaluations[i] \leftarrow predfalse$
            end if
        end if
    end for
    for each transition $tran \in t.OutgoingTransitions$ do
        if ($P_i$ participates in $tran$) $\wedge$ ($tran.NextTargetProcess = P_i$) $\wedge$ ($tran.NextTargetEvent = e.sn$) $\wedge$ ($tran.eval = unset$) then
            if $flag = true$ then //at least one transition evaluated true for ConjunctsEvaluations[i]
                if $tran.ConjunctsEvaluations[i] = predfalse$ then
                    $tran.eval \leftarrow predfalse$
                else if $\forall k, 0 < k \leq n : tran.ConjunctsEvaluations[k] = predtrue$ then
                    $tran.eval \leftarrow predtrue$
                end if
            else
                $tran.ConjunctsEvaluations[i] = unset$
                $tran.eval = unset$
            end if
        end if
    end for
end procedure
```

In Line 15 If at least one transition had $predtrue$ for P_i's conjuncts, then in Line 17 all the transitions that had P_i's conjuncts evaluated to $predfalse$ will have $eval$ set to $predfalse$ as well. This is to ensure the order of events remains consistent, since if M_i was to enable a transition based on event e_k^j, it should not enable another transition based on e_m^j where $m \geq k$.

For example, consider the distributed computation in Fig 4.1, when the tokens object created by M_1 - after the occurrence of event e_1^1 - is checking both outgoing transitions at P_2, it will first encounter event e_1^2 which would enable the transition to q_1. If the other transitions to q_2 is not disabled, then when event e_2^2 occurs both transitions would be enabled at the same time. Which is not correct when tracing the left most path in the lattice (that starts by advancing event e_1^1).

In Line 19, if all the conjuncts in the enumerated transition are evaluated to $predtrue$, then the transition's $eval$ will be set to $predtrue$. If the state of event e did not satisfy any transitions's conjuncts, then the evaluation of the conjuncts in the enumerated transitions will be set to $unset$ instead of $predfalse$ to allow it to be processed for next events.

4.2.0.10 Terminate

Procedure Terminate is responsible for preparing the monitor process to termination, it kicks off the following operations:

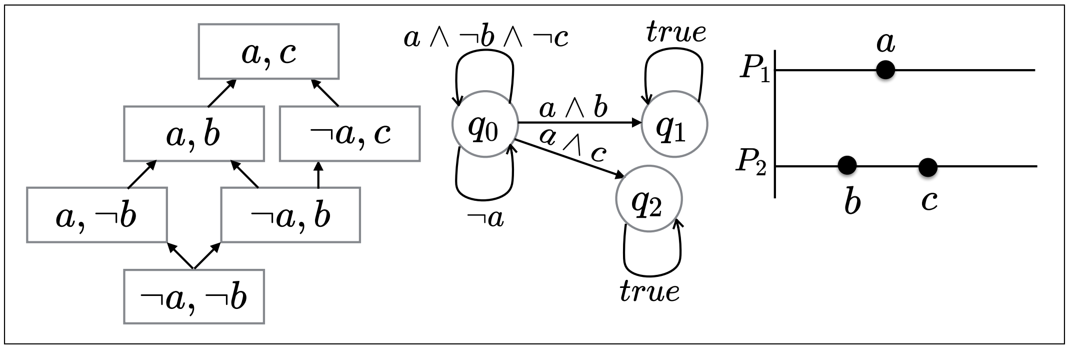


Figure 4.1: A computation, automaton and lattice for a distributed system with two processes and an initial global states with all propositions set to $false$.

- Return all the waiting tokens in the set w_tokens that are waiting for events beyond the last event in the $history$ vector.
- Inform other monitor processes that P_i has terminated and will not be producing events beyond the last event in the $history$ vector.
- Receiving program and monitor processes termination signals from other monitor processes.
- Sending program and monitor processes termination signals to other monitor processes.
- Terminating the monitor if all other monitors have sent termination signals.

4.3 Monitor Algorithm Optimizations

In this section, we describe some optimizations that we applied to the proposed algorithm to decrease the monitor communication overhead and monitor detection latency.

4.3.1 Aggregating token messages

Since a monitor process can have multiple global views sending and receiving token messages, we attempt to send messages to other monitoring processes after all global views have finished processing procedure PROCESSEVENT, and aggregate messages targeting the same monitor process to decrease the communication overhead between monitor processes.

4.3.2 Avoiding duplicate global views

When a new event enables a self-loop transition for a global view and has the same set of possible outgoing transitions as a previous event, we avoid both forking a new global view and invoking CHECKOUTGOINGTRANSITIONS if there already exists a global view forked for checking the same set of possibly enabled outgoing transitions for the previous event(s). This is inspired by the slicing technique [20], since the new event is considered to be an element in the slice being constructed by the preceding global view.

4.3.3 Avoid checking disjunctive transitions

Due to the design of our algorithm and its dependency on predicate detection, we were forced to split transitions with disjunctive predicates into multiple transitions with conjunctive predicates only. This resulted in possibly two automaton states having more than one transition between them. Which could cause a global view to exert more effort trying to satisfy all these transitions although they lead to the same automaton state, while in practice only satisfying one transition suffices. Therefore, whenever a transition is enabled, we delete pending transitions that have the same automaton state destination.

4.4 Monitoring Algorithm Analysis

In this section, we show the complexity analysis of the monitoring algorithm presented in section 4.

4.4.1 Monitoring Messages Overhead

When a monitoring message (token) is created by a global view to check the satisfaction of some transitions, the monitoring processes are responsible for routing the token until all transitions are either enabled or disabled. We argue that the routing complexity which we measure by the maximum number of visits a token makes (until it returns to the parent, and then either gets deleted or the token's parent event is changed) depends on the rate of inconsistencies that occur between processes. This can be inferred from the rules with which SENDTONEXTPROCESS procedure operates. The procedure first tries to return the token to its parent if any transition in the token is enabled. If not, it attempts to keep the token at the current monitor process if any transition is targeting it. Then finally if the previous two conditions are not met, it sends the token to a process targeted by at least one transition. If none is found, the token is returned to its parent. This indicates that in the worst case happens when a token bounces between monitoring processes (before returning to its parent) is $\mathcal{O}(E)$ times, where E is the total number of events in the distributed systems.

This could happen if the token has a transition that is targeting a process that is repeatedly inconsistent with other process(es) but not the parent (since if a token revisits the parent for inconsistencies, the parent event is updated and hence the token should be calculated as overhead for the new event), thus the token keeps jumping between the inconsistent processes until either the transition is satisfied or the parent is inconsistent, and finally the token returns the parent. Therefore, it is expected that as the communication frequency decreases, and consequently the inconsistency between processes decreases, the number of messages exchanged should decrease.

The number of tokens created at a monitoring process depends on the number of global views created. The latter depends on the communication pattern between the program processes (frequency of inconsistencies) and on the trace produced by the program. Whether the program generates events that cause the monitor process to stay at the same automaton state, or jumps between states. Therefore, the best case for the number of global views created per monitor process is $\mathcal{O}(1)$, while the worst case is $\mathcal{O}(E)$. However, we note that the final number of global views would be much less than that since the procedure MERGESIMILARGLOBALVIEWS would eventually merge global views with the same automaton state. Therefore, the final number of global views is bounded by the number of automaton states in the automaton.

4.4.2 Memory Overhead

Memory overhead in the algorithm can be attributed to two things: (1) the growing events list maintained by the algorithm, (2) the global views created.

4.4.2.1 Events List

An event e can easily be removed from the event list at a monitoring process M_i, if an extra type of communication between monitors is introduced to ensure all global views at all monitor nodes have $gcut[i]$ greater than $e.VC[i]$. We do not currently implement this optimization. Therefore, the size of the events list at each monitoring process is bounded by the number of events generated by the program process.

4.4.2.2 Global Views

As noted in the previous subsection, the number of global views created depends mainly on the program execution. However, as the number of program processes increase, the number of global views should not be affected. This is due to the fact that global views are created for events and their tokens are routed between processes.

4.5 Proof of Correctness

In order to proof the correctness of our proposed technique, we need to proof that the algorithm is deadlock-free, sound and complete.

4.5.1 Deadlock-Freedom

In this section, we present the proof of the deadlock freedom property of Algorithm 1.

Lemma 1 *A token t created by M_i and sent out to other processes, will eventually return to M_i in finite time.*

Proof 1 Let's assume that t never returns to M_i, then it is either waiting indefinitely at some monitor process M_j or it is in a continuous send/receive state between other processes.

For a token to wait indefinitely at some process, this implies that $t.NextTargetEvent$ is equal to a number greater than the sequence number of the last event at P_j. This is a contradiction, since in procedure TERMINATE when the P_j sends the termination event to M_j, the latter sets all its conjuncts evaluation of the waiting tokens to false and sends them out to the next process or their parent (depending on the transitions state). And since we assume that all our processes are terminating processes, then returning all waiting tokens is bound to happen.

According to procedure SENDTONEXTPROCESS, t is sent out to a monitor process M_i according to the following rules:

1. M_i is the parent of t and there exist at least one enabled transitions.

2. M_i is the current process at which t reside, and at least one transition is still searching for an event that can enable it.

3. M_i is targeted by at least one transition since the transition is inconsistent or incomplete. (M_i might be the parent of t)

With every call of PROCESSTOKEN, the $NextTargetEvent$ and $NextTargetProcess$ of each transition are updated to reflect the state of the transition, and then the $NextTargetEvent$ and $NextTargetProcess$ of t are set according to the previous rules. Let's imagine that monitor process M_k is currently hosting t while it is waiting for the next event e^x that can enable its conjuncts in at least one transition, when the next event occurs which is a receive event from P_j, all the transitions will be in inconsistent state with process P_j (since t has outdated information about P_j), so t is sent to M_j targeting the inconsistent event which is guaranteed to have occurred. When M_j receives t, it looks up the target event and updates the transitions and invokes SENDTONEXTPROCESS which sends back t to M_k with target event e^{x+1}. If P_k keeps on receiving events from M_j and P_k's conjuncts participating in the transitions never become true, then t will keep on being sent back and forth between M_k and M_j, however since the sequence number of the $NextTargetEvent$ is always incremented by updating $gcut[k] + 1$, then when either P_k terminates, the conjunct will be set to false, and hence the sending/receiving cycle will be broken. We note that if the parent process of t is in an inconsistent state, then when t returns to the monitor process of the parent M_i, the new event might delete t or disable some transition or add new transitions.

Lemma 2 *A monitor process M_i never deadlocks.*

Proof 2 Each monitor process starts by: (1) sending out tokens to forbidding processes, and then (2) the monitor process waits to receive the tokens back. As shown in the previous lemma, each token returns in finite time, hence the routine is guaranteed to never deadlock.

Theorem 1 *Algorithm 1 eventually terminates when it monitors a terminating program.*

Proof 3 First, we note that our algorithm is designed for terminating programs and that a terminating program only produces finite computations, hence producing a finite number of local events in all normal processes. In order to prove the lemma, let us imagine that when the program terminates, it sends a *stop* signal to all normal and monitor processes. When such a signal is received by a normal process, it will not produce new events. When received by a monitor process M_i, it starts processing all the events stored in its $history$ vector. There can be two cases with respect to the waiting tokens list w_tokens in M_i. If processing events in $history$ results in the truthfulness of a local proposition for which there exists a waiting token in w_tokens, then the token will be sent back to the owner of the token with $true$ valuation for the local proposition. Otherwise, if $history$ becomes empty, then all waiting tokens should be sent back to the owners with value $false$ for the corresponding propositions. This would simply result in termination of all pending actions and, hence, the algorithm.

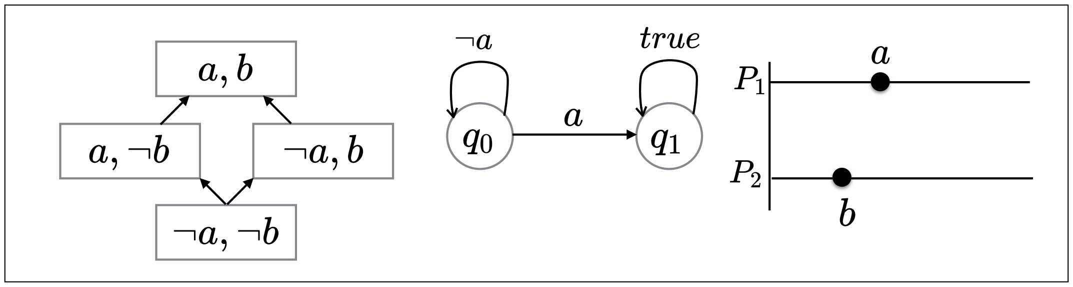


Figure 4.2: Lattice, automaton and computation for a distributed system with two processes and an initial global states with all propositions set to $false$.

4.5.2 Soundness and Completeness

In this section, we show that the algorithm presented is **s**ound and **c**omplete. First, we present the following definitions that help us describe the oracle's lattice:

Definition 16 (A lattice path π) *A path in the oracle's lattice is a sequence of nodes, each node representing a global state. Each global state is the result of exactly one process advancing its vector clock on one of the node's immediate parent state.*

For example in the lattice in Fig 4.2, the final node $\langle a, b\rangle$ with $VC = [1, 1]$ is the result of process P_2 advancing its state from $\langle a, \neg b\rangle$ with $VC = [1, 0]$ or P_1 advancing state from $\langle \neg a, b\rangle$ with $VC = [0, 1]$.

Definition 17 (Pivot global state g_p^q) *A pivot global state is a global state that exists as a node in the oracle's lattice and is labelled by the oracle with an automaton state q that is different than any of g_p^q's immediate parent nodes in the lattice.*

For example in the lattice in Fig 4.2, nodes $\langle a, \neg b\rangle$ and $\langle a, b\rangle$ are pivot global states with monitor state q_1 since they both have a parent node with a different monitor state.

Definition 18 (Process Progress Path π_i) *For each process in the distributed system, there exist a progress path π_i in the oracle's lattice that attempts to advance the global state on P_i only, until P_i receives a message from another process P_j which forces π_i to advance on both processes at the same time. Afterwards the path splits into multiple paths, where one of these new paths takes the role of π_i by advancing the global state on P_i only, if possible.*

To prove the soundness of the algorithm we need to prove that any path followed by a monitor process M_i to detect a pivot global state g_p^q at automaton state q is a path in the oracle's lattice. On the other hand to prove the completeness of the algorithm we need to prove that every path in the lattice that detects pivot global states is traced in at least one monitor process.

4.5.2.1 Soundness

Lemma 3 *If an event e^i received by the monitor process M_i participates in a pivot global state g_p^q in the oracle lattice, M_i will eventually detect g_p^q.*

Proof 4 When the monitor process M_i receives the event e^i, the event is added to the global view's $gv.gcut$ and $gv.gstate$ which might cause $gcut$ to be inconsistent (out of date) with some processes in $gstate$ array. This inconsistency occurs since the new event might have more updated information about other processes than $gcut$, which indicates that P_i has received messages from other processes. We show our proof for both cases, consistent $gcut$ and inconsistent $gcut$.

- If $gcut$ is consistent ($\nexists k, 0 < k \leq n : gv.gcut[k] < e.VC[k]$), there are three possibilities for the next state of the lattice:

  1. There exist only an enabled self-loop transition:
     Which indicates that e^i does not participate in any pivot global state, since there is no outgoing transition to a new state.
  2. There exist only an enabled outgoing transition:
     gv will advance to the new monitor state, thus detecting the pivot global state and advancing π_i on P_i . This is guaranteed in Algorithm 2 – Line 21, where the monitor state of gv is advanced and $gv.keepAfterFork$ is also set to $true$ to ensure that if there are next possible outgoing transitions, gv will not be deleted.
  3. There exists an enabled self-loop or an enabled outgoing transition in addition to a possibly enabled outgoing transition:
     After advancing to the new monitor state in the case of an enabled outgoing transition, or staying in the same monitor state in the case of enabled self-loop, gv will send a $token$ object that returns in finite time as shown in lemma 1, if the outgoing transition is enabled then gv forks a new global view for the newly detected pivot global state. If $token$ returns with a disabled or an inconsistent state for the outgoing transition, then e^i does not participate in a pivot global state.

  Note that it is not possible to have a consistent global state with no enabled self-loops or enabled outgoing transitions.

- If $gcut$ is inconsistent, i,e $\exists k, 0 < k \leq n : gv.gcut[k] < e.VC[k].VC$, then $gv.keepAfterFork$ will be set to $false$. Afterwards, there are three options:

  1. There are no possible outgoing transitions:
     This indicates that the event does not participate in any pivot global state, however, gv stays in the same automaton state since the automaton is complete, there must exist a possibly enabled self-loop. We do not attempt to fix this global state inconsistency since the monitor automaton state is not changed. The global state in this lattice path may have inconsistent data for other processes but the monitor automaton state is correct. Also, the inconsistent nodes will be corrected once there is a possibly enabled outgoing transition. Hence, gv will have with the same monitor state for e^i.
  2. There exists at least one possibly outgoing transition:
     This indicates that the event may participate in a pivot global state, then when CHECK-OUTGOINGTRANSITIONS is invoked and $gv.token$ is created and sent out to other processes with the possibly enabled transitions, $gv.token$ may return back to P_i with at least one enabled transition thus detecting pivot global state(s).

The previous breakdown for the algorithm shows that M_i will always detect if a received event participates in a pivot global state.

Theorem 2 *A monitor process M_i running at P_i always traces progress path π_i.*

Proof 5 When M_i receives an event it attempts to maintain gv as long as there is an enabled self-loop transition. If there are no enabled self-loop transitions, then as shown in lemma 3 at least one outgoing transition must be enabled and at least one new global view will be forked from gv and then gv will be deleted. The newly created global view will continue tracing progress path π_i in a similar manner.

Theorem 3 *Any path followed by a monitor process M_i to detect a pivot global state g_p^q at automaton state q is a path that exists in the oracle's lattice.*

Proof 6 Since the oracle lattice include every possible execution path for the program, then any path detected by a monitoring process that has consistent global states nodes exists in the oracle lattice. As shown in the previous lemmas, all global states detected by a monitoring process are consistent. Therefore, any path detected by a monitoring process is a path in the oracle lattice.

4.5.2.2 Completeness

Theorem 4 *Every pivot global state in the oracle lattice is detected by at least one monitor process.*

Proof 7 We will prove this theorem by contradiction, let's assume that there is a pivot global state that is undetected by any monitor process. If this pivot global state coincide with a process progress path π_i then there is a contradiction with Theorem 2 since the monitor process M_i must have detected this pivot global state. If the pivot global state node is spawned by multiple process progress paths, then as shown in Theorem 2 all the monitor processes tracing these process progress paths will fork a new global view that detects the pivot global state.

Chapter 5

Experimental Results

In this section, we present the results of a set of experiments to evaluate our monitoring algorithm. Section 5.1 introduces our case study, while Sections 5.2 and 5.3 describe the experimental settings and result, respectively.

5.1 Case Study

To illustrate the resilience of our proposed algorithm, we ran our monitoring algorithm on a network of iOS devices. Each device is running a simple program that updates local variables and communicates with other devices using peer to peer communication. The program running on the devices loads a trace file containing the wait time between events. Events are either a variable's value change (internal) event or a send event to other processes. A process can be idle or doing computations that does not affect the local propositions during the wait time. The wait time between events is generated using normal distribution, we show results for different values of mean and sigma. Each program running on process P_i has two propositions: $P_i.p$ and $P_i.q$, the values of the propositions are also read from the trace file.

We generated the automaton for six different LTL$_3$ properties to demonstrate how the complexity of the automaton affects the performance of the monitor. Note that that automaton complexity differs according to the number of propositions and processes participating in the property. As the number of processes and propositions increase, the number of transitions increase, also the number of conjuncts in the predicates labelling the transitions increase.

- **Property A**

  $$\square((P_0.p \wedge P_1.p)\, \mathbf{U}\, (P_2.p \wedge P_3.p))$$

  That is, it is always the case that proposition p for processes P_0 and P_1 should stay true until the propositions p for processes P_2 and P_3 are true concurrently. For simplicity, fig. 5.2a shows the LTL$_3$ monitor automaton for Property A with only 2 processes.

- **Property B**

  $$\lozenge(P_0.p \wedge P_1.p \wedge P_2.p \wedge P_3.p)$$

  That is, eventually, the propositions for all the processes becomes true concurrently. For simplicity, fig. 5.2b shows the LTL$_3$ monitor automaton for Property B with 2 processes. We note that since the automaton only includes one outgoing transition, the overhead should be relatively lower than other automatons overhead.

- **Property C**

$$\Box((P_0.p)\,\mathbf{U}\,(P_1.p \wedge P_2.p \wedge P_3.p))$$

  That is, it is always the case that proposition p for process P_0 should stay true until proposition p for processes P_1, P_2 and P_3 are all true concurrently. We note that this automaton resembles the automaton for property A, therefore we skip showing the automaton diagram. The main difference between property A and C is that for property A, processes P_0 and P_1 will have a higher load than the other two processes since the propositions for P_2 and P_3 would only need to be checked if the propositions at P_0 and P_1 are false. However, automatons A and C for the 2 processes and 3 processes experiments are identical.

- **Property D**

$$\Box((P_0.p \wedge P_1.p \wedge P_2.p \wedge P_3.p)\,\mathbf{U}\,(P_0.q \wedge P_1.q \wedge P_2.q \wedge P_3.q))$$

  That is, it is always the case that the propositions p for process P_0, P_1, P_2 and P_3 must remain true until the proposition q for process P_0, P_1, P_2 and P_3 become true concurrently. For simplicity, fig. 5.2c shows the LTL_3 monitor automaton for Property A with 2 processes. We note that all processes appear equally in all outgoing transitions which indicate fair overhead among processes.

- **Property E**

$$\Diamond(P_0.p \ \wedge P_1.p \wedge P_2.p \wedge P_3.p \wedge P_0.q \ \wedge P_1.q \wedge P_2.q \wedge P_3.q)$$

  That is, eventually, all the propositions for all the processes become true. Fig. 5.3a shows the LTL_3 monitor automaton for Property E with 2 processes. We note that since the only outgoing transition is a conjunction between all the propositions, then a process would only send a message if locally both propositions are true.

- **Property F**

$$\Box((P_0.p\,\mathbf{U}\,(P_1.p \wedge P_2.p \wedge P_3.p) \ \wedge (P_0.q\,\mathbf{U}\,(P_1.q \wedge P_2.q \wedge P_3.q)))$$

  That is, it is always the case that (1) the propositions p for process P_0 must remain true until the proposition p for process P_1, P_2 and P_3 become true concurrently, and (2) the propositions q for process P_0 must remain true until the proposition q for process P_1, P_2 and P_3 become true concurrently. For simplicity, fig. 5.3b shows the LTL_3 monitor automaton for Property A with 2 processes.

We note that properties A, C and D can be reduced to a simple automaton that verifies that all propositions are not false at the same time, however we use the complicated version of the automaton for two reasons: (1) it provides more information as q_1 is a $q_?$ state, and (2) it provides a more complicated structure to test our algorithm.

The variable valuation change events were designed such that there would be a path in the execution lattice that would lead to a final state.

Table 5.1 shows the number of transitions in the generated automatons for each property. Also Fig. 5.1a and Fig. 5.1b shows the graphical representation for all transitions and outgoing transitions only count respectively.

Table 5.1: Number of transitions per automaton

Number of Processes	2			3			4			5		
	Total	Out-going	Self-loops	Total	Out-going	Self-loops	Total	Out-going	Self-loops	Total	Out-going	Self-loops
Property A	7	4	3	11	7	4	15	11	4	21	16	5
Property B	4	1	3	5	4	1	6	1	5	7	1	7
Property C	7	4	3	11	7	4	15	11	4	19	13	6
Property D	15	11	4	27	22	5	43	35	7	63	56	7
Property E	6	1	5	8	1	7	10	1	9	12	1	11
Property F	31	23	8	49	37	12	67	51	16	85	65	20

Table 5.2: Technical specification of the iOS devices

Device	CPU	Memory
iPhone 5s	Dual-core 1.3 GHz Cyclone (ARM v8-based)	1 GB RAM DDR3
iPad mini 3	Dual-core 1.3 GHz Cyclone (ARM v8-based)	1 GB RAM DDR3
iPad Air 2	Triple-core 1.5 GHz	2 GB RAM
iPhone 6 (simulator)	Dual-core 1.4 GHz Typhoon (ARM v8-based)	1 GB RAM DDR3

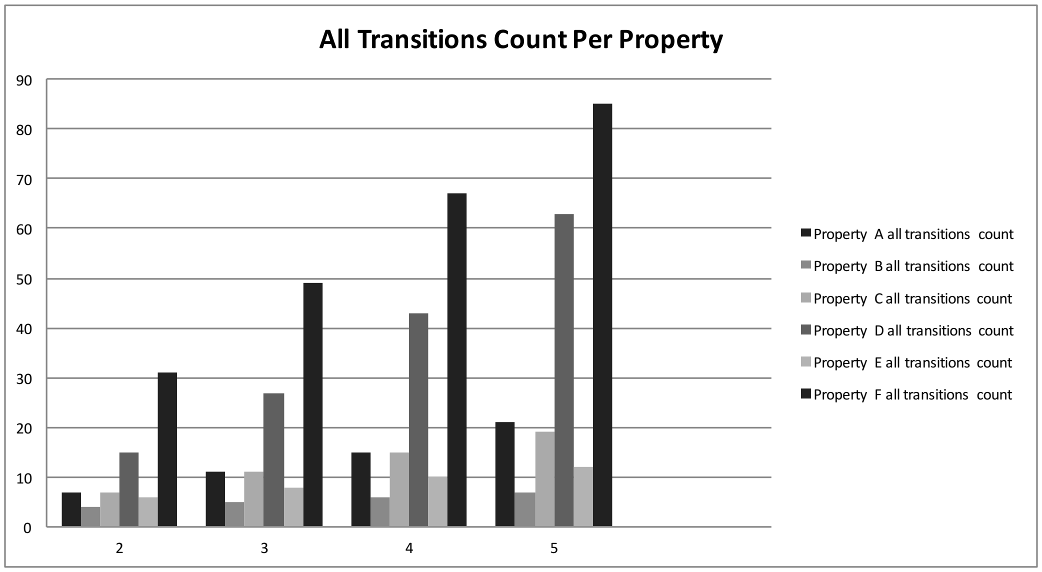
5.2 Experimental Settings

We tested our distributed monitoring algorithm on a heterogeneous collection of iOS devices connected by wifi. The experiments included 2 iPhone 5S devices, and 1 iPad Air 2 and 1 iPad mini 3 and 1 iPhone 6 simulator, Table 5.2 shows the specifications for the used devices.

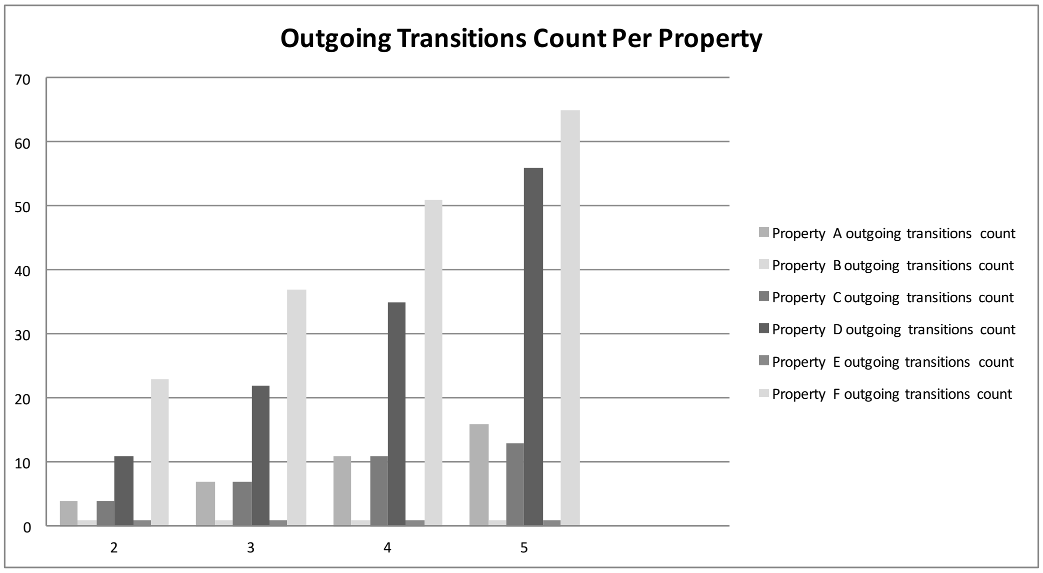
The algorithm is implemented in C and was integrated with Objective-C code that is responsible for loading the generated traces, updating the variables and communicating with other devices. We consider the following experimental parameters:

- LTL$_3$ Properties A, B, C, D, E and F.
- Number of iOS devices; i.e., 2, 3, 4 and 5 devices.
- Normal distribution parameters (Evt_μ and Evt_σ) for the wait time of variable's valuation change events.
- Normal distribution parameters ($Comm_\mu$ and $Comm_\sigma$) for the wait time of processes communication events. We note that when a communication event occurs, the program at P_i sends a message to each other process in the distributed system.

We measure the following metrics:



(a) All transitions (outgoing and self-loops) count per experiment size for each property.

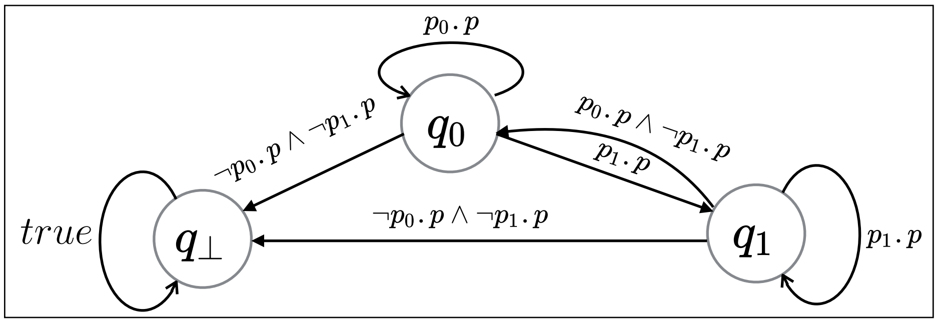


(b) Outgoing transitions count per experiment size for each property.

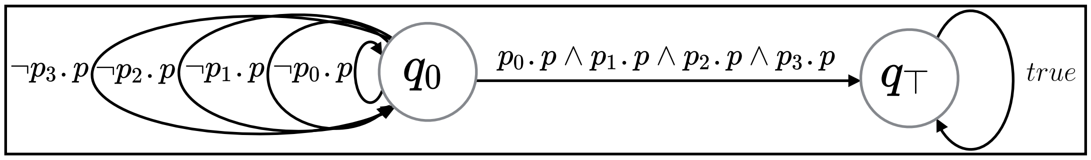
Figure 5.1: Transitions count per property

- Total number of monitoring messages sent from all monitor processes.
- Average delay in terms of the mean number of events delayed at the monitor process waiting for other monitor processes.
- Percentage of extra monitoring time needed to process the program events versus the actual program time. To normalize this metric with respect to the trace complexity, we divide the percentage by the total number of global states found in the execution.
- The total number of global views created in all the monitoring processes.

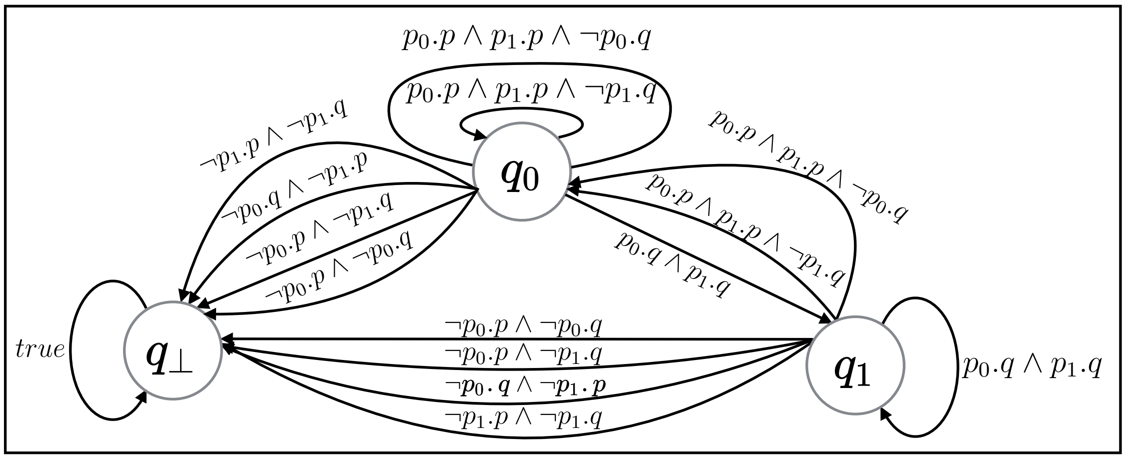
We have replicated the experiments three times with different randomly generated traces and averaged the results.


$p_0 . p$
q_0
$\neg p_0 . p \wedge \neg p_1 . p$
$p_0 . p \wedge \neg p_1 . p$
$p_1 . p$
$true$
$q_\bot$
$\neg p_0 . p \wedge \neg p_1 . p$
q_1
$p_1 . p$

(a) Property A for 2 processes



(b) Property B for 4 processes


$p_0.p \wedge p_1.p \wedge \neg p_0.q$
$p_0.p \wedge p_1.p \wedge \neg p_1.q$
q_0
$\neg p_1.p \wedge \neg p_1.q$
$\neg p_0.q \wedge \neg p_1.p$
$\neg p_0.p \wedge \neg p_1.q$
$\neg p_0.p \wedge \neg p_0.q$
$p_0.p \wedge p_1.p \wedge \neg p_0.q$
$p_0.p \wedge p_1.p \wedge \neg p_1.q$
$p_0.q \wedge p_1.q$
$true$
$q_\bot$
$\neg p_0.p \wedge \neg p_0.q$
$\neg p_0.p \wedge \neg p_1.q$
$\neg p_0 . q \wedge \neg p_1 . p$
$\neg p_1.p \wedge \neg p_1.q$
q_1
$p_0.q \wedge p_1.q$

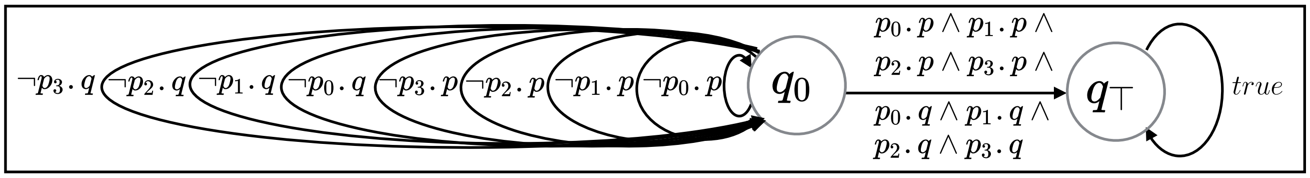
(c) Property D with 2 processes

Figure 5.2: Monitor Automaton for properties A, B and D

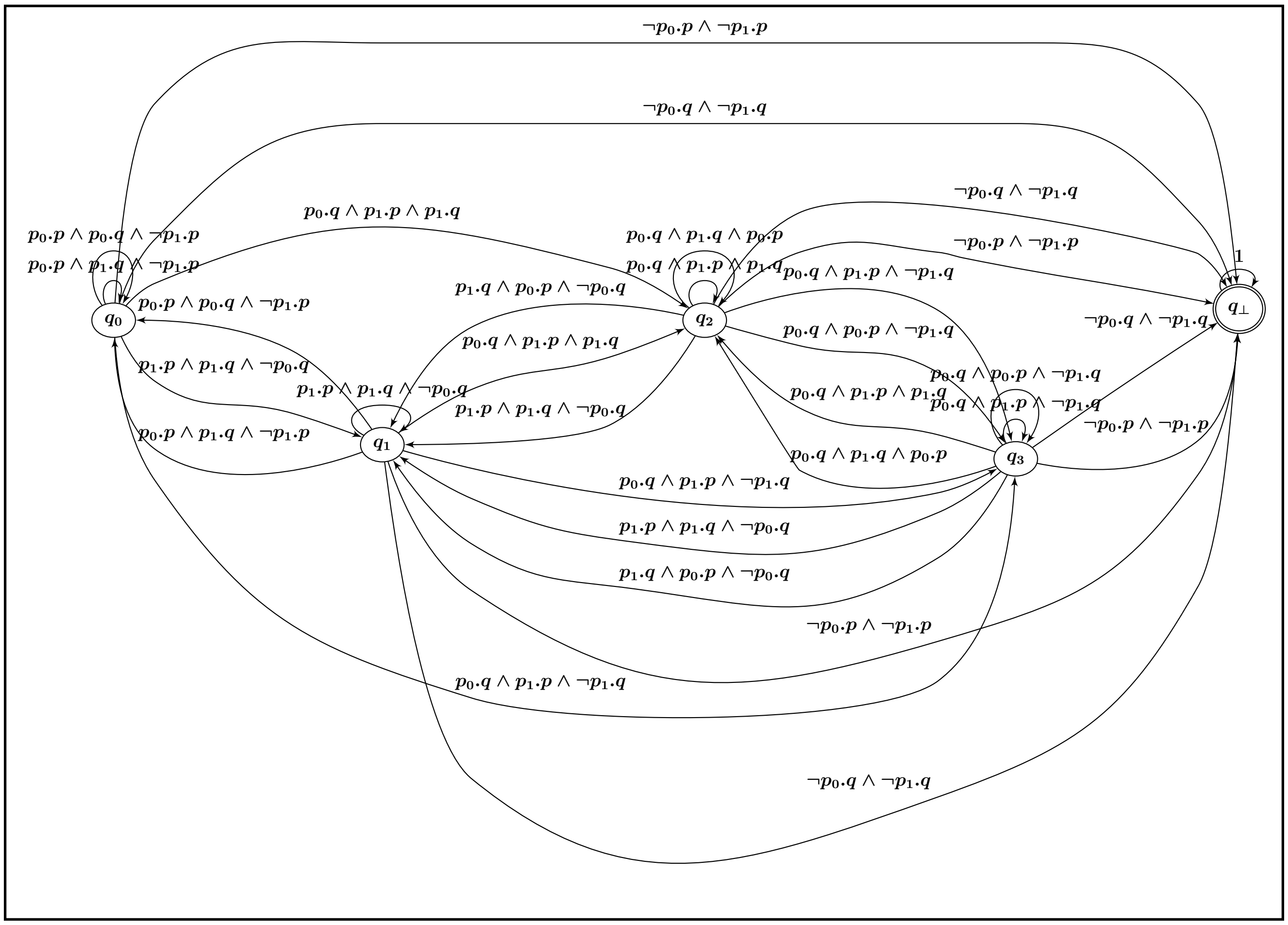
5.3 Results

5.3.0.1 Monitoring Communication Overhead

- *(Property A)* As can be seen in Fig 5.4a, the number of monitoring messages is growing linearly with the number of processes and events. This indicates that the monitor is able to scale well with the number of events generated by the program.
- *(Property B)* As can be seen in Fig 5.4b, the growth in the number of monitoring messages is sub-linear to the number of all events. This is because for the only outgoing transition from the initial state, monitor processes do not have to check other processes unless their local proposition is evaluated to true.
- *(Property C)* As can be seen in Fig 5.4c, similar to property A, the number of messages is growing linearly with the events.
- *(Property D)* As can be seen in Fig 5.5a, the number of messages is growing linearly with the events.



(a) Property E for 4 processes



q_0
q_1
q_2
q_3
$q_\perp$
$\neg p_0.p \wedge \neg p_1.p$
$\neg p_0.q \wedge \neg p_1.q$
$p_0.q \wedge p_1.p \wedge p_1.q$
$p_0.p \wedge p_0.q \wedge \neg p_1.p$
$p_0.p \wedge p_1.q \wedge \neg p_1.p$
$p_1.p \wedge p_1.q \wedge \neg p_0.q$
$p_1.q \wedge p_0.p \wedge \neg p_0.q$
$p_0.q \wedge p_1.q \wedge p_0.p$
$p_0.q \wedge p_1.p \wedge p_1.q$
$p_0.q \wedge p_1.p \wedge \neg p_1.q$
$p_0.q \wedge p_0.p \wedge \neg p_1.q$
1

(b) Property F for 2 processes

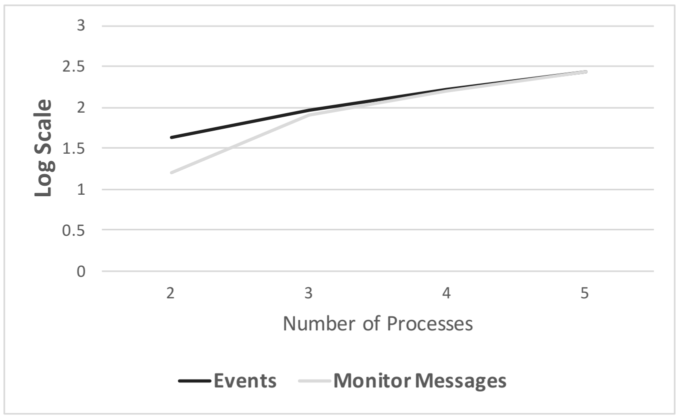
Figure 5.3: Monitor Automaton for properties E and F

- *(Property E)* As can be seen in Fig 5.5b, the growth is similar to the growth of property B, since both properties have only one outgoing transition.
- *(Property F)* As can be seen in Fig 5.5c, the number of messages is growing linearly with the events.

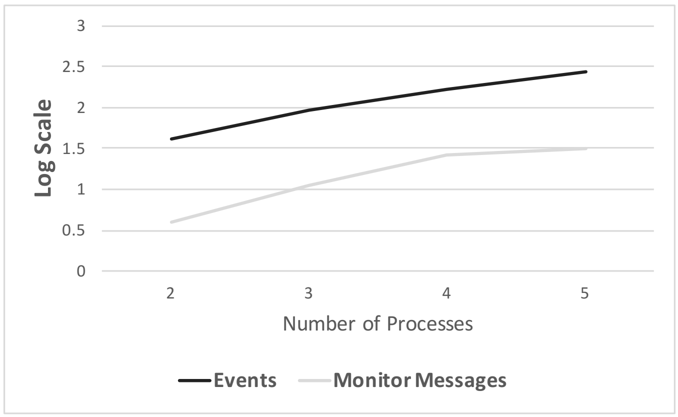
We argue that since the number of messages grows linearly with the number of events, one can piggyback the monitoring messages on the processes communication and decrease the number of monitoring messages significantly.

5.3.0.2 Detection Latency

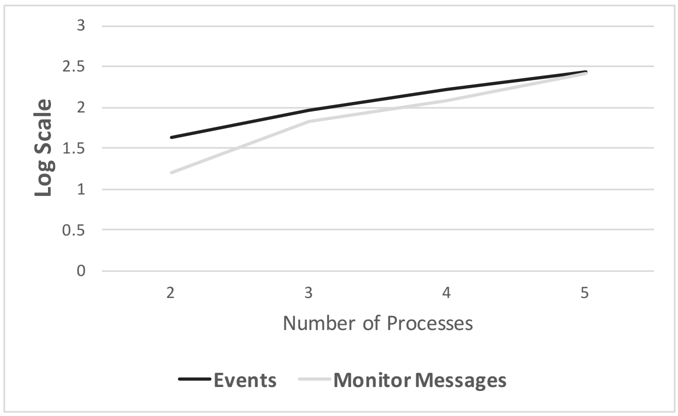
In Fig. 5.6a and Fig 5.6b, we use the following formula to calculate the delay time percentage per global state:



(a) Property A Monitoring Messages with $Comm_{\mu} = 3sec$, $Comm_{\sigma} = 1sec$, $Evt_{\mu} = 3sec$ and $Evt_{\sigma} = 1sec$



(b) Property B Monitoring Messages with $Comm_{\mu} = 3sec$, $Comm_{\sigma} = 1sec$, $Evt_{\mu} = 3sec$ and $Evt_{\sigma} = 1sec$

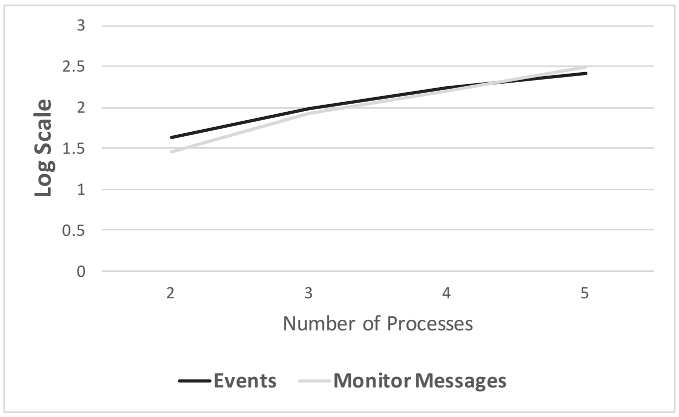


(c) Property C Monitoring Messages with $Comm_{\mu} = 3sec$, $Comm_{\sigma} = 1sec$, $Evt_{\mu} = 3sec$ and $Evt_{\sigma} = 1sec$

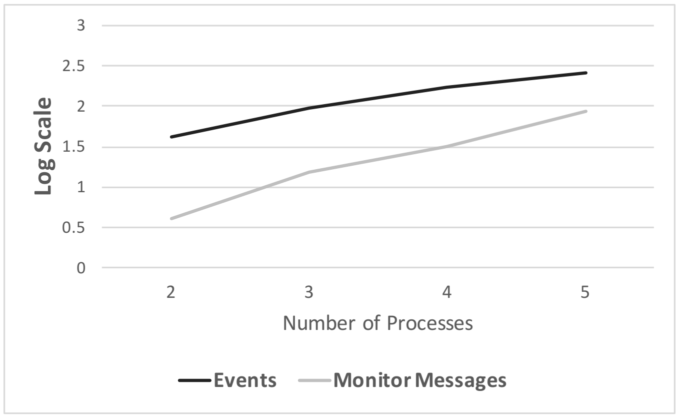
Figure 5.4: Messages Overhead for properties A,B and C

$$((MonitorExtraTime/ProgramTime) * 100)/TotalGlobalStatesCreated$$

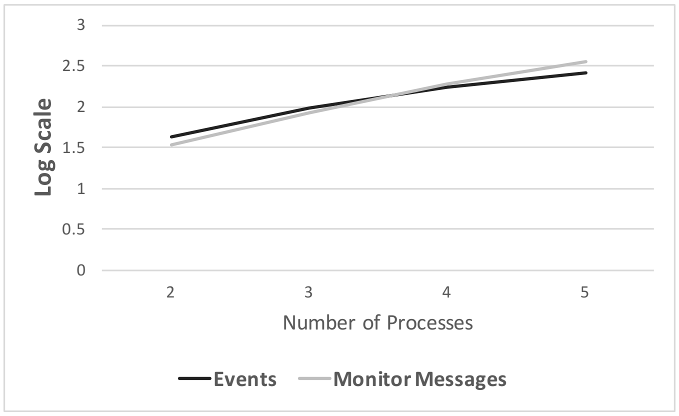
The purpose of this formula is to capture normalized delay in terms of time instead of delayed events. Monitors are expected to terminate with the program, however, in our experiments, we ran across experiments whose monitoring lasted more than the program time due to the program



(a) Property D Monitoring Messages with $Comm_{\mu} = 3sec$, $Comm_{\sigma} = 1sec$, $Evt_{\mu} = 3sec$ and $Evt_{\sigma} = 1sec$



(b) Property E Monitoring Messages with $Comm_{\mu} = 3sec$, $Comm_{\sigma} = 1sec$, $Evt_{\mu} = 3sec$ and $Evt_{\sigma} = 1sec$



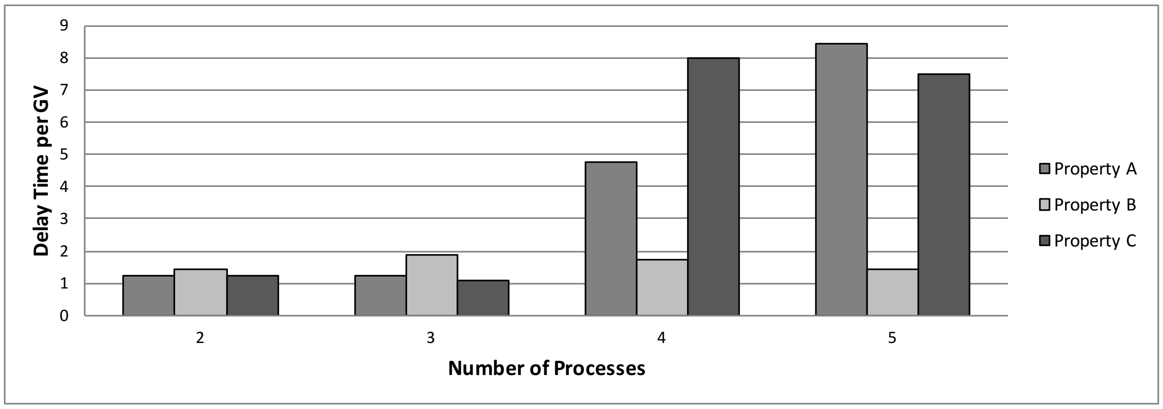
(c) Property F Monitoring Messages with $Comm_{\mu} = 3sec$, $Comm_{\sigma} = 1sec$, $Evt_{\mu} = 3sec$ and $Evt_{\sigma} = 1sec$

Figure 5.5: Messages Overhead for properties D,E and F

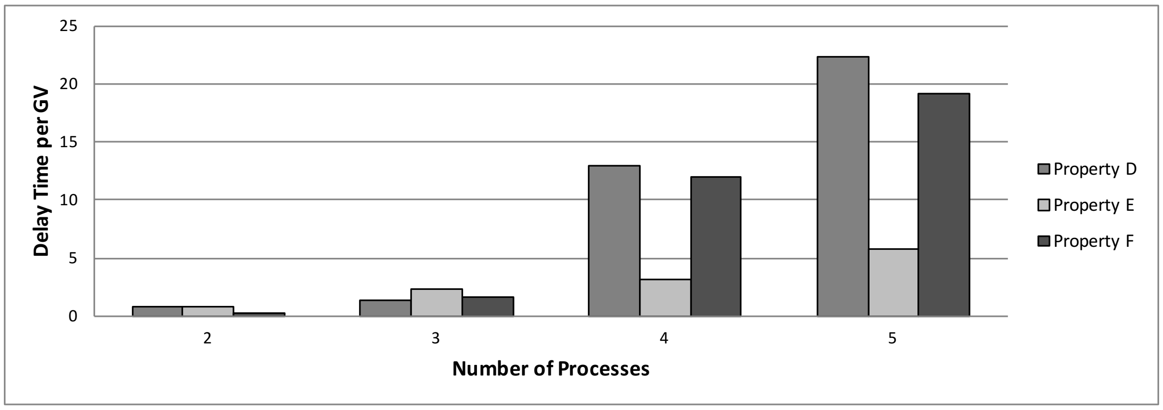
execution that resulted in a large number of global states.

In Fig. 5.7a and Fig 5.7b, we show the average number of events queued (delayed) at the monitor processes while the monitor was processing final events.

Both graphs show clearly that the increase in the delay is significant as the number of processes increase for properties A,C,D and F, while properties B and E have a linear growth in the



(a) Delay measured as the extra time percentage required by the monitor to detect final state.



(b) Delay measured as the extra time percentage required by the monitor to detect final state.

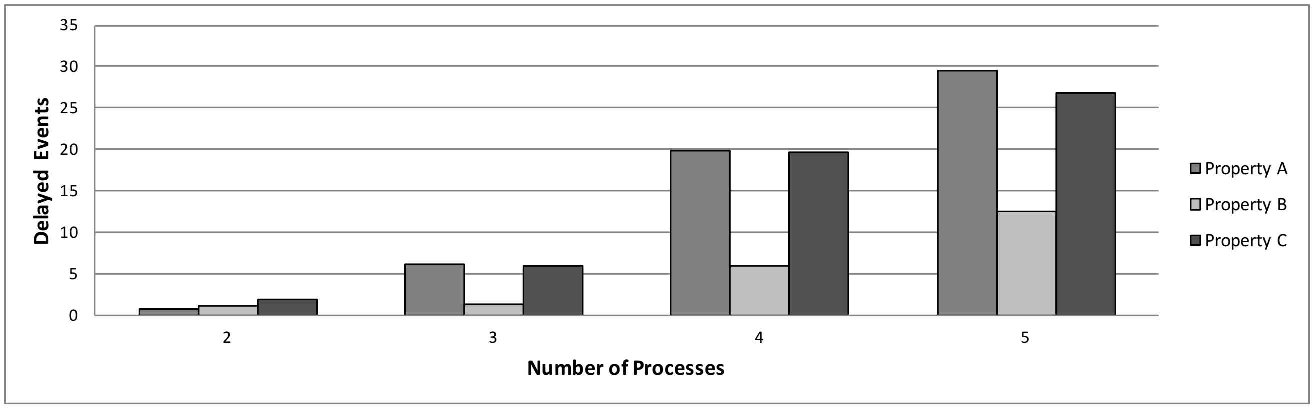
Figure 5.6: Delay Time Percentage

delay. This can be attributed to the simple design of these two properties leading to lower number of spawned global states.

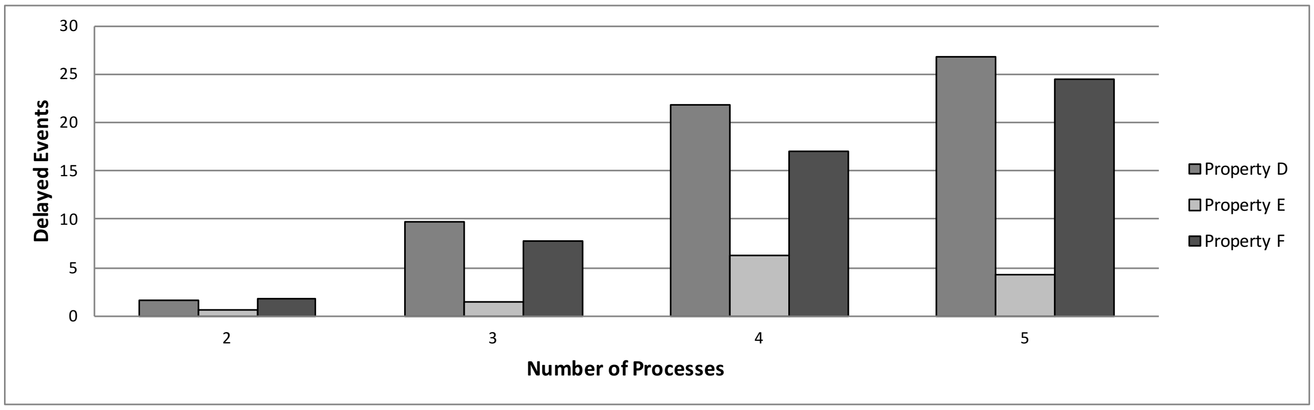
Also, the increase in the delay as the number of processes increase can be attributed to our algorithm design that attempts to decrease the communication overhead (by routing all token's transitions together and aggregating global views tokens in send messages) on the expense of the delay.

5.3.0.3 Memory Overhead

Fig. 5.8a and Fig 5.8b show the memory overhead represented in the total number of global views created in all processes for all properties. As can be seen, the general trend is that the growth is linear with the number of processes, which indicates that the memory usage of the algorithm is scalable as the number of processes increase. Also, the graphs indicate that the complexity of the automaton affects the number of global views created. For example, properties B and E have the least total number of global views, this can be attributed to their simple automaton shape (only one outgoing transition). While more complicated automatons such as property F automaton (85 transitions for the 5 processes experiment) has significant higher number of global views.



(a) Delay measured as the average number of events queued while processing final events.



(b) Delay measured as the average number of events queued while processing final events.

Figure 5.7: Delayed Events

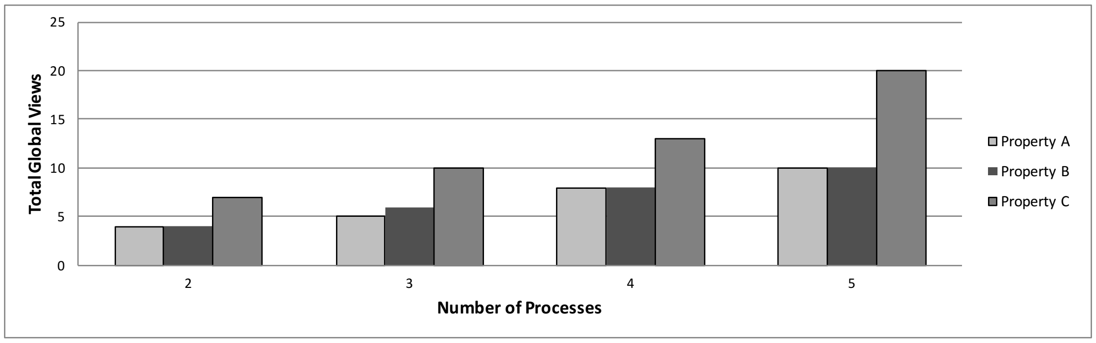
5.3.0.4 Communication Frequency

Fig. 5.9a shows the effect of varying the communication frequency on messages overhead for the 4 processes experiment running property C. As can be seen the number of events decrease when the communication frequency decreases since receive events are counted along with the total events because receive events affects the vector clock of the receiving process. As a result, the messages overhead also decreases. However, another factor contributes to the decrease in the number of messages overhead, which is the fact that less communication means less inconsistent global views leading to fewer messages exchange between monitors to fix the inconsistent states.

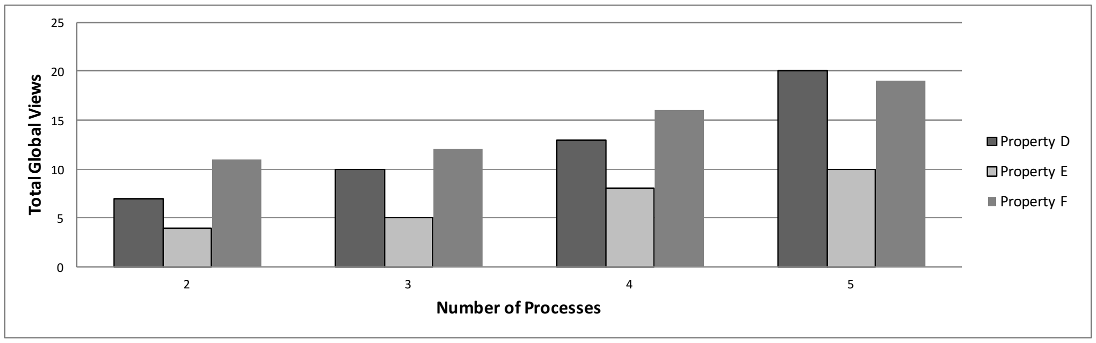
Fig 5.9b shows the effect of varying the communication frequency on the delay for the 4 processes experiment running property C. As can be seen as the communication frequency decreases, the delay decreases as well, since fewer events are inconsistent with local global views and processed directly without waiting for other monitors to fix the inconsistency.

We note that the delay is significantly larger for the no communication experiment since as the communication between processes disappear, any event e at process P_i is considered concurrent with event f at process P_j. Which shows that absence of communication affects delay as much as the abundance of communication does. This also shows that must be an optimum frequency of communication that optimizes the delay and communication overhead.

Fig 5.9c shows the effect of varying the communication frequency on the total number of global views created for the 4 processes experiment running property C. As can be seen the total number of global views is generally increasing as the frequency of communication decreases.



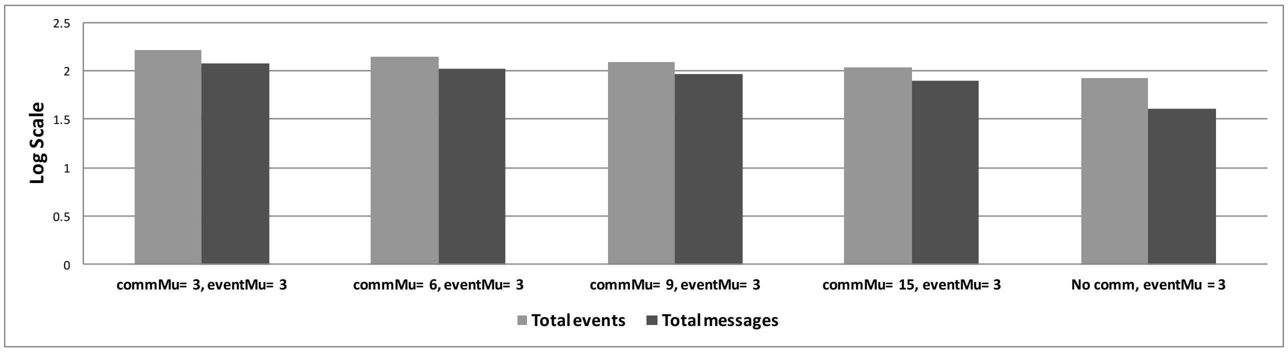
(a) Memory overhead measured as the total number of global views created through out the experiment.



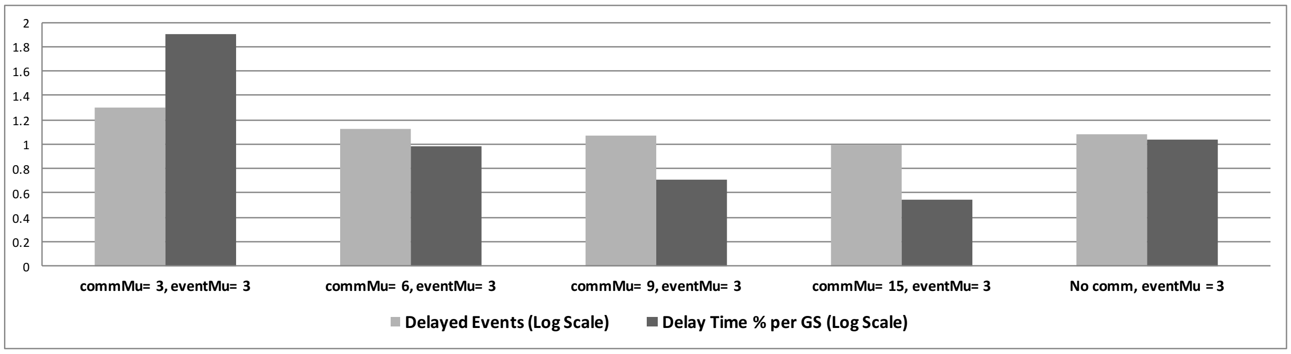
(b) Memory overhead measured as the total number of global views created through out the experiment.

Figure 5.8: Memory Overhead

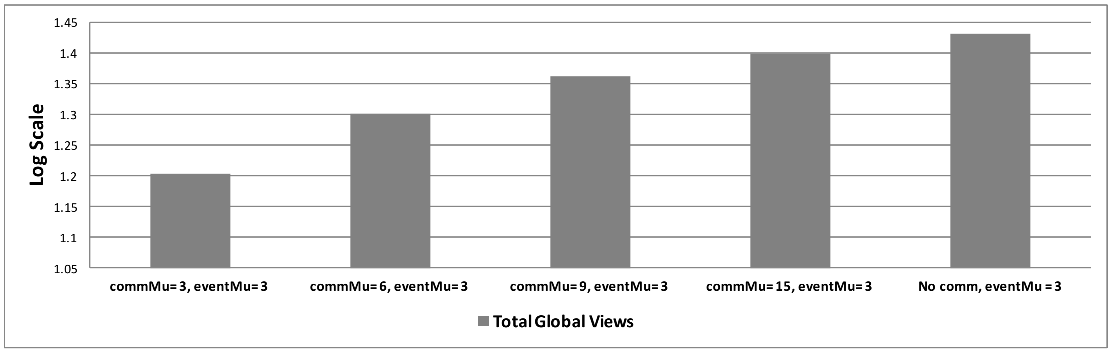
This is due to the increased width of the execution lattice as more events are concurrent, therefore more global views are created to explore the paths in the lattice.



(a) Varying communication frequency effect on monitoring messages for 4 processes running property C.



(b) Varying communication frequency effect on delay for 4 processes running property C.



(c) Varying communication frequency effect on memory measured as the total global views created through out the 4 processes experiment running property C.

Figure 5.9: Communication Frequency

Chapter 6

Related Work

In this chapter we discuss the related work for the three different aspects of this work:

- Formal Verification
- Distributed Systems Verification
- Runtime verification with Linear Temporal Logic

6.1 Formal Verification

6.1.1 Verification Methods

The taxonomy in Fig. 6.1 shows the two types of formal verification: online and offline. In *Static Analysis* [27], the source code for the program is analyzed against all possible inputs to ensure a certain correctness property is satisfied. For example, to check a program is free of the division by zero bug, all possible inputs for the division operator will be generated and tested. A good static analysis technique would only consider the inputs that can actually be passed on to the division operator not all possible values for the operands. Static analysis tools are languages specific and can not reason about pointers.

Unlike static analysis, *Model Checking* [27] deals with the model of the program, not the code. A model for a program typically consists of states and transitions, where states describe the program state, variables, and stack while the transitions describe how the program advances from a state to another. Model checking algorithms verifies recursively that all the states reachable from the initial state satisfy the correctness property and provides a counter-example if the

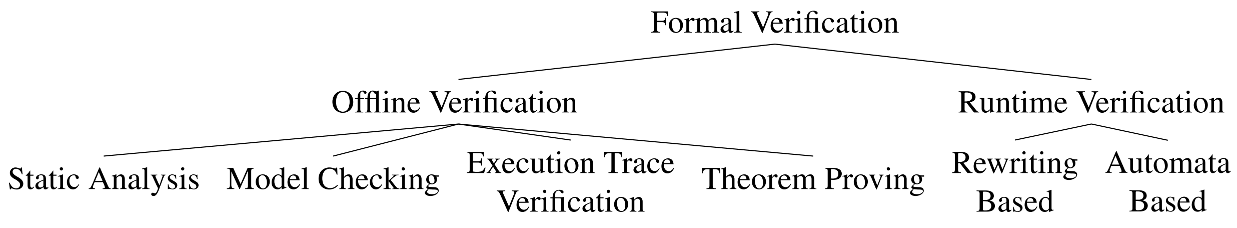


Figure 6.1: Formal Verification Taxonomy

model does not conform to the property. Model checking suffers from state-space explosion for relatively large programs. Also model checking accuracy depends on the model accuracy, i,e, if a system model representation is incorrect, model checking will yield incorrect results.

In *Execution Trace verification* the execution trace of a terminated program is passed on to the verification algorithm. The algorithms verify the execution trace conforms to the correctness property. Note that the verification algorithm only checks the execution path executed by the program, not every possible execution path as in model checking or static analysis. Since execution trace verification is an offline technique, there is no way to rectify the incorrect behavior of the program. Also, it can only detect incorrect behavior if they appear in the trace, i,e, some bugs may go unnoticed since they have not been executed.

Theorem Proving [14] aims at verifying that an implementation satisfies a correctness property through mathematical reasoning. The implementation and the correctness property are both expressed as logical formulas and an equivalence or implication relation between them is proved. Theorem provers reason in the syntactic domain through constraints on states, unlike model checkers which reason in the semantics domain through enumerating all possible states, theorem provers are better suited for data-intensive programs where model checkers run into state-space explosion problems [23]. However, theorem provers suffer from the lack of complete automation, language limitations such as lack of pointers support, proofs are large and difficult to understand and lastly practical systems that can be efficiently represented by logical formulas are limited.

Runtime Verification is considered a nice tradeoff between ad-hoc testing and heavy techniques such as model checking and theorem proving. Runtime verification is a lightweight technique, where the verification algorithm runs in parallel with the program under scrutiny. The algorithm reads the events that occur in the program immediately and analyzes the state of the program with respect to the correctness property. Since the verification occurs in runtime, the verifying algorithm can issue warnings to the program to rectify/terminate the incorrect behavior. However, as with execution trace verification, runtime verification only inspects execution paths that occurred. So it may fail to detect a violation to the correctness property that requires specific scenario to occur.

Linear Temporal Logic (LTL) is used heavily in runtime verification as it is best suited to model infinite execution traces. In [13], the authors present a technique to monitor programs for an LTL property using formula rewriting where the LTL formula consumes each state in the trace and produces a new formula, and at the end of the trace, the formula should be evaluated to either true or false. In [4], the authors present a technique to synthesize a deterministic finite state machine given an LTL_3 property, such that the FSM can identify the trace as satisfying or falsifying a property as early as possible.

6.1.2 Distributed Systems Formal Verification

Formal verification of distributes systems is crucial for many systems [17, 16, 24] especially safety critical systems where the non-determinism nature of distributed systems can be catastrophic if not anticipated. Therefore, critical systems usually undergo both online and offline verification next to other mechanisms such as redundancy and diversity.

6.2 Distributed Systems Runtime Verification

In this subsection, we describe the different parameters that affect the design of distributed systems verification.

6.2.1 Online Versus Offline

In online monitoring, the monitor(s) are running alongside the monitored program nodes and are expected to detect violation/satisfaction to the correctness properties as soon as they occur.

Offline monitoring takes place after the distributed program has terminated. Offline monitoring is best used for testing purposes, where the program is executed many times with different inputs and then the test logs are processed by the monitor(s). Usually, in offline monitoring, a single monitor is enough since no runtime requirements are present, however in some cases where the terminated program traces are large, decentralized monitors may be used.

6.2.2 Synchronous Versus Asynchronous Distributed Systems

The design of the distributed systems affects greatly the design of the monitoring system. Synchronous distributed systems that depend on global clock are relatively easier to monitor than asynchronous distributed systems, where each node has local clock. In a synchronous distributed system, a monitoring node(s) can easily order different events generated by different nodes and apply them sequentially to the correctness property.

However, asynchronous distributed systems suffer greatly from clock drifts and therefore, the monitoring node(s) can not use the events' timestamps from different nodes to order the events. Instead, monitoring node(s) use partial order inferred by the communication between the nodes to order the events. Note that total order can not be accomplished in an asynchronous setting, therefore the monitoring node(s) has to consider some events to be concurrent.

6.2.3 Monitor Design

In this subsection we talk briefly about the different approaches for distributed systems monitoring designs. Table 6.1 shows a brief comparison between the pros and cons of the different approaches discussed below.

6.2.3.1 Centralized Monitor

A centralized monitor resides on either, a node dedicated for monitoring the program nodes participating in the distributed systems, or resides on one of the program nodes. A centralized monitor is responsible for receiving all event from program nodes, evaluating the correctness property and declaring violation or satisfaction. The central monitor is also responsible for ordering the events it receives from the program nodes, which is a harder job if the system is asynchronous.

In [3], the authors present a framework for detecting and analyzing synchronous distributed systems faults in a centralized manner using LTL properties.

In [18], the authors present a monitoring technique that uses symbolic composition of events with the monitor to detect satisfaction/violation of LTL properties in an asynchronous distributed system.

6.2.3.2 Decentralized Monitors

Decentralized monitoring aims at decentralizing the monitoring load from one centralized monitoring node to several monitoring nodes. Each monitoring node is attached to one program node and receives the events from the program node as soon as they happen. The decentralized design offers many advantages compared to the centralized design such as the absence of single point of failure/attack, faster notification of failures or violations, distributed memory and computation overload among the monitoring nodes. However, decentralized monitors are required to communicate together to evaluate the correctness property leading to a complicated design.

In [26], the authors design a method for monitoring safety properties in distributed systems using the past-time linear temporal logic (PLTL). However, their algorithm is not sound, meaning that valuation of some predicates and properties may be overlooked. This is because monitors gain knowledge about the state of the system by piggybacking on the existing communication among processes. That is, if processes rarely communicate, then monitors exchange very little information and, hence, some violations of properties may remain undetected.

Lattice-theoretic centralized and decentralized online predicate detection has been studied in [7, 20]. However, this line of work does not address monitoring properties with temporal requirements. This shortcoming is addressed in [22] for a fragment of temporal operators, but for offline monitoring.

Also, in [5], the authors introduce parallel algorithms for runtime verification of sequential programs. Finally, in [10], the authors introduce a lower-bound on the number of values that a logic must have in order to monitor safety properties in distributed algorithms in a decentralized fashion in the presence of crash faults.

6.2.3.3 Migration

In the migrating monitors design, the monitor process migrates from a program process to another with the objective of minimizing computation and communication overhead.

In [2], the authors present a runtime verification algorithm of LTL for synchronous distributed systems, where processes share a single global clock. The LTL property is progressed as it migrates between processes using formula rewriting techniques.

6.2.3.4 Choreography

In the choreography design presented first in [8], the authors present a technique where the monitor nodes are arranged in a tree-like structure, and the LTL formula is divided into subformulas, each leaf node is responsible for evaluating local propositions, while the intermediate nodes aggregate the results and forwards it upwards until the formula is evaluated. In [11], dynamic choreography is proposed which is similar to state choreography presented in [8], but dynamic choreography rearranges the network of monitors during execution allowing monitoring dynamic properties such as properties that are created or evolved during runtime.

Table 6.1: Comparison between the pros and cons of different monitor design approaches

	Centralized	Decentralized	Migration	Choreography
Pros	Simple design	No single point of failure/attack	Monitor location is optimized to reduce overhead	Security is enforced as processes do not expose data to each other
	Little overhead incurred on program processes	Decentralized network overhead	Little overhead incurred on program processes	Distributed communication load
Cons	Single point of failure/attack	Memory and computational overhead on program processes	Moving single point of failure/attack	Sensitive to the depth of the LTL formula
	Events history can cause memory overhead	Complicated design	Complicated design	May require more nodes than the program nodes.
	Centralized network overhead	Security concerns if some nodes are not trusted	Migration process adds overhead and delay	

6.3 Predicate Detection versus Linear Temporal Logic

Predicate Detection is considered to be an easier problem than monitoring LTL properties, this is due to the fact that the latter is a subset problem of the former. Predicate detection aims at collecting the consistent cuts that satisfy a single predicate, while LTL monitoring aims at (1) identifying a consistent cut that satisfies any of the predicate labelling the current state's outgoing transitions, (2) advancing the automaton state, then (3) identifying a new consistent cut for any of the predicates labelling the outgoing transitions of the new automaton state. Therefore, techniques such as *Computation Slicing* introduced in [20] can not be applied as-is in LTL monitoring since the predicate is constantly changing, and the monitor could possibly be dealing with more than one predicate at the same time. However, we borrow some of the techniques of computation slicing, particularly the technique for finding the least consistent cut that satisfies the predicate.

Chapter 7

Conclusion

7.1 Summary

In this thesis, we proposed an algorithm for runtime verification of asynchronous distributed applications with respect to LTL formulas. Our technique addresses the shortcomings of the state of the art, such as requiring global clock [2], or sacrificing soundness in favour of minimizing communication [26]. Our approach is lightweight and provides a robust monitoring technique per process that is essential for critical distributed systems.

Our specification language is full LTL and, hence, our method can monitor temporal properties as well as logical predicates. Furthermore, since our algorithm depends on finite state machines, our monitoring technique does not depend on LTL directly, and therefore can be used with any language that can generate a finite state machine.

Finally, our algorithm is sound and complete, meaning that if the total order of events in the system under inspection can be constructed and presented in a lattice, and then each path in the lattice is processed through the finite state machine, then the combined final verification verdicts (if exists) will be determined by our algorithm as well (and vice versa). We also showed how the automaton shape and communication frequency affect the complexity of the algorithm.

We implemented and tested our algorithm on a network of five iOS devices communicating over wifi network with six LTL_3 properties. We reported the experimental results experiments for monitor overhead in terms of communication overhead, memory overhead, and detection latency.

Our experimental results clearly shows that our algorithm does not result in an explosion in communication or local memory usage, due to the following reasons:

1. Each monitor process explores a subset of all paths in the total order event lattice.
2. The algorithm does not attempt to collect information about the global state unless it results in automaton state change (ignores self-loop transitions).

The experimental results also show that the monitoring overhead grows only in the *linear order* of the number of processes and events that need to be monitored.

7.2 Future Work

Three are several open problems for further research and we elaborate on them next.

7.2.1 Augmented Time

In [9], the authors presented a technique for augmenting vector clocks with real time to enable better ordering of events. However, most distributed operating systems (e.g., a network of smartphones) attempt to keep real time to within 50 milliseconds of true time, so it would only be useful for applications that produce events with frequency less than 50 milliseconds. Therefore, this idea is more applicable to applications that run on devices that are not connected to NTP (Network Time Protocol), such as drones flying in the air using an ad-hoc network.

7.2.2 Automaton Static Analysis

This idea involves analyzing the automaton transitions and states in order to provide the monitoring algorithm with data that can help routing token messages to be more efficient. For example, if the monitoring algorithm knows that two paths in the automaton lead to the same final state, it would prioritize exploring the shorter path first.

7.2.3 Program Static Analysis

In this idea, the distributed program under monitoring is statically analyzed to detect local states from different processes that can never occur at the same time, thus transition with conjunctions of such states can be ignored and never explored.

7.2.4 Monitoring Algorithms for Dynamic Networks

Dynamic networks where processes can join and leave is particularly challenging in distributed systems, as the system is always changing and suffering from instability. Decentralized monitoring of such systems can be more challenging as the monitoring algorithm needs to be able to add and remove monitoring processes, and recover lost messages.

7.2.5 Monitoring Global Expressions

The idea basically is that a decentralized monitoring algorithm can verify a global arithmetic expressions or optimization objectives over the set of processes. For example, monitoring that a swarm of drones maximizes their inter-distance in a 3D plane.

References

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

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

[3] Andreas Bauer, Martin Leucker, and Christian Schallhart. Model-based runtime analysis of distributed reactive systems. In *Australian Software Engineering Conference (ASWEC'06)*, pages 10–pp. IEEE, 2006.

[4] Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime verification for ltl and tltl. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 20(4):14, 2011.

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

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

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

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

[9] M Demirbas and S Kulkarni. Beyond truetime: Using augmentedtime for improving google spanner. In *Workshop on Large-Scale Distributed Systems and Middleware (LADIS)*, 2013.

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

[11] Adrian Francalanza, Andrew Gauci, and Gordon J Pace. Distributed system contract monitoring. *J. Log. Algebr. Program.*, 82(5-7):186–215, 2013.

[12] Vijay K Garg, Neeraj Mittal, and Alper Sen. Applications of lattice theory to distributed computing. *ACM SIGACT Notes*, 34(3):40–61, 2003.

[13] Klaus Havelund and Grigore Roşu. Monitoring programs using rewriting. In *Automated Software Engineering, 2001.(ASE 2001). Proceedings. 16th Annual International Conference on*, pages 135–143. IEEE, 2001.

[14] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

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

[16] Sarah M Loos, André Platzer, and Ligia Nistor. Adaptive cruise control: Hybrid, distributed, and now formally verified. In *FM 2011: Formal Methods*, pages 42–56. Springer, 2011.

[17] Sarah M Loos, David Renshaw, and André Platzer. Formal verification of distributed aircraft controllers. In *Proceedings of the 16th international conference on Hybrid systems: computation and control*, pages 125–130. ACM, 2013.

[18] Thierry Massart and Cédric Meuter. Efficient online monitoring of ltl properties for asynchronous distributed systems. *Université Libre de Bruxelles, Tech. Rep*, 2006.

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

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

[21] Menna Mostafa and Borzoo Bonakdarpour. Decentralized runtime verification of ltl specifications in distributed systems. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 494–503. IEEE, 2015.

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

[23] Martin Ouimet and Kristina Lundqvist. Formal software verification: Model checking and theorem proving. *Embedded Systems Laboratory, MIT*, 2007.

[24] Lucia Pallottino, Vincenzo G Scordio, Antonio Bicchi, and Emilio Frazzoli. Decentralized cooperative policy for conflict resolution in multivehicle systems. *Robotics, IEEE Transactions on*, 23(6):1170–1183, 2007.

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

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

[27] Vijay D Silva, Daniel Kroening, and Georg Weissenbacher. A survey of automated techniques for formal software verification. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 27(7):1165–1178, 2008.