Decentralized Crash-Resilient Runtime Verification

DECENTRALIZED CRASH-RESILIENT RUNTIME VERIFICATION

BY

SHOKOUFEH KAZEMLOU, M.Sc.

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTING & SOFTWARE

AND THE SCHOOL OF GRADUATE STUDIES

OF MCMASTER UNIVERSITY

IN PARTIAL FULFILMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

Master of Applied Science (2017)	McMaster University
(Computing & Software)	Hamilton, Ontario, Canada

TITLE:	Decentralized Crash-Resilient Runtime Verification
AUTHOR:	Shokoufeh Kazemlou M.Sc.
SUPERVISOR:	Dr. Borzoo Bonakdarpour
NUMBER OF PAGES:	viii, 76

Abstract

Runtime Verification is a technique to extract information from a running system in order to detect executions violating a given correctness specification. In this thesis, we study distributed synchronous/asynchronous runtime verification of systems. In our setting, there is a set of distributed monitors that have only partial views of a large system and are subject to failures. In this context, it is unavoidable that monitors may have different views of the underlying system, and therefore may have different valuations of the correctness property. In this thesis, we propose an automata-based synchronous monitoring algorithm that copes with f crash failures in a distrbuted setting. The algorithm solves the synchronous monitoring problem in $f+1$ rounds of communication, and significantly reduces the message size overhead. We also propose an algorithm for distributed crash-resilient asynchronous monitoring that consistently monitors the system under inspection without any communication between monitors. Each local monitor emits a verdict set solely based on its own partial observation, and the intersection of the verdict sets will be the same as the verdict computed by a centralized monitor that has full view of the system.

Acknowledgements

Foremost, I would like to express my sincere gratitude to my supervisor Dr. Borzoo Bonakdarpour for the continuous support of my M.Sc. study and research, for his patience, motivation, and immense knowledge. I could not have imagined having a better mentor for my M.Sc. study. I am very grateful to Professor Ryszard Janicki and Professor Mark Lawford for graciously agreeing to be my examiners and for their invaluable comments and suggestions. I am also grateful for the funding and support provided by the Natural Sciences and Engineering Research Council of Canada (NSERC). Finally, I would like to thank my loved ones, who have supported me and encouraged me throughout my years of study and through the process of researching and writing this thesis.

Contents

List of Figures

Chapter 1

Introduction

In a computing system, *correctness* refers to the assertion that a system satisfies its specification. Software errors are an everyday problem. They occur for a variety of reasons like coding, hardware or network errors. There exist many techniques to ensure correctness in computing systems, most notably:

- Model checking is a fully automated push-button method to check that every execution path of the system under scrutiny satisfies its specification, usually given in some temporal logic formula.

- Theorem proving as a semi-automated way to rigorously demonstrate that the system under inspection complies with its specification usually defined in higher-order logic.

- Testing is a way to detect such failures by checking for the presence of faults for a set of executions.

Recently, there has been an emerging interest on monitoring a software system to discover bugs in obscure corner cases that can only be observed during online execution. Runtime verification (RV) refers to a technique, where a monitor checks at run time whether or not the execution of a system under inspection satisfies a given correctness property. RV complements exhaustive verification methods such as model checking and theorem proving, as well as incomplete solutions such as testing and debugging. Exhaustive verification often requires developing a rigorous abstract model of the system and suffers from the infamous *state-explosion problem*. Testing and debugging, on the other hand, provide us with under-approximated confidence about the correctness of a system as these methods only check for the presence of defects for a limited set of scenarios.

1.1 Decentralized Runtime Verification

Most existing RV methods employ a central monitor that collects the executions of all components and then checks the system's global behaviour in terms of a linear-time temporal logic (LTL) formula. The existing work on RV techniques where the monitor consists of a set of components, each having a partial view of the system, is limited to the following:

- Lattice-theoretic centralized and decentralized online predicate detection in distributed systems has been studied in Chauhan *et al.* (2013); Mittal and Garg (2005). However, this line of work does not address monitoring properties with temporal requirements. This shortcoming is addressed in Ogale and Garg (2007) for a fragment of temporal operators, but for offline monitoring and in Mostafa and Bonakdarpour (2015) for distributed monitoring of LTL specifications.

- In Sen *et al.* (2004), the authors design a method for monitoring safety properties in distributed systems using the past-time linear temporal logic (PLTL). This work, however, is not sound, meaning that valuation of some predicates and properties may be overlooked. This is because monitors gain knowledge about the state of the system by piggybacking on the existing communication among processes. That is, if processes rarely communicate, then monitors exchange very little information and, hence, some violations of properties may remain undetected.

- Runtime verification of LTL for synchronous distributed systems, where processes share a single global clock, has been studied in Bauer and Falcone (2016); Colombo and Falcone (2016).

Another short coming of existing RV methods is that they assume a fault-free setting, where each individual monitor is resilient to failures. In fact, handling monitors subject to failures, creates significant challenges specially in asynchronous monitoring, as local monitors would not be able to agree on the same perspective of the global system state, due to the impossibility of consensus Fischer *et al.* (1985). Therefore, it is inevitable that local monitors emit different local verdicts about the current run, and a consistent global verdict with respect to a correctness specification must be constructed from these verdicts. In this area, the work in the literature is limited to Bonakdarpour *et al.* (2016), where the authors propose a crash-resilient decentralized algorithm for monitoring LTL formulas in a wait-free setting.

This thesis studies the decentralized synchronous/asynchronous monitoring in a failure-prone environment, i.e., a faulty monitor stops executing prematurely. After it has crashed, a monitor does nothing. Before crashing, a monitor behaves correctly.

In our synchronous setting, we do not employ a centralized monitor to check the system's global behaviour. Rather, the satisfaction or violation of specifications can be detected by local monitors alone. We show that in a framework of several synchronous or asynchronous *unreliable* monitors, naive local monitoring may lead to inconsistent global verdicts for φ. More specifically, the set of verdicts emitted by the monitors may not be sufficient to distinguish executions that violate the formula from those that satisfy it. Intuitively, this is because each monitor has only a partial view of the system under inspection, and after a finite number of rounds of communication among monitors, still many different perspectives about the global system state remain.

1.2 Decentralized Synchronous Monitoring

In the decentralized synchronous setting, we assume the system under scrutiny generates a finite trace $\alpha = s_0 s_1 \cdots s_k$ and is inspected by a set of synchronous monitors $\mathcal{M} = \{M_1, M_2, \cdots, M_n\}$ with respect to a correctness property expressed by an LTL formula φ. The monitors communicate with each other by sending and receiving messages in *synchronous rounds*, and through point-to-point bidirectional reliable communication links. The decentralized crash-resilient synchronous monitoring can be reduced to the uniform consensus problem in the crash failure model.

Uniform Consensus in the Crash Failure Model In the consensus problem, each process proposes a value, and the processes have to collectively agree on the same value. Of course, a process can crash before deciding a value. Moreover, in order to be meaningful, the value that is decided has to be related to the values that are proposed. This is captured by the following specification.

- **Validity:** A decided value is a proposed value.
- **Agreement:** No two processes decide different values.
- **Termination:** Every correct process decides.

The validity and agreement properties define the safety property of the consensus problem. Validity relates the output to the inputs, while agreement captures the difficulty of the problem. Termination is the liveness property of the consensus problem. It states that at least the processes that do not crash have to decide. In a synchronous setting and in the presence of faults, consensus is solvable in $f + 1$ rounds, where f is the maximum number of processes than can crash.

In the synchronous monitoring problem, the validity specification is that the decided value must be the same value that a centralized monitor that has full view of the system would compute.

1.3 Decentralized Asynchronous Monitoring

In a decentralized asynchronous setting, there is a set of monitors $\mathcal{M} = \{M_1, M_2, \cdots, M_n\}$ that verify a finite trace $\alpha = s_0 s_1 \cdots s_k$ produced by the system under inspection, with respect to a correctness property φ. The monitors can communicate with each other via a read/write shared memory, they are asynchronous wait-free processes, and any of them can fail by crashing.

We work in the shared memory model because we can consider runs composed of any interleaving of monitor operations, facilitating analysis. Also to including the extreme case where any number of monitors may fail. In message passing partitions may happen if half of the monitors can fail. Given that in a wait-free distributed monitoring system it is impossible for the monitors to solve consensus, it is unavoidable that different verdicts are emitted. We show how, given any LTL formula, a set of verdicts collectively provided by the local monitors can be used to compute the verdict computed by a centralized monitor that has full view of the system under scrutiny.

Decentralized asynchronous monitoring can also be reduced to consensus in the asynchronous setting with the three properties mentioned in Section 1.2.

1.4 Thesis Statement

The main difficulty created by the crash failure model is the following. Suppose M is a monitor that crashes during a round r while it is broadcasting a message m. Since the broadcast operation by M is not atomic, and there is no predetermined sending order with respect to the destination monitors, a message m can be received by any subset of monitors. This subset is completely arbitrary, and can be the empty set. This means that the crash of a monitor in a send phase of a round generates non-determinism, as it is impossible to know what are the monitors that will receive message m. This uncertainty is the main challenge in designing a distributed monitoring algorithm in the presence of crash failure.

Our research hypothesis is that synchronous/asynchronous monitoring in fault-tolerant distributed environment can be performed efficiently. More specifically, we can design distributed algorithms that can solve the synchronous monitoring problem with relatively small message size overhead, and solve the asynchrnous monitoring problem with little communication between the monitors.

1.5 Contributions

We validate our research hypothesis by proposing an automata-based distributed LTL monitoring algorithm for the decentralized crash-resilient synchronous monitoring problem, where monitors are processes that communicate by message passing and can crash. Our monitoring algorithm reduces the message size overhead from $|AP|$ per message, where AP is a set of atomic propositions, to $\log(m_q)$, where m_q is the number of outgoing transitions from the current monitor state in each local monitor's automaton. Our main contribution is to construct an automaton that is employed in each local monitor's algorithm. This automaton, which is called an Extended LTL$_3$ monitor, is a deterministic finite state machine that is constructed based on the LTL$_3$ monitor Bauer *et al.* (2011) of a given LTL formula. The intuition behind constructing an Extended LTL$_3$ monitor is the idea of calculating the intersection of the verdict sets emitted by a set of distributed monitors that have partial view of the global system state. The extension ensures that monitors share enough detail about their local state, so that crash failures do not compromise their soundness.

Our second contribution is an algorithm for distributed crash-resilient asynchronous RV that can consistently monitor the system under inspection without any round of communication between asynchronous monitors. Each local monitor emits a verdict set solely based on its own partial observation, and the intersection of the verdict sets will be the same as the verdict computed by a centralized monitor that has full view of the system.

1.6 Thesis Organization

The rest of this thesis is organized as follows: Chapter 2 presents the related work and Chapter 3 provides the preliminary concepts. Chapter 4 discusses the synchronous monitoring in crash-resilient distributed environment and presents an algorithm to construct an Extended LTL_3 monitor which is the main contribution of this thesis. In Chapter 5 the decentralized crash-resilient asynchronous monitoring is introduced. Finally, in Chapter 6, we make concluding remarks and discuss future work.

Chapter 2

Literature Review

Runtime verification is now an established area of research with many applications. Research efforts in the literature of runtime verification expands on many different areas including theory and system development with even annual competition dedicated for efficient tool development Bartocci *et al.* (2018). The literature, however, is mainly focused on centralized systems or at least centralized monitors:

- The seminal technique on runtime verification Havelund and Rosu (2001b,a, 2004); d'Amorim and Rosu (2005); Chen and Rosu (2007); Havelund and Rosu (2002) uses formula rewriting. In particular, the monitor observes the incoming states of the system produced at run time and rewrites the formula using the newly observed events. As long as the formula does not result in logical true or false, the valuation of the formula can go either way in the future.

- Automata-based monitoring was first proposed in Bauer *et al.* (2011). The idea here is to synthesize an automaton from a given LTL formula that acts as a monitor. The work in Bauer *et al.* (2011) employs three truth values

$\{\top, \bot, ?\}$, where $\top$ (respectively, $\bot$) means that the formula is permanently satisfied (respectively, violated), and truth value '?' means that the valuation is currently unknown (i.e., both $\top$ and $\bot$ are possible for future extensions).

- More domain specific techniques have also been proposed. For example, in Bonakdarpour *et al.* (2011); Navabpour *et al.* (2011, 2015); Bonakdarpour *et al.* (2013), the authors propose *time-triggered* RV techniques for monitoring real-time systems, where schedulability is required. Monitoring security policies have always been an area of interest. The notion of security automata was first introduced in Schneider (2000) and revisited in Basin *et al.* (2013). Other efforts in the area of RV for security policies include the work on monitoring hyperproperties Agrawal and Bonakdarpour (2016); Brett *et al.* (2017); Finkbeiner *et al.* (2017) and on monitoring compliance policies Basin *et al.* (2016). Finally, monitoring cyber-physical systems, where computing systems interact with physical environments has been studied in Nguyen *et al.* (2017); Deshmukh *et al.* (2015), by monitoring formulas in the signal temporal logic (STL) and its variants, and in Medhat *et al.* (2015) by monitoring under resource constraints.

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

2.1 Lattice-based Distributed Monitoring

Lattice theory has been an important tool for design and analysis of distributed algorithms Garg (2002). The notion of *happened before* by Lamport Lamport (1978) basically defines a partial order of events, which essentially constructs a distributive lattice. Such a lattice is often called a *computation*. Each node in the lattice is essentially a possible state of the distributed system.

Predicate detection is the problem of identifying states of a distributed computation that satisfy a predicate. Detecting a predicate in a computation is a challenging problem Garg (2002); Stoller and Schneider (1995). The reason is the combinatorial blow up in the number of possible states. Given n processes each with k local states, the number of possible global states in the computation could be as large as $O(k^n)$. Determining whether a global state satisfies the given predicate may, therefore, require looking at a large number of consistent cuts. In fact, it is shown that the problem is in general NP-complete Mittal and Garg (2001). *Computation slicing* Mittal and Garg (2005) is a technique for reducing the size of the computation and, hence, the number of global states to be analyzed for detecting a predicate. The slice of a computation with respect to a predicate is the (sub)computation satisfying the following two conditions. First, it contains all global states for which the predicate evaluates to true. Second, among all computations that satisfy the first condition, it contains the least number of consistent cuts. Intuitively, slice is a concise representation of consistent cuts satisfying a given property. In Mittal and Garg (2005), the authors propose an algorithm for detecting regular predicates. This idea was then extended to full blown distributed algorithm for distributed monitoring Chauhan *et al.* (2013). One shortcoming of this line work is that it does not address monitoring properties with temporal requirements. This shortcoming is addressed in Ogale and Garg (2007) for a fragment of temporal operators. In Mostafa and Bonakdarpour (2015), the authors 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. The technique for evaluating LTL properties is inspired by distributed computation slicing described above. The monitoring technique is fully decentralized in that each process in the distributed program under inspection maintains a replica of the monitor automaton. Each monitor may maintain a set of possible verification verdicts based upon existence of concurrent events. LTL formulas in this work are in terms of conjunctive predicates; i.e., predicates that are conjunctions are local propositions of processes.

One of the problems with lattice-based techniques, especially when it comes to temporal properties in distributed LTL monitoring, is that if processes communicate rarely, then the computation will include many concurrent global states and the lattice will become very wide. To tackle this problem in Yingchareonthawornchai *et al.* (2016), the authors proposed an algorithm and analytical bounds if a combination of logical and physical clocks (called *hybrid* clocks) are used. This method is enriched with SAT solving techniques in Valapil *et al.* (2017).

2.2 Distributed Monitoring for Past-time LTL

In Sen *et al.* (2004), the authors propose a decentralized monitoring algorithm that monitors a distributed program with respect to safety properties in PT-DTL, a variant of past time linear temporal logic. PT-DTL is designed in Sen *et al.* (2004) to express temporal properties of distributed systems by drawing relation to particular process and their knowledge of the local state of other processes at any point in time. In the monitoring algorithm, monitors gain knowledge about the state of the system by piggybacking on the existing communication among processes. In such a framework, the valuation of some predicates and properties may be overlooked. That is, if processes rarely communicate, then monitors exchange little information and, hence, some violations of properties may remain undetected. This method was then used to design an algorithm for monitoring multi-threaded programs Sen *et al.* (2006).

2.3 Synchronous Distributed Monitoring

The most relevant work to this thesis is arguably the algorithms proposed in Bauer and Falcone (2016); Colombo and Falcone (2016). The algorithm in Bauer and Falcone (2016) for monitoring synchronous distributed systems with respect to LTL formulas is designed such that satisfaction or violation of specifications can be detected by local monitors alone. The framework employs a different alphabet for each process in the system. When a local monitor has a part of the formula that it cannot evaluate, it sends a message to the monitor that hosts the subformula. The subformula is then progressed over synchronous rounds and the monitor generates a suformula that represents the evaluation of the monitor in the respective round number. The authors also present an implementation and show that our algorithm introduces only a negligible delay in detecting satisfaction/violation of a specification. Moreover, the practical results show that the communication overhead introduced by the local monitors is generally lower than the number of messages that would need to be sent to a central data collection point.

In Colombo and Falcone (2016), the authors introduce a way of organizing submonitors for LTL subformulas in a synchronous distributed system, called *choreography.* In particular, the monitors are organized as a tree across the distributed system, and each child feeds intermediate results to its parent in a manner similar to diffusing computation. They formalize choreography-based decentralized monitoring by showing how to synthesize a network from an LTL formula, and give a decentralized monitoring algorithm working on top of an LTL network.

These articles are different from this thesis in the following ways: (1) the framework in in Bauer and Falcone (2016); Colombo and Falcone (2016) is fault-free, and (2) in this thesis we do not assume that components have disjoint propositions.

2.4 Fault-tolerant Distributed Monitoring

In Fraigniaud *et al.* (2013), the authors consider the problem of whether in a distributed task, one can determine the set of inputs and outputs satisfy their intended specification in a wait-free setting subject to crash faults. They call this problem *checkability* which is a special instance of runtime monitoring for simple safety specifications. They show that this problem is as hard as solving consensus. They extend their results in Fraigniaud *et al.* (2014a,b) and show that if runtime monitors employ multiple *opinions* (instead of the conventional false/true valuations), then it is possible to monitor distributed tasks in a consistent manner.

Building on the work in Fraigniaud *et al.* (2013, 2014a,b), the authors in Bonakdarpour *et al.* (2016) show that employing the LTL four-valued logic Bauer *et al.* (2010) will result in inconsistent distributed monitoring for some formulas. The first main contribution of this work is a family of logics, called LTL_{2k+4}, that refines the 4-valued LTL incorporating $2k + 4$ truth values, for each $k \geq 0$. The truth values of LTL_{2k+4} can be effectively used by each monitor to reach a consistent global set of verdicts for each given formula, provided k is sufficiently large. The second contribution of this work is an algorithm for monitor construction enabling fault tolerant distributed monitoring based on the aggregation of the individual verdicts by each monitor.

Chapter 3

Preliminaries

In this chapter, we review the preliminary concepts.

3.1 Linear Temporal Logics

Let AP be a set of *atomic propositions* and $\Sigma = 2^{AP}$ be the *alphabet*. We call each element of Σ an *state*. A *trace* is a sequence $s_0 s_1 \cdots$, where $s_i \in \Sigma$ for every $i \geq 0$. The set of all finite (resp., infinite) traces over Σ is denoted by Σ^* (resp., Σ^ω). Throughout the thesis, we denote finite traces by the letter α, and infinite traces by the letter σ. We denote the empty trace by ϵ. Finally, For a finite trace $\alpha = s_0 s_1 \cdots s_n$, by α^i we mean trace suffix $s_i s_{i+1} \cdots s_n$ of α.

LTL Syntax. Formulas in *linear temporal logic* (LTL) Manna and Pnueli (1979) are defined using the following grammar:

$$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid \mathbf{X}\varphi \mid \varphi \, \mathbf{U} \, \varphi$$

where $p \in AP$ is an atomic proposition, $\mathbf{U}$ is the "until" operator, and $\mathbf{X}$ is the "next operator". Additionally, we allow the following operators as syntactic sugar, each of which is defined in terms of the above ones: $\mathsf{true} = p \vee \neg p, \mathsf{false} = \neg\mathsf{true}, \varphi_1 \wedge \varphi_2 = \neg(\neg\varphi_1 \vee \neg\varphi_2), \mathbf{F}\varphi = \mathsf{true}\,\mathbf{U}\,\varphi$,and $\mathbf{G}\varphi = \neg\mathbf{F}(\neg\varphi)$. Formulas without temporal operators are called state formulas.

LTL Semantics. The semantics of LTL is defined w.r.t. infinite traces. Let $\sigma = s_0 s_1 \cdots$ be an infinite trace in Σ^ω, $i \geq 0$, and $\models$ denote the *satisfaction* relation. The semantics of LTL is defined as follows:

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

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

$$\varphi_{ra} = \mathbf{G}\Big(\neg a \wedge \neg r\Big) \;\vee\; \Big((\neg a\,\mathbf{U}\,r) \;\wedge\; \mathbf{F}a\Big)$$

This formula requires that (1) if a request is emitted (i.e., $r = \mathsf{true}$), then it should eventually be acknowledged (i.e., $a = \mathsf{true}$), and (2) an acknowledgment happens only in response to a request.

3.2 LTL for Runtime Verification

3.2.1 Finite LTL (FLTL)

The semantics of LTL is defined over infinite traces. In the context of runtime verification, since a system only generates finite traces, the standard LTL semantics does not seem to be the appropriate formalism. Finite LTL (denoted FLTL Manna and Pnueli (1995)) allows us to reason about finite traces for verifying properties at run time. The syntax of FLTL is identical to that of LTL and the semantics is based on the truth values $\mathbb{B}_2 = \{\top, \bot\}$. The semantics of FLTL for atomic propositions and Boolean operators are identical to those of LTL. We now recall the semantics of FLTL for the temporal operators. Let φ, φ_1, and φ_2 be LTL formulas, $\alpha = s_0 s_1 \cdots s_n$ be a non-empty finite trace, and $\models_F$ denote satisfaction in FLTL. We have

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

and

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

To illustrate the difference between LTL and FLTL, let $\varphi = \mathbf{F}p$ and $\alpha = s_0 s_1 \cdots s_n$. If $p \in s_i$ for some $i \in [0,n]$, then we have $[\alpha \models_F \varphi] = \top$. Otherwise, $[\alpha \models_F \varphi] = \bot$, and this holds even if the program under inspection extends α in the future to a state where p becomes true.

3.2.2 3-valued LTL

As illustrated above, for a finite trace α, FLTL ignores the possible future extensions of α, when evaluating a formula. The 3-valued semantics of LTL (denoted LTL$_3$ Bauer *et al.* (2011)) evaluates LTL formulas for finite traces with an eye on possible future extensions. In LTL$_3$, the set of truth values is $\mathbb{B}_3 = \{\top, \bot, ?\}$, where '$\top$' (resp., '$\bot$') denotes that the formula is permanently satisfied (resp., violated), no matter how the current execution extends, and '?' denotes an unknown truth value; i.e., there exist an extension that can falsify the formula, and another extension that can truthify the formula.

Now, let $\alpha \in \Sigma^*$ be a non-empty finite trace. The truth value of an LTL$_3$ formula φ with respect to α, denoted by $[\alpha \models_3 \varphi]$, is defined as follows:

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

The LTL$_3$ *monitor* of a formula φ is the unique deterministic finite state machine $\mathcal{M}_3^\varphi = \{\Sigma, Q, q_0, \delta, \lambda\}$, where Q is a set of states, q_0 is the initial state, $\delta : Q \times \Sigma \rightarrow Q$ is the transition function, and $\lambda : Q \rightarrow \mathbb{B}_3$, is a function such that:

$$\lambda(\delta(q_0, \alpha)) = [\alpha \models_3 \varphi]$$

for every finite trace $\alpha \in \Sigma^*$. In Bauer *et al.* (2011), the authors introduce an algorithm that takes as input an LTL formula and constructs as output an LTL$_3$ monitor. For example, Fig. 3.1 shows the LTL$_3$ monitor for the LTL formula $\varphi =$

$a \, \mathbf{U} \, b$.

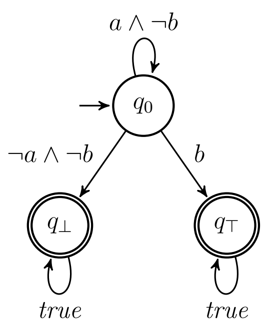


Figure 3.1: LTL$_3$ monitor for $\varphi = a \, \mathbf{U} \, b$.

3.2.3 RV-LTL

RV-LTL Bauer *et al.* (2010), which we will denote in this thesis by LTL$_4$, refines the truth value ? into $\perp_p$ and $\top_p$. That is, $\mathbb{B}_4 = \{\top, \top_p, \perp_p, \perp\}$. More specifically, evaluation of a formula in LTL$_4$ agrees with LTL$_3$ if the verdict is $\perp$ or $\top$. Otherwise, (i.e., when the verdict in LTL$_3$ is ?), LTL$_4$ utilizes FLTL to compute a more refined truth value.

Now, let $\alpha \in \Sigma^*$ be a finite trace. The truth value of an LTL$_4$ formula φ with respect to α, denoted by $[\alpha \models_4 \varphi]$, is defined as follows:

$$[\alpha \models_4 \varphi] = \begin{cases} \top & \text{if} \quad [\alpha \models_3 \varphi] = \top \\ \perp & \text{if} \quad [\alpha \models_3 \varphi] = \perp \\ \top_p & \text{if} \quad [\alpha \models_3 \varphi] = ? \ \wedge \ [\alpha \models_F \varphi] = \top \\ \perp_p & \text{if} \quad [\alpha \models_3 \varphi] = ? \ \wedge \ [\alpha \models_F \varphi] = \perp \end{cases}$$

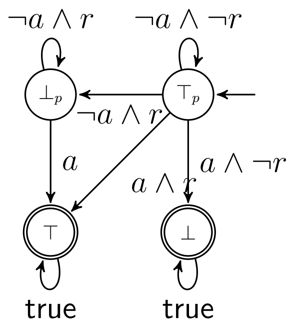


Figure 3.2: LTL$_4$ monitor of φ_{ra}.

The LTL$_4$ *monitor* of a formula φ is the unique deterministic finite state machine $\mathcal{M}_4^\varphi = \{\Sigma, Q, q_0, \delta, \lambda\}$, where Q is a set of states, q_0 is the initial state, $\delta : Q \times \Sigma \to Q$ is the transition function, and $\lambda : Q \to \mathbb{B}_4$, is a function such that:

$$\lambda(\delta(q_0, \alpha)) = [\alpha \models_4 \varphi]$$

for every finite trace $\alpha \in \Sigma^*$. In Bauer *et al.* (2011) , the authors introduce an algorithm that takes as input an LTL formula and constructs as output an LTL$_4$ monitor. For example, Fig. 3.2 shows the LTL$_4$ monitor for the request/acknowledgement formula $\varphi_{ra} = \mathbf{G}(\neg a \wedge \neg r) \;\vee\; [(\neg a \,\mathbf{U}\, r) \;\wedge\; \mathbf{F}a]$.

Chapter 4

Decentralized Automata-Based Monitoring

An LTL$_3$ monitor can evaluate an LTL formula φ in a centralized setting where each proposition represents the global state of the system. We show in Section 4.1 and Chapter 5 that in a framework of several synchronous or asynchronous *unreliable* monitors, naive local monitoring may lead to inconsistent global verdicts for φ. More specifically, the set of verdicts emitted by the monitors may not be sufficient to distinguish executions that violate the formula from those that satisfy it. Intuitively, this is because each monitor has only a partial view of the system under inspection, and after a finite number of rounds of communication among monitors, still many different perspectives about the global system state remain. We use the LTL formula $\varphi = \mathbf{F}(a \wedge b)$ throughout this Chapter to explain the concepts.

This Chapter is organized as follows: We discuss the synchronous monitoring problem in a failure-prone distributed environment in Section 4.1. The model of computation and terminology are discussed in Section 4.2. The problem statement is given in Section 4.3, and in Section 4.4 we discuss the challenges in synchronous monitoring. The synchronous automata-based monitoring is discussed in Section 4.5 where we introduce our automata-based monitoring algorithm, and present the algorithm to construct an Extended LTL_3 monitor.

4.1 Synchronous Monitoring Algorithm Sketch

In this section, we propose a framework for synchronous distributed fault-tolerant runtime verification (RV). To this end, we make a link between RV and consensus in a failure-prone distributed environment by proposing an automata-based algorithm.

We consider a distributed monitoring system made up of a fixed number n of monitors $\mathcal{M} = \{M_1, M_2, \ldots, M_n\}$ that communicate by sending and receiving messages through point-to-point bidirectional communication links. Each communication link is reliable, that is, we assume no loss or alteration of messages. Each monitor locally executes an identical sequential algorithm. Each run of a monitor consists of a sequence of rounds that are identified by the successive integers 1, 2, etc. The round number is a global variable and its progress is ensured by the synchrony assumption. Each round is made up of three consecutive steps: *send*, *receive*, and *local computation*. The principle property of the round-based synchronous model is the fact that a message sent by a monitor M_i to another monitor M_j during a round r is received by M_j at the very same round r.

Throughout this chapter, the system under inspection produces a finite trace $\alpha = s_0 s_1 \cdots s_k$, and is inspected with respect to an LTL formula φ by a set of synchronous distributed monitors.

Algorithm sketch: For every $j \in [0, k-1]$, between each s_j and s_{j+1}, each monitor:

1. reads the value of a subset of propositions in s_j, which may result in a *partial* observation of s_j;
2. at every synchronous round, *broadcasts* a message containing its current observation of the underlying system, and then waits for messages from other monitors;
3. based on the messages received at each round, executes a local computation, updates its current observation by incorporating observations of other monitors, and composing the message to be sent at next round, and
4. finally evaluates φ at the end of communication rounds and subsequently emits a truth value from $\mathbb{B}_3$.

The skeleton of the algorithm is shown in Algorithm 1.

```
Data: LTL formula $\varphi$ and state $s_j$
Result: a verdict from $\mathbb{B}_3$
1 Let $\mathcal{S}_i^{s_j}$ be the initial concrete local state of the monitor
2 $LS_i^1 \leftarrow \mu(\mathcal{S}_i^{s_j}, \varphi)$ ;   /* computes the initial abstract local state based on the initial
    concrete local state */
3 for $r = 1, 2, \cdots$ do
4     Send: broadcasts its current abstract local state $LS_i^r$ ; /* r is the round number
      */
5     Receive: let $\Pi_i^r = \{LS_j^r\}_{j \in [1,n]}$ be the set of all messages received at round $r$.
6     Computation: $LS_i^{r+1} \leftarrow LC(\Pi_i^r)$ ;   /* calculates a new abstract local state */
7 emits a verdict from $\mathbb{B}_3$ ; /* evaluates $\varphi$ according to the final abstract local state */
```

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

4.2 Model of Computation and Terminology

We now present our computation model, notation, and terminology.

Definition 4.1. *A 'concrete local state' $\mathcal{S}_i^{s_j}$ of a monitor M_i at global state s_j is a mapping from the set AP of atomic propositions to the set $\{\mathsf{true}, \mathsf{false}, \natural\}$, where $\natural$ denotes an unknown value, and for all $ap \in AP$, we have:*

$$(\mathcal{S}_i^{s_j}(ap) = \mathsf{true} \rightarrow ap \in s^j) \wedge (\mathcal{S}_i^{s_j}(ap) = \mathsf{false} \rightarrow ap \notin s_j)$$

When a state s_j is reached in a finite trace $\alpha = s_0 s_1 \cdots s_k$, each monitor $M_i \in \mathcal{M}$, for $1 \leq i \leq n$, takes a sample from s_j, which results in obtaining a concrete local state $\mathcal{S}_i^{s_j}$. Hence, in the concrete local state of a monitor, if the value of an atomic proposition is not unknown, then its value is consistent with state s_j. Thus, two monitors M_i and M_l cannot have inconsistent concrete local states. That is, for any state s_j and concrete local states $\mathcal{S}_i^{s_j}$, $\mathcal{S}_l^{s_j}$, and for every $ap \in AP$, we have:

$$(\mathcal{S}_i^{s_j}(ap) \neq \mathcal{S}_l^{s_j}(ap) \ \rightarrow \ (\mathcal{S}_i^{s_j}(ap) = \natural \vee \mathcal{S}_l^{s_j}(ap) = \natural)$$

Definition 4.2. *An 'abstract local state' LS_i is a symbolic representation of a monitor M_i's concrete local state $\mathcal{S}_i^{s_j}$ with respect to an* LTL *formula φ computed by an 'abstraction function' μ, where $LS_i = \mu(\mathcal{S}_i^{s_j}, \varphi)$.*

Note that Definition 4.2 does not prescribe a specific symbolic representation or abstraction function. We will present a choice for this function in Section 4.5. The idea here is that monitors communicate their abstract local states rather than concrete local states for space and communication efficiency. During the computation step, the monitor computes the message that it will to broadcast during the next round. Let LS_i^r denote the abstract local state of M_i at the beginning of round r. In the local computation step, a monitor M_i modifies its abstract local state according to the messages it has received from other monitors (including its own message). Let

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

be the set of all messages received by monitor M_i during round r.

Definition 4.3. *The 'local computation function' of a monitor M_i is a function LC that computes M_i's new abstract local state in each round r, given the set of messages Π_i^r received in round r. Formally,*

$$LS_i^{r+1} = LC(\Pi_i^r)$$

Similar to abstraction function, we will describe local computation function in Section 4.5.

State Coverage: We say that a set of monitors *cover* a global state if and only if the collection of concrete local states of these monitors covers the value of all atomic propositions. The formal definition is given below.

Definition 4.4. *A set $\mathcal{M} = \{M_1, M_2, \ldots, M_n\}$ satisfies 'state coverage' for a state s if and only if for every $ap \in AP$, there exists $M_i \in \mathcal{M}$ such that $\mathcal{S}_i^s(ap) \neq \natural$.*

Definition 4.5. *We say monitor M_i is 'aware' of proposition ap at round r, if:*

- *$\mathcal{S}_i^s(ap) \neq \natural$, or*
- *M_i receives a message from a monitor M_j at round $r' \in [1, r)$, where M_j is aware of ap at round r'.*

Fault Model: In our setting, the fault model specifies that each monitor may fail by *crashing* (i.e., halt and never recover). We assume that up to $n-1$ monitors can crash, where $n = |\mathcal{M}|$. A monitor may crash at any round. To ensure the state coverage, we assume that, if there is a proposition $ap \in AP$, such that at round r monitor M_i is the only monitor aware of ap, then the message sent by M_i at round r, must be received by at least one non-faulty monitor in round r.

Note that in order to weaken the latter condition, one might assume that each proposition $ap \in AP$ is read by sufficiently large number of local monitors such that it is ensured that at least one monitor which is aware of ap does not crash, e.g., by assuming that each proposition is read by at least $f + 1$ monitors.

4.3 Problem Statement

Suppose $\alpha = s_0 s_1 \cdots s_k$ is a finite trace generated by the system under inspection, and φ is an LTL formula with respect to which we monitor the system. Each monitor $M_i \in \mathcal{M}$, $i \in [1, n]$, runs Algorithm 1 as follows. For any given new state s_j, monitor M_i first obtains an initial concrete local state by taking a sample from s_j (cf. Line 1). Recall from Definition 4.1 that the value of an atomic proposition in a concrete local state is either true, false, or $\natural$. After obtaining the initial concrete local state, monitor M_i computes the initial local state based on the initial concrete local state (cf. Line 2). After intialization, each monitor M_i executes a sequence of send, receive, and computation actions (cf. Lines 4-6) for some a priori known number of rounds. In Line 4, monitor M_i sends its current abstract local state to all other monitors in $\mathcal{M}$. In Line 5, it receives messages from other monitors and stores them (along with its own message) in a set Π_i^r. In line 6, which is the computation step, monitor M_i computes and updates its abstract local state based on messages in Π_i^r. Finally, after a certain number of rounds, the for-loop ends, and M_i evaluates φ and emits a truth value from $\mathbb{B}_3$ based on its final abstract local state (cf. Line 7). Note that Algorithm 1 is executed whenever a new global state is reached in α.

Our formal problem statement is the termination requirement for Algorithm 1. Roughly speaking, we require that when a non-faulty monitor runs Algorithm 1 to the end, it should compute and emit a verdict that a centralized monitor that has global view of the system would compute. This termination condition is formally, the following

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

where ν_i is the truth value emitted by monitor M_i at the end of running Algorithm 1.

4.4 Challenges in Synchronous Monitoring

It is easy to see that our decentralized synchronous monitoring problem, described in Section 4.3, is similar to the uniform consesus problem that was described in Section 1.2. It is also straightforward to verify that the lower bound on the number of rounds required to consistently monitor the system is $f + 1$, where f is the total number of crashes the system can tolerate. The proof would be similar to the proof of the lower bound on the number of rounds required for the consensus algorithm that copes with f process crashes.

Compared to the processing capacity of monitors, the communication links are low bandwidth, and hence, the communication costs are of concern. The communication cost depends on the number of messages transmitted by monitors, and the size of these messages. An increase in the message size enforced by the algorithm is referred to as the message size overhead. And an increase in the number of messages that must be transmitted is called the network traffic overhead.

The following example illustrates the worst case scenario in which $f+1$ rounds are required to distributedly monitor the system, where f is the total number of faults tolerated. It also shows how message size can dramatically increase with the state space of the system under inspection.

Example: Let $\varphi = \mathbf{F}(a \wedge b)$, $AP = \{a, b\}$, and $\mathcal{M} = \{M_1, M_2, M_3, M_4\}$. Suppose $s = \{a, b\}$ is the current global state of the system, and the initial concrete local states of the monitors are as follows:

$$\begin{array}{ll}
\mathcal{S}_1^s[1](a) = \mathsf{true} & \mathcal{S}_1^s[1](b) = \natural \\
\mathcal{S}_2^s[1](a) = \natural & \mathcal{S}_2^s[1](b) = \mathsf{true} \\
\mathcal{S}_3^s[1](a) = \natural & \mathcal{S}_3^s[1](b) = \natural \\
\mathcal{S}_4^s[1](a) = \natural & \mathcal{S}_4^s[1](b) = \natural
\end{array}$$

where $\mathcal{S}_i^s[r]$ represents the concrete local state of monitor M_i at the begining of round r. Let $f = 2$, i.e., at most 2 monitors may crash, and suppose M_1 and M_2 are faulty monitors. The worst case scenario is when one and only one monitor crashes at each round. Suppose M_1 crashes at round 1 and M_2 crashes at round 2. Since M_1 is the only monitor that is aware of proposition a, according to our failure model assumption (in order to preserve the state coverage), at least one non-faulty monitor must receive a message from M_1 at round 1. Let M_2 be the monitor that receives M_1's message at round 1.

According to Algorithm 1, the monitors are to broadcast their current abstract local states at each round. In this case, let the abstract local state of each monitor be the same as its concrete local state, i.e., $LS_i^r = \mu(\mathcal{S}_i^s[r], \varphi) = \mathcal{S}_i^s[r]$ where $\mathcal{S}_i^s[r]$ is the concrete local state of monitor M_i at round r. In this case, each message sent by a monitor is a *register* that consists of $|AP|$ elements, one for each atomic proposition in AP. At the end of each round, each monitor M_i reads the values of all received messages and copies them into its local register as follows:

$$\forall p \in AP : ((\mathcal{S}_i^s[r](p) = \natural) \wedge (\exists j \in [1, n] : \mathcal{S}_j^s[r](p) \neq \natural)) \rightarrow (\mathcal{S}_i^s[r](p) \leftarrow \mathcal{S}_j^s[r](p))$$

Hence, at the end of round 1 (i.e., begining of round 2), the concrete local states are as follows:

$$\begin{array}{ll} \mathcal{S}_2^s[2](a) = \mathsf{true} & \mathcal{S}_2^s[2](b) = \mathsf{true} \\ \mathcal{S}_3^s[2](a) = \natural & \mathcal{S}_3^s[2](b) = \mathsf{true} \\ \mathcal{S}_4^s[2](a) = \natural & \mathcal{S}_4^s[2](b) = \mathsf{true} \end{array}$$

Suppose monitor M_2 crashes at round 2 and since it is the only monitor that is currently aware of proposition a, at least one non-faulty monitor must receive a message from M_2 at round 2, suppose M_3 is the monitor which receives the message. Thus, the new concrete local states at the end of round 2 are as follows:

$$\begin{array}{ll} \mathcal{S}_3^s[3](a) = \mathsf{true} & \mathcal{S}_3^s[3](b) = \mathsf{true} \\ \mathcal{S}_4^s[3](a) = \natural & \mathcal{S}_4^s[3](b) = \mathsf{true} \end{array}$$

Finally, at round 3, since there is no faulty monitor, each monitor receives messages from all other monitors, and the concrete local states at the end of this round will be as follows:

$$\begin{array}{ll} \mathcal{S}_3^s[4](a) = \mathsf{true} & \mathcal{S}_3^s[4](b) = \mathsf{true} \\ \mathcal{S}_4^s[4](a) = \mathsf{true} & \mathcal{S}_4^s[4](b) = \mathsf{true} \end{array}$$

Hence, at the end of round 3 (namely., round $f+1$) all non-faulty monitors are aware of all propositions in AP, and they emit the correct truth value:

$$\nu_3 = \nu_4 = [\{a, b\} \models_3 \varphi] = \top$$

The following tables summarize the scenario:

sample

	a	b
M_1	true	$\natural$
M_2	$\natural$	true
M_3	$\natural$	true
M_4	$\natural$	true

round 1

	a	b
M_1	crashed	crashed
M_2	true	true
M_3	$\natural$	true
M_4	$\natural$	true

round 2

	a	b
M_1	crashed	crashed
M_2	crashed	crashed
M_3	true	true
M_4	$\natural$	true

round 3

	a	b
M_1	crashed	crashed
M_2	crashed	crashed
M_3	true	true
M_4	true	true

One can see in the above example, in case each monitor broadcasts its concrete local state, namely, if the abstract local state is the same as the concrete local state, then each message sent by a monitor is a register that consists of $|AP|$ elements, one for each atomic proposition in AP. Our goal is to decrease the message size overhead, hence we introduce an algorithm that decreases the message size from $|AP|$ bits to something significantly lower. In Section 4.5, we introduce an algorithm which decreases the message size overhead in synchronous distributed monitoring. The algorithm solves the synchronous distributed monitoring problem in $f + 1$ rounds of communication with message size of $\log(m_q)$, where m_q is the number of outgoing transitions from monitor state q in an Extended LTL$_3$ monitor that will be introduced in Section 4.5.3.

4.5 Synchronous Automata-based Monitoring

In this section, we introduce an automata-based algorithm that solves the decentralized synchronous monitoring problem in $f+1$ rounds of communication, where f is the maximum number of crash failures tolerated. To this end, first, in Section 4.5.1, we introduce an algorithm that is used by each local monitor to synchronously monitor the system under inspection. The general idea is that each local monitor evaluates the input formula and computes a *possible* set of verdicts, as a monitor may not know the value of all propositions. Then, through synchronous communication, the monitors share their verdict sets with each other. Finally, applying some function on the verdicts sets (e.g., computing their intersection) computes the verdict that a centralized monitor that has the global view of the system would compute.

Our algorithm uses LTL$_3$ monitor $\mathcal{M}^\varphi$ for a formula φ in order to generate the set of possible verdicts. Thus, in our first attempt, in Section 4.5.1, we use the LTL$_3$ monitors in our algorithm, but we also show that $\mathcal{M}^\varphi$, as defined in Section 3.2.2, is not sufficient to consistently monitor the system for LTL formulas. Then, in Section 4.5.3, we introduce an 'Extended LTL$_3$ monitor' which we will use in each local monitor's algorithm to consistently monitor the global state of the system with respect to any LTL formula.

Note that in this work, we only consider monitoring of properties that are specified as LTL formulas, as Pnuelis LTL Pnueli (1977) is a well-accepted linear-time temporal logic for specifying properties of infinite traces. However, In runtime verification, our goal is to check LTL properties given finite prefixes of infinite traces. Therefore, one has to interpret their semantics with respect to finite prefixes as they arise in observing actual systems. Although it is possible to use infinite semantics of LTL, namely by using nondeterministic Büchi automata as monitors and explore all nondeterministic choices, it is more convenient to use the finite semantics of LTL to monitor LTL formulas at run time. To this end, in Bauer *et al.* (2011) introduced LTL$_3$ as a linear-time temporal logic which has the same syntax as in LTL but deviates in its semantics for finite traces. To implement the idea that, for a given LTL$_3$ formula, its meaning for a prefix of an infinite trace must correspond to its meaning considered as an LTL formula for the full infinite trace, they use three truth values: true, false, and inconclusive, denoted respectively by $\top$, $\bot$, and ?.

4.5.1 Synchronous Monitoring Using LTL$_3$ Monitors

Recall that an LTL$_3$ monitor $\mathcal{M}^\varphi$ for LTL formula φ is a deterministic finite state machine (FSM) represented as $\mathcal{M}^\varphi = \{\Sigma, Q, q_0, \delta, \lambda\}$, where Σ is a finite alphabet, Q is a finite non-empty set of states (we refer to them as monitor states), q_0 is the initial monitor state, $\delta : Q \times \Sigma \to 2^Q$ is a transition function, and $\lambda : Q \to \mathbb{B}_3$ is a mapping function which maps each monitor state to a truth value in $\mathbb{B}_3 = \{\top, \bot, ?\}$.

Observe that a transition t^i_j from monitor state q_i to monitor state q_j is the set of all states $s \in \Sigma$ such that $\delta(q_i, s) = q_j$. More formally

$$t^i_j = \{s \in \Sigma \mid \delta(q_i, s) = q_j\}$$

Now, let $\mathcal{M}^\varphi$ be the LTL_3 monitor for an LTL formula φ. We denote by T_q the set of all outgoing transitions from monitor state q. Formally, $T_q = \{t_1^q, \cdots, t_{m_q}^q\}$ where m_q is the total number of outgoing transitions from monitor state q and each t_j^q is an outgoing transition from monitor state q. Given that an LTL_3 monitor is a deterministic FSM, the followings hold

- $\forall j \in [1 \cdots m_q] : t_j^q \subseteq \Sigma$,
- $\forall j, k \in [1 \cdots m_q] : j \neq k \Rightarrow\ t_j^q \cap t_k^q = \emptyset$, and
- $t_1^q \cup \cdots \cup t_{m_q}^q = \Sigma$.

Let $\mathcal{S}_i^s$ be the concrete local state of a monitor M_i at global state s. We denote the set of all possible global states from the viewpoint of monitor M_i by $E(\mathcal{S}_i^s)$:

$$E(\mathcal{S}_i^s) = \big\{s' \in \Sigma \mid \forall ap \in AP : (\mathcal{S}_i^s(ap) \neq \natural) \rightarrow ((\mathcal{S}_i^s(ap) = \mathsf{true} \rightarrow ap \in s') \wedge (\mathcal{S}_i^s(ap) = \mathsf{false} \rightarrow ap \notin s'))\big\}$$

Informally, $E(\mathcal{S}_i^s)$ is the set of all states $s' \in \Sigma$ that are possible to be the global state s, from viewpoint of monitor M_i. Obviously, for any global state s, we have:

$$\forall i \in [1, n].\ s \in E(\mathcal{S}_i^s)$$

Now, suppose LTL_3 monitor M_i is at state q, and $\mathcal{S}_i^s$ is M_i's concrete local state. We denote the set of *possible verdicts* by monitor M_i as follows

$$V_i = \{\delta(q, s') \mid s' \in E(\mathcal{S}_i^s)\}$$

It should be noted that each verdict v_j from a verdict set V_i is a monitor state. The mapping function λ shall be applied to obtain the truth value of a verdict, i.e.,

$\lambda(v_j) \in \mathbb{B}_3$. Moreover, note that due to the synchrony assumption, all monitors always are at the same monitor state. Obviously, if all monitors are at monitor state q and the new global state of the system is s, then we have

$$\forall i \in [1, n].\ \delta(q, s) \in V_i$$

Notation: Let q be the current monitor state. We denote by $\mathcal{I}^q$ the intersection of all verdict sets emitted by all monitors in $\mathcal{M}$. Formally

$$\mathcal{I}^q = \bigcap_{i=1}^{n} V_i$$

Obviously, at every monitor state $q \in Q$, we have $\delta(q, s) \in \mathcal{I}^q$, where s is a global state of the system. If $|\mathcal{I}^q| = 1$, then we have $\mathcal{I}^q = \{\delta(q, s)\}$. This case happens, when the set of all possible states of at least one monitor consists of only one state. In this case, the intersection represents the verdict of a centralized monitor that has global view of the system. This is formalized in the following lemma.

Lemma 4.1. *Let s be a global state of the system and q be the current monitor state. If there is at least one monitor M_i such that $\forall p \in AP.\ \mathcal{S}_i^s(ap) \neq \natural$, then we have $|\mathcal{I}^q| = 1$.*

Proof. It is easy to verify that if there is a monitor M_i such that $\forall p \in AP. \mathcal{S}_i^s(ap) \neq \natural$, then according to our definition, we have $E(\mathcal{S}_i^s) = \{s\}$. Consequently, we have $V_i = \{\delta(q, s)\}$ and it follows that $|\mathcal{I}^q| = 1$. $\square$

Abstraction Function in Automata-Based Algorithm. Here we define the abstract local state LS_i of a monitor M_i to be the verdict set V_i emitted by the monitor. Given the concrete local state $\mathcal{S}_i^s$ of a monitor M_i and the $\textsc{Ltl}_3$ monitor $\mathcal{M}^\varphi$ of an $\textsc{Ltl}$ formula φ, the abstraction function first computes the set of possible global states $E(\mathcal{S}_i^s)$ from viewpoint of monitor M_i, and then calculates the verdict set based on $E(\mathcal{S}_i^s)$. More formally

$$\begin{aligned}
&LS_i = \mu_2(E(\mathcal{S}_i^s), \mathcal{M}^\varphi) = \{\delta(q, s')\}_{s' \in E(\mathcal{S}_i^s)} = V_i \\
&E(\mathcal{S}_i^s) = \mu_1(\mathcal{S}_i^s) = \{s' \in \Sigma \mid \forall ap \in AP : (\mathcal{S}_i^s(ap) \neq \natural) \rightarrow \\
&\qquad\qquad ((\mathcal{S}_i^s(ap) = \mathsf{true} \rightarrow ap \in s') \wedge (\mathcal{S}_i^s(ap) = \mathsf{false} \rightarrow ap \notin s'))\}
\end{aligned}$$

where μ_1 and μ_2 are the abstraction functions. μ_1 receives as input a concrete local state $\mathcal{S}_i^s$ and computes the set of all possible global states $E(\mathcal{S}_i^s)$, and μ_2 receives as input a set of global states and a monitor $\mathcal{M}_\varphi$, and returns the set of all monitor states in $\mathcal{M}_\varphi$ that can be reached by the given global states.

Local Computation Function in Automata-Based Algorithm. The local computation function LC of a monitor M_i calculates the intersection of the messages (which are the verdict sets emitted by nonfaulty monitors) received in Π_i^r, at each round r. Formally,

$$LS_i^{r+1} = LC(\Pi_i^r) = \bigcap_{j \in [1,n]} \{LS_j^r\} = \bigcap_{j \in [1,n]} \{V_j^r\}$$

4.5.2 Detailed Description of The Algorithm

Each local monitor $M_i \in \mathcal{M}$, $i \in [1, n]$, runs Algorithm 2 that we shall describe in detail. For any given new state s_j, monitor M_i first obtains an initial concrete local state by taking a sample from s_j (cf. Line 1). Recall from Definition 4.1 that the value of an atomic proposition in a concrete local state is either true, false, or $\natural$. After obtaining the initial concrete local state, monitor M_i computes the initial abstract local state based on the initial concrete local state, by applying the abstraction functions μ_1 and μ_2 (cf. Line 2). After initialization, each monitor M_i executes a sequence of send, receive, and computation actions (cf. Lines 4-6) for $f+1$ number of rounds. In Line 4, monitor M_i sends its current local state to all other monitors in $\mathcal{M}$. In Line 5, it receives messages from other monitors and stores them, along with its own message, in a set Π_i^r. In line 6, which is the computation step, monitor M_i computes and updates its abstract local state based on the messages in Π_i^r, by applying the local computation function LC which simply calculates the intersection of the verdict sets in Π_i^r. Finally, after $f+1$ rounds, the for-loop ends, and M_i emits $\lambda(v_i) \in \mathbb{B}_3$, where $\{v_i\} = LS_i^{f+2}$(cf. Line 7).

```
Data: LTL$_3$ monitor $\mathcal{M}^\varphi$ and state $s_j$
Result: a verdict from $\mathbb{B}_3$
1 Let $\mathcal{S}_i^{s_j}$ be the initial concrete local state of the monitor
2 $LS_i^1 \leftarrow \mu_2(\mu_1(\mathcal{S}_i^{s_j}, \mathcal{M}_\varphi)) = V_i^1$   /* computes the initial abstract local state based on
    the initial concrete local state. $V_i^r$ denotes the verdict set emitted by monitor $M_i$ to
    be broadcasted at round $r$ */
3 for $r = 1, \cdots, f+1$ do
                                      /* $f$ is the maximum number of crash failures tolerated */
4     Send: broadcasts its current abstract local state $LS_i^r = V_i^r$     /* $r$ is the round
      number */
5     Receive: let $\Pi_i^r = \{V_j^r\}_{j\in[1,n]}$ be the set of all messages received at round $r$.
6     Computation: $LS_i^{r+1} \leftarrow LC_i(\Pi_i^r) = \bigcap_{j\in[1,n]}\{V_j^r\}$   /* calculates a new abstract
      local state */
7 emit $\lambda_e(v_i)$                                  /* where $\{v_i\} = LS_i^{f+2}$ */
```

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

Now let us look at the following example to see how each local monitor implements Algorithm 2 to emit a verdict. In the following example each monitor employs LTL$_3$ monitor $\mathcal{M}^\varphi$ of a given LTL formula φ, in order to compute an abstract local state based on its concrete local state.

Example: Let $\varphi = \mathbf{F}(a \wedge b)$

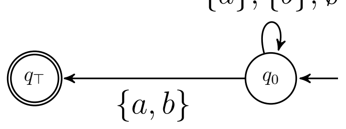


Figure 4.1: LTL$_3$ monitor of $\varphi = \mathbf{F}(a \wedge b)$.

Consider $\mathcal{M} = \{M_1, M_2, M_3, M_4\}$, $s = \{a, b\}$, $S_1^s(a) = \mathsf{true}$, $S_1^s(b) = \natural$, $S_2^s(a) = \natural$, $S_2^s(b) = \mathsf{true}$, $S_3^s(a) = \natural$, $S_3^s(b) = \natural$, $S_4^s(a) = \natural$, $S_4^s(b) = \natural$, and let $f = 2$. According to Algorithm 2, each local monitor M_i computes an abstract local state LS_i^1 based on its concrete local state using abstraction functions μ_1 and μ_2 (cf. Line 1). The initial abstract local states are given in Table below (sample).

sample

	a	b	LS_i^1
M_1	true	$\natural$	$\{q_0, q_\top\}$
M_2	$\natural$	true	$\{q_0, q_\top\}$
M_3	$\natural$	$\natural$	$\{q_0, q_\top\}$
M_4	$\natural$	$\natural$	$\{q_0, q_\top\}$

round 1

	LS_i^2
M_1	crashed
M_2	$\{q_0, q_\top\}$
M_3	$\{q_0, q_\top\}$
M_4	$\{q_0, q_\top\}$

round 2

	LS_i^3
M_1	crashed
M_2	crashed
M_3	$\{q_0, q_\top\}$
M_4	$\{q_0, q_\top\}$

round 3

	LS_i^4
M_1	crashed
M_2	crashed
M_3	$\{q_0, q_\top\}$
M_4	$\{q_0, q_\top\}$

M_1 knows that the value of proposition a is true in state s, but it does not know the value of proposition b in s. Therefore, from its viewpoint state s can be either $\{a\}$ or $\{a, b\}$. We say $\{a\}$ and $\{a, b\}$ are possible global states from veiwpoint of M_1. Thus M_1's initial abstract local state is the verdict set $LS_1^1 = \{q_0, q_\top\}$ which includes the monitor states that can be reached by states $\{a\}$ and $\{a, b\}$, as $\delta(q_0, \{a\}) = q_0$ and $\delta(q_0, \{a, b\}) = q_\top$. Similarly, the possible global states from viewpoint of monitor M_2 are $\{b\}$ and $\{a, b\}$, therfore its initial abstract local state is the verdict set $LS_2^1 = \{q_0, q_\top\}$, which are the monitor states that can be reached by states $\{b\}$ and $\{a, b\}$. M_3 and M_4 do not know the value of any proposition in s, thus from their viewpoint, all global states $\emptyset$, $\{a\}$, $\{b\}$, and $\{a, b\}$ are possible to be the global state s, hence $LS_3^1 = LS_4^1 = \{q_0, q_\top\}$.

Suppose monitor M_1 crashes at round 1 and since it is the only monitor which knows the value of proposition a, we assume its message is received by at least one nonfaulty monitor, e.g. M_2. Therefore, after one round of communication, each monitor updates its abstract local state by calculating the intersection of its own verdict set with the verdict sets received from other monitors (cf. Line 6):

$LS_2^2 = \{q_0, q_\top\}$, $LS_3^2 = \{q_0, q_\top\}$, $LS_4^2 = \{q_0, q_\top\}$

In round 2, monitor M_2 crashes, and again, since it is the only monitor whose abstract local state encapsualtes proposition a, its message must be received by at least one monitor, e.g. M_3, at this round. The abstract local states at the end of round 2 will be updated as follows:

$LS_3^3 = \{q_0, q_\top\}$, $LS_4^3 = \{q_0, q_\top\}$

Finally at round 3, no monitor crashes and M_4 and M_3 receive messages from each other and update their abstract local states:

$LS_3^4 = \{q_0, q_\top\}$, $LS_4^4 = \{q_0, q_\top\}$

As we observe, at the end of round 3 (namely, $f + 1$), the local monitors still cannot decide a single verdict since $|LS_i^4| > 1$. This is because the LTL$_3$ monitor of $\varphi = \mathbf{F}(a \wedge b)$ is not sufficient to distinguish the correct verdict when local monitors have partial view of the system. In particular, monitors M_3 and M_4 both have $\{q_0, q_\top\}$ as their verdicts, while $[\{a, b\} \models_3 \mathbf{F}(a \wedge b)] = \top$. That is, the monitors cannot map their collective verdicts to the verdict of a monitor that has the global view of the system.

In order to resolve this insufficiency, we introduce an algorithm that constructs an 'Extended LTL$_3$ monitor'. The algorithm receives as input an LTL$_3$ monitor and solely based on the structure of the input monitor, it determines whether to add new monitor states to the original LTL$_3$ monitor. The Extended LTL$_3$ monitor then is used in each local monitor M_i's algorithm (Algorithm 2) to consistently solve the decentralized synchronous monitoring problem. As described earlier, the intuition behind this algorithm is to monitor the system under inspection by taking the intersection of the sets of verdicts emitted by a set of distributed monitors.

4.5.3 Synchronous Automata-Based Monitoring Using Extended LTL$_3$ Monitor

In this Section, first we present an algorithm to construct an Extended LTL$_3$ monitor $\mathcal{M}_e^\varphi$ which can be used in Algorithm 2 to solve the synchronous monitoring problem that was described in Section 4.3, for any given LTL formula φ. Then we provide an example to show how $\mathcal{M}_e^\varphi$ is used in each local monitor's algorithm to emit their verdicts and consistently monitor the system.

4.5.3.1 Extended LTL$_3$ Monitor Construction

Let $\mathcal{M}^\varphi = \{\Sigma, Q, \delta, q_0, F\}$ be the LTL$_3$ monitor for LTL formula φ. Our goal is to construct an Extended LTL$_3$ monitor $\mathcal{M}_e^\varphi = \{\Sigma, Q_e, q_0, \delta_e, \lambda_e\}$ such that $|\mathcal{I}^q| = 1$ at every monitor state $q \in Q_e$, where $\mathcal{I}^q$ is the intersection of the verdict sets emitted by a set of distributed monitors whose partial views (namely, concrete local states) cover the global state of the system under inspection (see Definition 4.4).

Definition 4.6. *Let $\mathcal{M}^\varphi = \{\Sigma, Q, q_0, \delta, \lambda\}$ be the* LTL$_3$ *monitor of an* LTL *formula φ. An* Extended LTL$_3$ monitor *of φ is a deterministic finite state machine $\mathcal{M}_e^\varphi = \{\Sigma, Q_e, q_0, \delta_e, \lambda_e\}$, where Q_e is a set of states s.t. $Q \subseteq Q_e$, q_0 is the initial state, δ_e :*

$Q_e \times \Sigma \to 2^{Q_e}$ is a transition function, and $\lambda_e : Q_e \to \mathbb{B}_3$ is a mapping function, such that (1) for every non-empty finite trace $\alpha \in \Sigma^*$, we have $\lambda_e(\delta_e(q_0, \alpha)) = \lambda(\delta(q_0, \alpha))$, and (2) at every $q \in Q_e$ we have $|\mathcal{I}^q| = 1$.

Algorithm 3 constructs an Extended LTL$_3$ monitor given an LTL$_3$ monitor $\mathcal{M}^\varphi$.

```
   Input: $\mathcal{M}^\varphi = \{\Sigma, Q, q_0, \delta, \lambda\}$
   Output: $\mathcal{M}_e^\varphi = \{\Sigma, Q_e, q_0, \delta_e, \lambda_e\}$
1  $Q_e \leftarrow Q$
2  for every $q_i \in Q$ do
3      Obtain the set of outgoing transitions $T_i$ from monitor state $q_i$
4      for every $t_j^i \in T_i$ do
                                                      /* $t_j^i = \{s \in \Sigma \mid \delta(q_i, s) = q_j\}$ */
           /* $N_j$ denotes the number of transitions from which $t_j^i$ is indistinguishable, and $K_j$ denotes the number of transitions
            indistinguishable from $t_j^i$ */
5          $N_j \leftarrow 0$ , $K_j \leftarrow 0$
6          for every $t_k^i \in T_i \backslash \{t_j^i\}$ do
7              if indisting?($t_j^i$, $t_k^i$) then
8                  $N_j \leftarrow N_j + 1$
9              if indisting?($t_k^i$, $t_j^i$) then
10                 $K_j \leftarrow K_j + 1$
11         if $N_j > 0$ then
12             $\{t_{j1}^i, t_{j2}^i\} \leftarrow SPLIT(t_j^i, N_j, K_j, T_i)$
13             $T_i \leftarrow \{t_{j1}^i, t_{j2}^i\} \cup T_i \backslash \{t_j^i\}$
14             $Q_e \leftarrow \{q_{j1}, q_{j2}\} \cup (Q_e \backslash \{q_j\})$
15             if $i \neq j$ then
16                 for every $t_k^i \in T_i$ do
17                     $\delta(q_i, s) = q_k$ for every $s \in t_k^i$
18                 $\delta(q_{j1}, s) \leftarrow \delta(q_i, s)$ for every $s \in \Sigma$
19                 $\delta(q_{j2}, s) \leftarrow \delta(q_i, s)$ for every $s \in \Sigma$
20             if $i = j$ then
21                 for every $t_k^i \in T_i$ do
22                     $\delta(q_{j1}, s) = q_k$ for every $s \in t_k^i$
23                 $\delta(q_{j2}, s) \leftarrow \delta(q_{j1}, s)$ for every $s \in \Sigma$
24             $\lambda_e(q_{j1}) \leftarrow \lambda(q_j)$
25             $\lambda_e(q_{j2}) \leftarrow \lambda(q_j)$
26         else
27             $\delta_e(q_i, s) \leftarrow q_j$ for every $s \in t_j^i$
28             $\lambda_e(q_j) \leftarrow \lambda(q_j)$
```

Algorithm 3: Extended LTL$_3$ monitor Construction

```
function SPLIT(transition $t^i_j$, $N_j$, $K_j$, set of transitions $T_i$)
    $N_{j,min} \leftarrow N_j$
    $K_{j,min} \leftarrow K_j$
    for every $\{t^{i,l}_{j1}, t^{i,l}_{j2}\} \in PARTITION(t^i_j)$ do
        $N_{jl} \leftarrow 0$ , $K_{jl} \leftarrow 0$
        for every $t^i_k \in (\{t^{i,l}_{j2}\} \cup T_i \backslash \{t^i_j\})$ do
            if indisting?($t^{i,l}_{j1}$, $t^i_k$) then
                $N_{jl} \leftarrow N_{jl} + 1$
            if indisting?($t^i_k$, $t^{i,l}_{j1}$) then
                $K_{jl} \leftarrow K_{jl} + 1$
        for every $t^i_k \in (\{t^{i,l}_{j1}\} \cup T_i \backslash \{t^i_j\})$ do
            if indisting?($t^{i,l}_{j2}$, $t^i_k$) then
                $N_{jl} \leftarrow N_{jl} + 1$
            if indisting?($t^i_k$, $t^{i,l}_{j2}$) then
                $K_{jl} \leftarrow K_{jl} + 1$
        if $(N_{jl} + K_{jl}) \leqslant (N_{j,min} + K_{j,min})$ then
            $l_{min} = l$
    $t^i_{j1} = t^{i,l_{min}}_{j1}$ , $t^i_{j2} = t^{i,l_{min}}_{j2}$
    return $\{t^i_{j1}, t^i_{j2}\}$
end function
function PARTITION(transition $t^i_j$)
    compute all partitions $\{t^{i,l}_{j1}, t^{i,l}_{j2}\}$ of $t^i_j$ where $l \in [1 \cdots \frac{2^{|t^i_j|}-2}{2}]$, s.t.
    for every pair $\{t^{i,l}_{j1}, t^{i,l}_{j2}\}$:
      • $t^{i,l}_{j1} \cup t^{i,l}_{j2} = t^i_j$
      • $t^{i,l}_{j1} \cap t^{i,l}_{j2} = \emptyset$
    return $\{\{t^{i,1}_{j1}, t^{i,1}_{j2}\}, \{t^{i2}_{j1}, t^{i2}_{j2}\}, \cdots, \{t^{i,2^{\frac{|t^i_j|-2}{2}}}_{j1}, t^{i,\frac{2^{|t^i_j|}-2}{2}}_{j2}\}\}$
end function
```

Algorithm 4: Functions SPLIT and PARTITION

4.5.3.2 Detailed Description of Algorithm 3

We now explain how Algorithm 3 constructs an Extended Ltl$_3$ monitor $\mathcal{M}_\varphi$ from an Ltl$_3$ monitor $\mathcal{M}^\varphi$. As described in Definition 4.6, the goal is to construct a deterministic finite state machine $\mathcal{M}^\varphi_e = \{\Sigma, Q_e, q_0, \delta_e, \lambda_e\}$ such that $|\mathcal{I}^q| = 1$ at every monitor state $q \in Q_e$. Algorithm 3 first initializes Q_e with Q (cf. Line 1). Then, for every $q_i \in Q$, it obtains the set of all outgoing transitions from q_i, which is denoted by T_i (cf. Line 3). Recall that $t^i_j = \{s \in \Sigma \mid \delta(q_i, s) = q_j\}$ is a transition from monitor state q_i to monitor state q_j. Two variables N_j and K_j are associated to every transition $t^i_j \in T_i$ (cf. Line 5). N_j keeps the number of transitions in T_i from which transition t^i_j is indistinguishable, and K_j keeps the number of transitions in T_i that are *indistinguishable* from transition t^i_j. The notion of 'indistinguishable' is defined in Definition 4.8 below. In Lines 7-10, the algorithm verifies for every transition $t^i_k \in T_i \backslash \{t^i_j\}$ whether t^i_j is indistinguishable from t^i_k, and/or t^i_k is indistinguishable form t^i_j, and updates N_j and K_j, respectively. If $N_j = 0$ it means that t^i_j is *distinguishable* from all transitions in $T_i \backslash \{t^i_j\}$, thus there is no need to split t^i_j, therefore the algorithm proceeds to Lines 27-28. In Lines 27 and 28 transition function δ_e and mapping function λ_e of the Extended LTL$_3$ monitor are updated by adding transitions $\delta_e(q_i, s) = q_j, \forall s \in t^i_j$, and the mapping $\lambda_e(q_j) = \lambda(q_j)$, respectively.

If $N_j > 0$ it means that there is at least one transition $t^i_k \in T_i \backslash \{t^i_j\}$ such that t^i_j is indistinguishable from t^i_k. In this case, the Algorithm proceeds to Lines 12-25. in Line 12 transition t^i_j is splitted into two new transitions t^i_{j1} and t^i_{j2} by using the function SPLIT. Function SPLIT is described below. In Line 13, the new transitions t^i_{j1} and t^i_{j2} are added to the set of outgoing transitions T_i from monitor state q_i and t^i_j is removed from T_i. In Line 14, monitor state q_j is replaced by two new monitor states q_{j1} and q_{j2} in Q_e. The transition function δ_e is updated in Lines 15-23. If the transition t^i_j is not a self-loop, i.e., $i \neq j$, then the transition function δ_e is updated through Lines 16-19. If t^i_j is a self-loop, then δ_e is updated through Lines 21-23, where the monitor state q_i is practically replaced by q_{j1}. Finally, in Lines 24-25, the mapping function λ_e is updated with the new mappings $\lambda_e(q_{j1}) = \lambda(q_j)$ and $\lambda_e(q_{j2}) = \lambda(q_j)$.

As mentioned earlier, when a transition t^i_j is splitted into two transitions t^0_{j1} and t^0_{j2}, consequently two new monitor states q_{j1} and q_{j2} are added to $\mathcal{M}^{\varphi}_e$. Note that

q_{j1} and q_{j2} have the same mapping and the same set of outgoing transitions as q_i. Namely, $\lambda(q_{j1}) = \lambda(q_{j2}) = \lambda(q_i)$, and $T_{j1} = T_{j2} = T_i$. This is in fact a necessary condition in order to have

$$[\alpha \models_3 \varphi] = [\alpha \models_3^e \varphi]$$

where $[\alpha \models_3^e \varphi] \in \mathbb{B}_3$ denotes the valuation of any finite trace α according to an Extended LTL$_3$ monitor.

Definition 4.7. *We say state s is 'covered' by transition t, and we denote it by covered?(s, t), if we have:*

$$\forall ap \in AP.\ \exists s' \in t.\ (ap \in s \Leftrightarrow ap \in s')$$

Definition 4.8. *We say a transition t_1 is 'indistinguishable' from another transition t_2, and denote it by indisting?(t_1, t_2), if the following holds:*

$$\exists s \in t_2.\ covered?(s, t_1)$$

Transition t_1 is distinguishable from t_2, denoted by disting?(t_1, t_2), if it is not indistinguishable form t_2.

Function SPLIT This function is in fact the main function in Algorithm 3. Given a transition $t^i_j \in T_i$, it splits t^i_j into two new transitions t^i_{j1} and t^i_{j2} such that the total number of transitions in $T_i \backslash \{t^i_j\}$ that are indistinguishable from t^i_{j1} and transitions that are indistinguishable from t^i_{j2}, plus the total number of transitions in $T_i \backslash \{t^i_j\}$ from which t^i_{j1} is indistinguishable and transitions from which t^i_{j2} is indistinguishable is minimum. This is because we are interested in generating the minimum number of new transitions, and consequently minimum number of new monitor states. Hence, we can claim that an Extended LTL$_3$ monitor constructed by Algorithm 3 is optimum in the terms that $|Q_e|$ is minimum. This is done as follows; first all *partition*s of t^i_j are calculated by function PARTITION which is described below. Two variables N_{jl} and K_{jl} are used in function SPLIT where N_{jl} is a counter for the total number of transitions in $T_i \backslash \{t^i_j\}$ from which t^i_{j1} is indistinguishable, and transitions from which t^i_{j2} is indistinguishable. K_{jl} is a counter for the total number of transitions in $T_i \backslash \{t^i_j\}$ that are indistinguishable from t^i_{j1}, and transitions that are indistinguishable from t^i_{j2}. Function SPLIT calculates $N_{jl} + K_{jl}$ for all partitions $\{t^{i,l}_{j1}, t^{i,l}_{j2}\} \in PARTITION$ and returns a partition with minimum value of $N_{jl} + K_{jl}$.

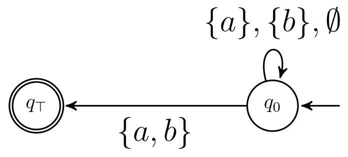
Function PARTITION Given a transition t^i_j, this function returns the set of all possible partitions $\{t^i_{j1}, t^i_{j2}\}$ such that:

- $t^i_{j1} \cap t^i_{j2} = \emptyset$
- $t^i_{j1} \cup t^i_{j2} = t^i_j$

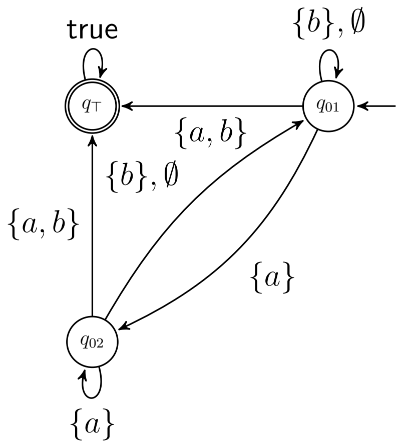
It is easy to verify that the total number of such partitions is equal to $\frac{2^{|t^i_j|}-2}{2}$.

Now we employ our Extended LTL$_3$ monitor in each local monitor's algorithm to consistently monitor the global state of the system. We replace the LTL$_3$ monitor $\mathcal{M}^\varphi$ in Algorithm 2 with an Extended LTL$_3$ monitor.

Example Let us construct an Extended LTL$_3$ monitor for $\varphi = \mathbf{F}(a \wedge b)$ whose LTL$_3$ monitor is given in Fig. 4.2(a).



(a) $\mathcal{M}^\varphi$



(b) $\mathcal{M}_e^\varphi$

Figure 4.2: LTL$_3$ monitor vs. Extended LTL$_3$ monitor for $\varphi = \mathbf{F}(a \wedge b)$

We have $\mathcal{M}^\varphi = \{\{a, b\}, \{q_0, q_\top\}, q_0, \delta, \lambda\}$, where $\delta(q_0, \{a\}) = \delta(q_0, \{b\}) = \delta(q_0, \emptyset) = q_0$ and $\delta(q_0, \{a, b\}) = q_\top$. The set of outgoing transitions from monitor state q_0 is $T_0 = \{t_0^0, t_\top^0\}$, where $t_0^0 = \{\{a\}, \{b\}, \emptyset\}$ and $t_\top^0 = \{\{a, b\}\}$ are the outgoing transitions from monitor state q_0 to monitor states q_0 and $q_\top$, respectively. We can verify that transition t_0^0 is indistinguishable from $t_\top^0$ since there is a state $\{a, b\} \in t_\top^0$ that is covered by transition t_0^0, i.e., *covered?*$(\{a, b\}, t_0^0) =$ true. But $t_\top^0$ is not indistinguishable from t_0^0. Therefore, we have $N_0 = 1$, $K_0 = 0$, $N_\top = 0$, and $K_\top = 0$. Since $N_0 > 0$, we split t_0^0 into two transitions t_{01}^0 and t_{02}^0. Different partitions of t_0^0 are as follows:

$$\begin{aligned} t_{01}^{0,1} &= \{\{a\}\} & t_{02}^{0,1} &= \{\{b\}, \emptyset\} \\ t_{01}^{0,2} &= \{\{b\}\} & t_{02}^{0,2} &= \{\{a\}, \emptyset\} \\ t_{01}^{0,3} &= \{\emptyset\} & t_{02}^{0,3} &= \{\{b\}, \{a\}\} \end{aligned}$$

Note that there are $\frac{2^{|t_0^0|}-2}{2} = 3$ different partitions. For each partition $t_{01}^{0,l}$ we calculate N_{0l} and K_{0l} as follows:

$$\begin{aligned} N_{01} &= 0 & K_{01} &= 0 \\ N_{02} &= 0 & K_{02} &= 0 \\ N_{03} &= 2 & K_{03} &= 1 \end{aligned}$$

We can verify that $indisting?(t_{02}^{0,3}, t_{\top}^{0}) = \mathsf{true}$ and $indisting?(t_{02}^{0,3}, t_{01}^{0,3}) = \mathsf{true}$, therefore $N_{03} = 2$. Also $indisting?(t_{02}^{0,3}, t_{01}^{0,3}) = \mathsf{true}$ results in $K_{03} = 1$. As we can see partitions $\{t_{01}^{0,1}, t_{02}^{0,1}\}$ and $\{t_{01}^{0,2}, t_{02}^{0,2}\}$ are both optimum partitions as they result in minimal value for $N_{0l} + K_{0l}$. Thus, we split t_0^0 into two transitions $t_{01}^0 = \{\{a\}\}$ and $t_{02}^0 = \{\{b\}, \emptyset\}$, and consequently add new monitor states q_{01} and q_{01} to $\mathcal{M}_e$ (see Figure 4.2(b)). Since transition t_0^0 is a self-loop, therefore monitor state q_0 is replaced by monitor state q_{01}. The mapping function for the new monitor states is as follows:

$$\lambda(q_{01}) = \lambda(q_0) = ?$$
$$\lambda(q_{02}) = \lambda(q_0) = ?$$

It is easy to verify that there are no more indistinguishable transitions in the monitor, therefore Figure 4.2 represents the final Extended LTL$_3$ monitor for $\varphi = \mathbf{F}(a \wedge b)$.

We now repeat the example from Section 4.5.1 for formula $\varphi = \mathbf{F}(a \wedge b)$ and this time we use Extended LTL$_3$ monitor in our algorithm (Algorithm 2). Let $\mathcal{M} = \{M_1, M_2, M_3, M_4\}$, $s = \{a, b\}$, $S_1^s(a) = \mathsf{true}$, $S_1^s(b) = \natural$, $S_2^s(a) = \natural$, $S_2^s(b) = \mathsf{true}$,

$S_3^s(a) = \natural$, $S_3^s(b) = \natural$, $S_4^s(a) = \natural$, $S_4^s(b) = \natural$, and let $f = 2$. According to Algorithm 2, each local monitor M_i computes an abstract local state LS_i^1 based on its concrete local state using abstraction functions μ_1 and μ_2 (cf. Line 2). Since all steps are as before except that we use Extended LTL$_3$ monitor in Algorithm 2, we skip the details in order to avoid redundancy, and just recalculate the new verdict sets emitted by each local monitor (note that all local monitors are at the initial monitor state q_{01}).

Verdict sets after obtaining initial concrete local states:

$$E(\mathcal{S}_1^s) = \{\{a\}, \{a,b\}\} \;\Rightarrow\; LS_1^1 = V_1^1 = \{q_{02}, q_\top\}$$
$$E(\mathcal{S}_2^s) = \{\{b\}, \{a,b\}\} \;\Rightarrow\; LS_2^1 = V_2^1 = \{q_{01}, q_\top\}$$
$$E(\mathcal{S}_3^s) = \{\{a\}, \{b\}, \{a,b\}\} \;\Rightarrow\; LS_3^1 = V_3^1 = \{q_{01}, q_{02}, q_\top\}$$
$$E(\mathcal{S}_4^s) = \{\{a\}, \{b\}, \{a,b\}\} \;\Rightarrow\; LS_4^1 = V_4^1 = \{q_{01}, q_{02}, q_\top\}$$

At the end of round 1:

$$LS_2^2 = V_2^2 = \{q_\top\}$$
$$LS_3^2 = V_3^2 = \{q_{01}, q_\top\}$$
$$LS_4^2 = V_4^2 = \{q_{01}, q_\top\}$$

At the end of round 2:

$$LS_3^3 = V_3^3 = \{q_\top\}$$
$$LS_4^3 = V_4^3 = \{q_{01}, q_\top\}$$

At the end of round 3:

$$LS_3^4 = V_3^4 = \{q_\top\}$$
$$LS_4^4 = V_4^4 = \{q_\top\}$$

The following tables summarize the scenario:

sample

	a	b	LS_i^1
M_1	true	$\natural$	$\{q_{02}, q_\top\}$
M_2	$\natural$	true	$\{q_{01}, q_\top\}$
M_3	$\natural$	$\natural$	$\{q_{01}, q_{02}, q_\top\}$
M_4	$\natural$	$\natural$	$\{q_{01}, q_{02}, q_\top\}$

round 1

	LS_i^2
M_1	crashed
M_2	$\{q_\top\}$
M_3	$\{q_{01}, q_\top\}$
M_4	$\{q_{01}, q_\top\}$

round 2

	LS_i^3
M_1	crashed
M_2	crashed
M_3	$\{q_\top\}$
M_4	$\{q_{01}, q_\top\}$

round 3

	LS_i^4
M_1	crashed
M_2	crashed
M_3	$\{q_\top\}$
M_4	$\{q_\top\}$

As we observe, at the end of round 3 (namely, $f + 1$), the abstract local states of all nonfaulty monitors include the single monitor state $q_\top$, and therefore they both emit the same truth value $\lambda(q_\top) = \top$.

4.5.3.3 Proof of Correctness of Algorithm 2

In order to prove the soundness of Algorithm 2, we have to prove that $|\mathcal{I}^q| = 1$ at every monitor state $q \in Q_e$. As described above, an Extended LTL$_3$ monitor is constructed such that at every monitor state $q \in Q_e$, every two outgoing transitions t_j^q and t_k^q from monitor state q are distinguishable. Therefore, to prove the soundness of Algorithm 2, it suffices to prove the following theorem.

Theorem 4.1. *If at every monitor state $q \in Q_e$, every two outgoing transitions t_j^q and t_k^q from monitor state q are distinguishable, then we have $|\mathcal{I}^q| = 1$. Formally*

$$(\forall t_j^q, t_k^q \in T_q.\ disting?(t_j^q, t_k^q)) \Leftrightarrow |\mathcal{I}^q| = 1$$

where T_q is the set of all outgoing transitions from monitor state q.

We prove theorem 4.1 in two steps. First, we prove that

$$(\forall t_j^q, t_k^q \in T_q.\ disting?(t_j^q, t_k^q)) \Rightarrow |\mathcal{I}^q| = 1$$

Proof is by contradiction. Suppose every transition in T_i is indistinguishable from all other transitions in T_i, and suppose $|\mathcal{I}^q| > 1$. Let s be the global state of the system and suppose $\delta(q, s) = q_k$, therefore we know $q_k \in \mathcal{I}^q$. Since $|\mathcal{I}^q| > 1$, thus there exists another monitor state $q_j \in \mathcal{I}^q$. Since $q_j \in \mathcal{I}^q$ therefore for every local monitor M_i there exists an state $s' \in t_j^q$ that is possible to be the global state. We also assumed that the set of monitors satisfy the state coverage, thus for every $ap \in AP$, there exists monitor M_i such that $\mathcal{S}_i^s(ap) \neq \natural$. Formally:

1. $\forall ap \in AP.\ \exists M_i \in \mathcal{M}.\ (\mathcal{S}_i^{s'}(ap) = \mathsf{true} \rightarrow ap \in s) \wedge (\mathcal{S}_i^{s'}(ap) = \mathsf{false} \rightarrow ap \notin s))$
2. $\forall M_i \in \mathcal{M}.\ \exists s' \in t_j^q.\ s' \in E_i$

Therefore, for every $ap \in AP$ there exists $s' \in t_j^q$ such that $(ap \in s' \Leftrightarrow ap \in s)$, which means that s is covered by t_j^q, and consequently t_j^q is indistinguishable from t_k^q, which is a contadiction, hence the proof is complete.

Now we have to prove that

$$|\mathcal{I}^q| = 1 \Rightarrow (\forall t_j^q, t_k^q \in T_q.\ disting?(t_j^q, t_k^q))$$

The proof, again, is by contradiction. Suppose $|\mathcal{I}^q| = 1$ and suppose there exist transitions t_j^q and t_k^q such that $indisting?(t_j^q, t_k^q) = \mathsf{true}$. According to Definition 4.8, transition t_j^q is indistinguishable from t_k^q if there exists an state $s' \in t_k^q$ such that s' is covered by t_j^q, i.e.,

$$\forall ap \in AP.\ \exists s \in t_j^q.\ (ap \in s' \Leftrightarrow ap \in s)$$

Now consider the case where the global state of the system under inspection is s' and let us verify how the local monitors emit their verdicts. Recall from Section 4.5.1, that $\mathcal{I}^q$ is the intersection of all verdict sets emitted by the local monitors which have partial view (concrete local state) of the global state of the system, such that their concrete local states satisfy the state coverage (see Definition 4.4). It is easy to verify that the worst case scenario, upon which an Extended LTL$_3$ monitor is constructed, is when each verdict set is emitted by a monitor which knows the value of at most one atomic proposition $ap \in AP$. Consider the global state s', since $s' \in t_k^q$ thus we have $\delta(q, s') = q_k$, where q_k is the monitor state for which t_k^q is an incoming transition. Therefore a local monitor M_i which has full view of the state s', i.e., for every $ap \in AP$, $\mathcal{S}_i^{s'}(ap) \neq \natural$, emits the verdict $\{q_k\}$. However, in the worst case scenario, each local monitor only reads the value of one atomic proposition. In this case, we can verify that

$$\forall i \in [1, n].\ q_j \in V_i^1$$

where V_i^1 is the verdict set emitted by monitor M_i at round 1 (recall from Algorithm 2 that V_i^1 is calculated based on M_i's concrete local state), and q_j is the monitor state for which t_j^q is an incoming transition. This holds because we have

$$\forall ap \in AP.\ \exists s \in t_j^q.\ (ap \in s' \Leftrightarrow ap \in s)$$

To see this more clearly, we need to recall the definition of the set of possible states E_i from viewpoint of a local monitor M_i,

$$E(\mathcal{S}_i^{s'}) = \{s \in \Sigma \mid \forall ap \in AP : (\mathcal{S}_i^{s'}(ap) \neq \natural) \rightarrow ((\mathcal{S}_i^{s'}(ap) = \mathsf{true} \rightarrow ap \in s) \wedge (\mathcal{S}_i^{s'}(ap) = \mathsf{false} \rightarrow ap \notin s))\}$$

Now all we need to do is to show that

$$\forall i \in [1, n].\ \exists s \in t_j^q.\ s \in E_i$$

and consequently, $q_j \in V_i^1$ for every $i \in [1, n]$.

In order to prove the above statement we claim that

$$\nexists ap \in AP.\ (\mathcal{S}_i^{s'}(ap) \neq \natural) \wedge (\nexists s \in t_j^q.\ (ap \in s' \Leftrightarrow ap \in s))$$

Informally, for every local monitor M_i, there is an atomic proposition $ap \in AP$ such that $\mathcal{S}_i^{s'} = \natural$ (since we assumed no local monitor has the full view of the system), and there exists state $s \in t_j^q$ such that $(ap \in s' \Leftrightarrow ap \in s)$ (since s' is covered by t_j^q). Therefore according to definition of E_i, we have $s \in E_i$, and hence $q_j \in V_i^1$. Thus we proved that

$$indisting?(t_j^q, t_k^q) \Rightarrow \exists s \in t_k^q.\ \forall i \in [1,n].\ \{(\exists ap \in AP.\ \mathcal{S}_i^s(ap) = \natural) \rightarrow q_j \in V_i^1\}$$

Therefore in the worst case scenario where no local monitor has the full view of the system, q_j will appear in the verdict set emitted by each monitor, and therefore $|\mathcal{I}^q| > 1$, which is a contradiction, and the proof is complete.

Algorithm 3 constructs an Extended $\text{L}{\small\text{TL}}_3$ monitor $\mathcal{M}_e^\varphi$ such that at every monitor state $q \in Q_e$, every outgoing transition is distinguishable from all other outgoing transitions, and therefore $|\mathcal{I}^q| = 1$ at every $q \in Q_e$.

Chapter 5

Decentralized Asynchronous Monitoring

This Chapter discusses the decentralized asynchronous monitoring problem in the presence of faulty monitors. The problem statement and the challenges in asynchronous monitoring are discussed in Sections 5.1 and 5.3, respectively. Model of computation and terminology are presented in Section 5.2. In Section 5.4 we propose an automata-based algorithm which employs LTL_3 monitor to compute the verdict sets emitted by local monitors. Finally in Section 5.5 we present an Algorithm that uses an Extended LTL_3 monitor to solve the decentralized asynchronous monitoring problem where local monitors emit their verdicts without any round of communication (i.e., accessing the shared memory).

5.1 Problem Statement

The system under inspection produces a finite trace $\alpha = s_0 s_1 \cdots s_k$, and is inspected with respect to an LTL formula φ by a set $\mathcal{M} = \{M_1, M_2, \cdots, M_n\}$ of asynchronous distributed wait-free monitors. The notion of wait-free distributed monitoring is formally introduced in Bonakdarpour *et al.* (2016) as follows.

Definition 5.1. *By wait-free monitoring we mean that each monitor (1) runs at its own speed, that may vary along with time and (2) may fail by crashing (i.e., halt and never recover), thus a monitor never waits for another monitor (in order to avoid a livelock).*

For every $j \in [0, k-1]$, between each s_j and $s_j + 1$, each monitor $M_i \in \mathcal{M}$, $i \in [1, n]$, in a wait-free manner:

1. reads the value of propositions in s_j, which may result in a partial observation of s_j;
2. repeatedly communicates its partial observation with other monitors through a single-writer/multi-reader shared memory;
3. updates its knowledge resulting from the aforementioned communication, and
4. evaluates φ and emits a verdict based on its current knowledge.

```
Data: LTL formula $\varphi$ and state $s_j$
Result: a verdict from $\mathbb{B}_3$
1 Let $\mathcal{S}_i^{s_j}$ be the initial concrete local state of the monitor ;
2 $Snap_i^{s_j} \leftarrow \mathcal{S}_i^{s_j}$;                    /* take sample from state $s_j$ */
3 for some fixed number of rounds $r \geqslant 1$ do
4     $SM_i^{s_j} \leftarrow \mathbf{project}(Snap_i^{s_j})$;    /* write current knowledge in shared memory */
5     $Snap_i^{s_j} \leftarrow SM^{s_j}$;                  /* take a snapshot of the shared memory */
6 emit $[\mathbf{x}(Snap_i^{s_j}) \dots \mathbf{x}(Snap_i^{s_j}) \models_4 \varphi]$;   /* evaluate $\varphi$ using extrapolation function */
```

Algorithm 5: Behavior of Monitor M_i, for $i \in [1, n]$

Each monitor M_i in $\mathcal{M}$ is a process, and the monitors run in the standard asynchronous wait-free read/write shared memory model Attiya and Welch (2004). We assume that up to $n-1$ monitors can crash. Every monitor that does not fail is required to emit a verdict. Hence, a distributed algorithm in this settings consists for each monitor in a bounded sequence of read/write accesses to the shared memory at the end of which a verdict is emitted. We thus assume without loss of generality that each monitor may access the shared memory a fixed number of times before emitting a verdict M. Herlihy and Rajsbaum (2013); Bonakdarpour *et al.* (2016).

In our setting the assumption is that the set of monitors satisfy the state coverage. Thus if a proposition is read by only one monitor, then this monitor is supposed to send its information to at least one nonfaulty monitor before crashing.

Algorithm 5 represents the aforementioned asynchronous monitoring scenario. Our problem statement is that when a non-faulty monitor runs Algorithm 5, it should compute and emit a verdict such that it ensures consistent distributed monitoring. Namely, one has to be able to map a collective set of verdicts of monitors (for any execution interleaving) to one and only one verdict of a centralized monitor that has the full view s_j. A necessary condition for this mapping is that, for every two finite traces $\alpha, \alpha' \in \Sigma^*$, if $[\alpha \models_3 \varphi] \neq [\alpha' \models_3 \varphi]$ (or alternatively, $[\alpha \models_4 \varphi] \neq [\alpha' \models_4 \varphi]$), then the monitors in $\mathcal{M}$ should compute different collective sets of verdicts for α and α', regardless of their initial partial observation and different read/write interleavings.

5.2 Model of Computation and Terminology

For every state s_j in $\alpha = s_0 s_1 \cdots s_k$, each monitor M_i maintains a local snapshot $Snap_i$. Each local snapshot has n registers, one per each monitor in $\mathcal{M}$. The local register of monitor M_i associated with monitor M_l for state s_j is denoted by $Snap_i^{s_j}[l]$. Each register has $|AP|$ elements, one for each atomic proposition in AP. The monitors in $\mathcal{M}$ communicate by means of shared memory. The structure of the shared memory SM is as follows: for each state s_j, SM^{s_j} consists of n atomic registers, one per monitor, and each register has $|AP|$ elements one for each atomic proposition (i.e, single-writer/multiple-reader (SWMR) registers). Thus, for state s_j, each monitor M_i can read the entire content of SM^{s_j}, but can only write into register $SM_i^{s_j}$. We assume that each monitor is aware of the change of state of the system under inspection. Thus, for a state s_j, a monitor M_i reads and writes in the associated local and shared memory locations, i.e., $Snap_i^{s_j}$ and SM^{s_j}.

The asynchronous monitoring algorithm. Each monitor $M_i \in \mathcal{M}$, $i \in [1, n]$, runs Algorithm 5. For any given new state s_j, Monitor M_i first initializes all elements of its local snapshot to $\natural$ (cf. Line 1). Then, M_i takes a sample from state

s_j and obtains a concrete local state $\mathcal{S}_i^{s_j}$(cf. Line 2). Recall from Definition 4.1 that the value of an atomic proposition in a concrete local state is either true, false, or $\natural$. We assume that the set of monitors satisfy the state coverage whose definiton was presented in Chapter 4. The set of values in the concrete local state is copied in local snapshot $Snap_i^{s_j}$. Then each monitor M_i executes a sequence of write/read actions (cf. Lines 4 and 5) for some a priori known number of times. in Line 4, it atomically writes the content of its local snapshot $Snap_i^{s_j}$ into its associated register $SM_i^{s_j}$ in the shared memory. In Line 5, M_i reads of all the registers in $SM_i^{s_j}$, and copies them into $Snap_i^{s_j}$, in a single atomic step.

5.3 Challenges in Asynchronous Monitoring

In Bonakdarpour *et al.* (2016) it is shown that RV-LTL is not sufficient to consistenly monitor the global system state in a distributed asynchronous environment. To overcome this insufficiency, they introduced a family of multi-valued logics (called LTL$_{2k+4}$), for every $k \geqslant 0$, where k relates to the notion of alternation number which is introduced and formally defined in Bonakdarpour *et al.* (2016).

We use the example from Bonakdarpour *et al.* (2016) and we show how each local monitor runs Algorithm 5 and emits a verdict from $\mathbb{B}_4$ based on the final content of its local snapshot.

Example Let $\mathcal{M} = \{M_1, M_2\}$ and consider the formula for two requests and acknowledgments:

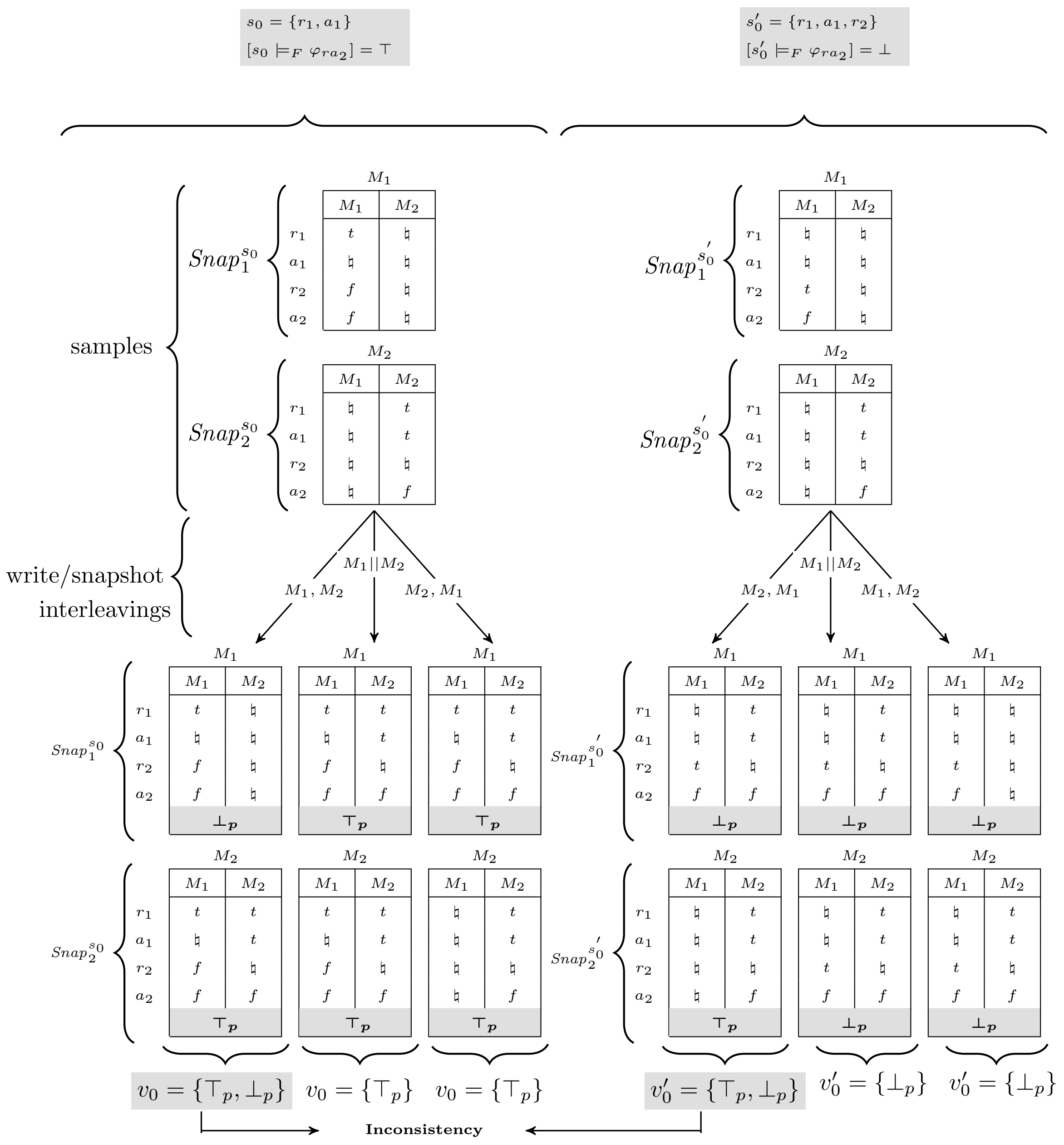
$$\varphi_{ra_2} = \Big(\mathbf{G}(\neg a_1 \wedge \neg r_1) \vee [(\neg a_1\, \mathbf{U}\, r_1) \wedge \mathbf{F} a_1]\Big) \wedge \Big(\mathbf{G}(\neg a_2 \wedge \neg r_2) \vee [(\neg a_2\, \mathbf{U}\, r_2) \wedge \mathbf{F} a_2]\Big)$$

Fig. 5.1 shows different execution interleavings of monitors M_1 and M_2 when running Algorithm 1 from states $s_0 = \{r_1, a_1\}$ and $s'_0 = \{r_1, a_1, r_2\}$. Based on the order of monitor write-snapshot actions: M_1, M_2 (resp., M_2, M_1) denotes the case where monitor M_1 (resp., M_2) executes a write-snapshot before monitor M_2 (resp., M_1) does, and $M_1||M_2$ denotes the case where monitors M_1 and M_2 execute their write-snapshot actions concurrently. In case of s_0, after executing Line 2 of Algorithm 5, monitor M_1's sample, i.e., the local snapshot $\mathit{Snap}_1^{s_0}[1]$, consists of $\mathcal{S}_1^{s_0}(r_1) = \mathsf{true}$, $\mathcal{S}_1^{s_0}(a_1) = \natural$, and $\mathcal{S}_1^{s_0}(r_2) = \mathcal{S}_1^{s_0}(a_2) = \mathsf{false}$. Moreover, initially, M_1 has no knowledge of M_2's sample. Monitor M_2's sample from s_0, i.e., the local snapshot $\mathit{Snap}_2^{s_0}[2]$, consists of $\mathcal{S}_2^{s_0}(r_1) = \mathcal{S}_2^{s_0}(a_1) = \mathsf{true}$, $\mathcal{S}_2^{s_0}(r_2) = \natural$, and $\mathcal{S}_2^{s_0}(a_2) = \mathsf{false}$ while it initially has no knowledge of M_1's sample. Likewise, for state s'_0, Fig. 5.1 shows different local snapshots by M_1 and M_2. Given two values v_1 and v_2, we define (an arbitrary) extrapolation function as follows:

$$\mathbf{x}_{ap}(v_1, v_2) = \begin{cases} \mathsf{true} & \text{if } (v_1 = \mathsf{true}) \vee (v_2 = \mathsf{true}) \\ \mathsf{false} & \text{otherwise} \end{cases}$$

where $ap \in \{a_1, r_1, a_2, r_2\}$. Finally, starting from s_0, if (1) the for loop of Algorithm 5 terminates after 1 communication round, and (2) the interleaving is M_1, M_2, then $\mathbf{x}(\llbracket \mathit{Snap}_2^{s_0} \rrbracket) = \{r_1, a_1\}$, and evaluation of φ_{ra_2} by M_2 in LTL_4 results in $[\mathbf{x}(\llbracket \mathit{Snap}_2^{s_0} \rrbracket) \models_4$

$\varphi_{ra_2}] = \top_p$.



Samples at s_0:

M_1	M_1	M_2
r_1	t	$\natural$
a_1	$\natural$	$\natural$
r_2	f	$\natural$
a_2	f	$\natural$

M_2	M_1	M_2
r_1	$\natural$	t
a_1	$\natural$	t
r_2	$\natural$	$\natural$
a_2	$\natural$	f

Samples at s'_0:

M_1	M_1	M_2
r_1	$\natural$	$\natural$
a_1	$\natural$	$\natural$
r_2	t	$\natural$
a_2	f	$\natural$

M_2	M_1	M_2
r_1	$\natural$	t
a_1	$\natural$	t
r_2	$\natural$	$\natural$
a_2	$\natural$	f

Interleavings at s_0 (M_1, M_2 / $M_1||M_2$ / M_2, M_1), $Snap_1^{s_0}$:

M_1	M_1	M_2	M_1	M_2	M_1	M_2
r_1	t	$\natural$	t	t	t	t
a_1	$\natural$	$\natural$	$\natural$	t	$\natural$	t
r_2	f	$\natural$	f	$\natural$	f	$\natural$
a_2	f	$\natural$	f	f	f	f
	$\bot_p$		$\top_p$		$\top_p$	

$Snap_2^{s_0}$:

M_2	M_1	M_2	M_1	M_2	M_1	M_2
r_1	t	t	t	t	$\natural$	t
a_1	$\natural$	t	$\natural$	t	$\natural$	t
r_2	f	$\natural$	f	$\natural$	$\natural$	$\natural$
a_2	f	f	f	f	$\natural$	f
	$\top_p$		$\top_p$		$\top_p$	

Interleavings at s'_0 (M_2, M_1 / $M_1||M_2$ / M_1, M_2), $Snap_1^{s'_0}$:

M_1	M_1	M_2	M_1	M_2	M_1	M_2
r_1	$\natural$	t	$\natural$	t	$\natural$	$\natural$
a_1	$\natural$	t	$\natural$	t	$\natural$	$\natural$
r_2	t	$\natural$	t	$\natural$	t	$\natural$
a_2	f	f	f	f	f	$\natural$
	$\bot_p$		$\bot_p$		$\bot_p$	

$Snap_2^{s'_0}$:

M_2	M_1	M_2	M_1	M_2	M_1	M_2
r_1	$\natural$	t	$\natural$	t	$\natural$	t
a_1	$\natural$	t	$\natural$	t	$\natural$	t
r_2	$\natural$	$\natural$	t	$\natural$	t	$\natural$
a_2	$\natural$	f	f	f	f	f
	$\top_p$		$\bot_p$		$\bot_p$	

Figure 5.1: Example: Monitors M_1 and M_2 monitoring formula φ_{ra_2} from two different states s_0 and s'_0.

Note that v_0 and v'_0 denote the sets of verdict emitted by monitors at states s_0

and s', respectively.

We observe that LTL$_4$ is not sufficienet to consistently monitor all LTL formulas. We can see that in Fig. 5.1, the shaded collective verdicts v_0 and v'_0 are both equal to $\{\bot_p, \top_p\}$, but $[s_0 \models_4 \varphi] \neq [s'_0 \models_4 \varphi]$. Therefore it violates the condition that, if $[\alpha \models_4 \varphi] \neq [\alpha' \models_4 \varphi]$, then the monitors in $\mathcal{M}$ should compute different collective sets of verdicts for α and α'.

5.4 Asynchronous Automata-Based Monitoring Algorithm Using LTL$_3$ Monitor

In this Section, we present a new approach to solve the asynchronous monitoring problem in a failure-prone distributed environment utilizing the idea of computing the intersection of the verdict sets emitted by the distributed monitors.

We use the abstraction functions and the local computation function that were defined in Chapter 4. We modify Algorithm 5 as follows; After a number of read/write accesses to the shared memory, each local monitor M_i computes a verdict set $V_i^{s_j}$ based on its local snapshot $Snap_i^{s_j}$, where $V_i^{s_j}$ is defined as follows:

$$V_i^{s_j} = \mu_2(E(Snap_i^{s_j})) = \{\delta(q,s)\}_{s \in E(Snap_i^{s_j})}$$
$$E(Snap_i^{s_j}) = \mu_1(Snap_i^{s_j}, \mathcal{M}^{\varphi}) = \{s \in \Sigma \mid \forall ap \in AP : (Snap_i^{s_j}(ap) \neq \natural) \rightarrow$$
$$((Snap_i^{s_j}(ap) = \mathsf{true} \rightarrow ap \in s_j) \wedge (Snap_i^{s_j}(ap) = \mathsf{false} \rightarrow ap \notin s_j))\}$$

where μ_1 and μ_2 are the abstraction functions. In fact each monitor M_i emits a set of all monitor states that can be reach by any of possible global states from viewpoint of M_i, i.e., by any $s \in E(Snap_i^{s_j})$ (cf. Line 6).

```
Data: LTL3 monitor $\mathcal{M}^{\varphi} = \{\Sigma, Q, q_0, \delta, \lambda\}$ and state $s_j$
Result: a set of monitor states
1 Let $\mathcal{S}_i^{s_j}$ be the initial concrete local state of the monitor ;
2 $Snap_i^{s_j} \leftarrow \mathcal{S}_i^{s_j}$;                       /* take sample from state $s_j$ */
3 for some fixed number of rounds $r \geqslant 1$ do
4     $SM_i^{s_j} \leftarrow$ project($Snap_i^{s_j}$);    /* write current knowledge in shared memory */
5     $Snap_i^{s_j} \leftarrow SM^{s_j}$;                  /* take a snapshot of the shared memory */
6 emit $V_i^{s_j} = \mu_2(\mu_1(Snap_i^{s_j}, \mathcal{M}^{\varphi}))$ ; /* emit a set of monitor states according to $\mathcal{M}_{\varphi}$
   */
```
Algorithm 6: Behavior of Monitor M_i, for $i \in [1, n]$

As we see in Line 3 of the Algorithm $r \geqslant 1$, i.e., each monitor has at least one read/write access to the shared memory before emitting a verdict. Thus there is at least one monitor (the one that accesses the shared memory last) that has the full view of state s_j before emitting its verdict. According to Lemma 4.1, If there is at least one monitor M_i such that $\forall p \in AP.\ \mathcal{S}_i^s(ap) \neq \natural$, then we have $|\mathcal{I}^q| = 1$, where q is the current monitor state. Therefore we claim that, at every monitor state $q \in Q$, the intersection of the verdict sets emitted by all local monitors (obviously the ones that have not crashed) includes only the monitor state q_c where $\delta(q, s_j) = q_c$, and we have:

$$[s_0 \cdots s_j \models_3 \varphi] = \lambda(q_c)$$

Therefore, regardless of initial partial observations of the local monitors and different read/write interleavings, a set of verdicts collectively provided by the local monitors can be used to compute the verdict computed by a centralized monitor that has full view of the system under scrutiny.

5.5 Asynchronous Automata-Based Monitoring Algorithm Using Extended Ltl$_3$ Monitor

In Algorithm 6 each monitor is required to access the shared memory at least once, i.e., $r \geqslant 1$ (see Line 3). As we observed in Section 5.4 this is in fact a necessary condition to ensure that there is at least one monitor that obtains the full view of the global state of the system before emitting its verdict. In this section we present an Algorithm that can solve our asynchronous monitoring problem without any round of communication (i.e., accessing the shared memory). This is done by employing an Extended Ltl$_3$ monitor that we introduced in Chapter 4, in each local monitor's algorithm.

```
Data: Extended Ltl3 monitor $\mathcal{M}_e^\varphi = \{\Sigma, Q_e, q_0, \delta_e, \lambda_e\}$ and state $s_j$
Result: a set of monitor states
1 Let $\mathcal{S}_i^{s_j}$ be the initial concrete local state of the monitor ;
2 $Snap_i^{s_j} \leftarrow \mathcal{S}_i^{s_j}$;                /* take sample from state $s_j$ */
3 emit $V_i^{s_j} = \mu_2(\mu_1(Snap_i^{s_j}, \mathcal{M}^\varphi))$ ;  /* emit a set of monitor states according to $\mathcal{M}_e^\varphi$ */
```

Algorithm 7: Behavior of Monitor M_i, for $i \in [1, n]$

Recall that we assumed the set of local monitors (namely, their concrete local states) satisfy the state coverage. As described in Chapter 4, an Extended LTL$_3$ monitor is constructed based on the worst case scenario where each local monitor reads the value of at most one atomic proposition, and we also recall that if each monitor uses an Extended LTL$_3$ monitor to calculate its verdict set, then we always have $|\mathcal{I}^q| = 1$ at every monitor state $q \in Q_e$, where $\mathcal{I}^q$ is the intersection of all verdict sets emitted by all monitors in $\mathcal{M}$. Therefore Algorithm 7 always computes the same verdict that is computed by a centralized monitor that has full view of the system, regardless of initial partial observations of the local monitors.

Let us look at the following example to see how Algorithm 7 is employed by local monitors to emit their verdict sets.

Example Let $\varphi = \mathbf{F}(a \wedge b)$ whose Extended LTL$_3$ monitor is given below:

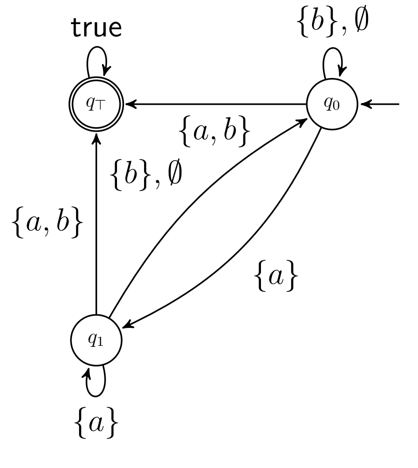


Figure 5.2: $\mathcal{M}_e^\varphi$ for $\varphi = \mathbf{F}(a \wedge b)$

Suppose monitors are at monitor state q_0, and let $s = \{a, b\}$. The following tables represent each monitor M_i's initial local snapshot $Snap_i^s$ and its verdict set V_i calculated based on only $Snap_i^s$.

$Snap_1^s$

	M_1	M_2	M_3
a	true	$\natural$	$\natural$
b	$\natural$	$\natural$	$\natural$
V_1	$\{q_1, q_\top\}$		

$Snap_2^s$

	M_1	M_2	M_3
a	$\natural$	$\natural$	$\natural$
b	$\natural$	true	$\natural$
V_2	$\{q_0, q_\top\}$		

$Snap_3^s$

	M_1	M_2	M_3
a	$\natural$	$\natural$	$\natural$
b	$\natural$	$\natural$	$\natural$
V_3	$\{q_0, q_1, q_\top\}$		

And by calculating the intersection of the verdict sets we obtain $\mathcal{I}_0^q = q_\top$, which is the verdict that a centralized monitor that has full view of state $s = \{a, b\}$ would compute. Note that a verdict form $\mathbb{B}_3$ can be emitted simply by applying the mapping function λ_e, i.e., by calculating $\lambda_e(v)$ where $v \in \mathcal{I}^q$, which in this example is $\lambda_e(q_\top) = \top$.

Chapter 6

Conclusion

6.1 Summary

In this thesis, we studied synchronous and asynchronous runtime verification of distributed systems and presented distributed monitoring algorithms for this purpose, which allow three-valued LTL monitoring. In particular,

- we proposed a synchronous monitoring algorithm that copes with f crash failures in a distributed setting. The algorithm solves the synchronous monitoring problem in $f+1$ rounds of communication, where at each round each local monitor broadcasts a message, receives messages from other monitors, and performs local computation based on the received messages and computes a message to be sent in the subsequent round. We proposed an automata-based algorithm where each local monitor's message is a set of monitor states that are reachable from the current monitor state (according to the input automaton) by the set of possible global states from viewpoint of that local monitor. Therefore our algorithm reduces the message size overhead from $|AP|$ to $\log(m_q)$ where m_q is the number of outgoing transitions from the current monitor state q. We showed that the input automaton (that is employed in each monitor's algorithm) must satisfy the following condition:

$$\forall q \in Q.\ |\mathcal{I}^q| = 1$$

where Q is the set of all monitor states in the input automaton, and $\mathcal{I}^q$ denotes the intersection of all verdict sets emitted by local monitors. Therefore we introduced an algorithm to construct an Extended LTL$_3$ monitor that satisfies the aforementioned condition.

- We proposed an algorithm for distributed crash-resilient asynchronous RV that consistently monitors the system under inspection without any communication between monitors. Each local monitor emits a verdict set solely based on its own partial observation, and the intersection of the verdict sets will be the same as the verdict computed by a centralized monitor that has full view of the system.

6.2 Future Work

Some open problems for further research are as follows:

- In our framework the fault model was crash failure, i.e., the monitors can only fail by crashing. From a more practical perspective, it would be interesting to address more severe, e.g., Byzantine failures.

- In the asynchronous monitoring, although we assumed the monitors are asynchronous wait-free processes, however, it was supposed that the global state of the system changes synchronously, i.e., all monitors observe the same global state. We can relax the timing model so that monitors observe, communicate, and emit verdicts between any two global states.
- Our results in the decentralized asynchronous monitoring can theoretically be transformed to more practical refinements such as message passing frameworks.
- It would of course be interesting to extend our results to the case where the input to the monitors is a sequence of global states and each monitor produces a sequence of verdict sets, one per each global state.

Bibliography

Agrawal, S. and Bonakdarpour, B. (2016). Runtime verification of k-safety hyperproperties in hyperltl. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium, CSF*, pages 239–252.

Attiya, H. and Welch, J. (2004). *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*. Wiley.

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

Basin, D. A., Jugé, V., Klaedtke, F., and Zalinescu, E. (2013). Enforceable security policies revisited. *ACM Transaction on Information Systems and Security*, **16**(1), 3:1–3:26.

Basin, D. A., Caronni, G., Ereth, S., Harvan, M., Klaedtke, F., and Mantel, H. (2016). Scalable offline monitoring of temporal specifications. *Formal Methods in System Design*, **49**(1-2), 75–108.

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

Bauer, A., Leucker, M., and Schallhart, C. (2010). Comparing LTL Semantics for Runtime Verification. *Journal of Logic and Computation*, **20**(3), 651–674.

Bauer, A., Leucker, M., and Schallhart, C. (2011). Runtime Verification for LTL and TLTL. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, **20**(4), 14:1–14:64.

Bonakdarpour, B., Navabpour, S., and Fischmeister, S. (2011). Sampling-based runtime verification. In *Formal Methods (FM)*, pages 88–102.

Bonakdarpour, B., Navabpour, S., and Fischmeister, S. (2013). Time-triggered runtime verification. *Formal Methods in System Design*, **43**(1), 29–60.

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

Brett, N., Siddique, U., and Bonakdarpour, B. (2017). Rewriting-based runtime verification for alternation-free hyperltl. In *Proceedings of the 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 77–93.

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

Chen, F. and Rosu, G. (2007). MOP: An Efficient and Generic Runtime Verification Framework. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 569–588.

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

d'Amorim, M. and Rosu, G. (2005). Efficient Monitoring of omega-Languages. In *Computer Aided Verification (CAV)*, pages 364–378.

Deshmukh, J. V., Donzé, A., Ghosh, S., Jin, X., Juniwal, G., and Seshia, S. A. (2015). Robust online monitoring of signal temporal logic. In *Runtime Verification - 6th International Conference, RV 2015 Vienna, Austria, September 22-25, 2015. Proceedings*, pages 55–70.

Finkbeiner, B., Hahn, C., Stenger, M., and Tentrup, L. (2017). Monitoring hyperproperties. In *Proceedings of the 17th International Conference on Runtime Verification (RV)*, pages 190–207.

Fischer, M. J., Lynch, N. A., and Peterson, M. S. (1985). Impossibility of distributed consensus with one faulty processor. *Journal of the ACM*, **32**(2), 373–382.

Fraigniaud, P., Rajsbaum, S., and Travers, C. (2013). Locality and checkability in wait-free computing. *Distributed Computing*, **26**(4), 223–242.

Fraigniaud, P., Rajsbaum, S., and Travers, C. (2014a). On the number of opinions needed for fault-tolerant run-time monitoring in distributed systems. In *Runtime Verification - 5th International Conference (RV)*, pages 92–107.

Fraigniaud, P., Rajsbaum, S., Roy, M., and Travers, C. (2014b). The opinion number of set-agreement. In *Principles of Distributed Systems - 18th International Conference (OPODIS)*, pages 155–170.

Garg, V. K. (2002). *Elements of distributed computing.* Wiley.

Havelund, K. and Rosu, G. (2001a). Monitoring Java Programs with Java PathExplorer. *Electronic Notes in Theoretical. Computer Science*, **55**(2).

Havelund, K. and Rosu, G. (2001b). Monitoring Programs Using Rewriting. In *Automated Software Engineering (ASE)*, pages 135–143.

Havelund, K. and Rosu, G. (2002). Synthesizing monitors for safety properties. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 342 –356.

Havelund, K. and Rosu, G. (2004). Efficient Monitoring of Safety Sroperties. *Software Tools and Technology Transfer (STTT)*, **6**(2), 158–173.

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

M. Herlihy, D. K. and Rajsbaum, S. (2013). *Distributed Computing Through Combinatorial Topology.* Morgan Kaufmann.

Manna, Z. and Pnueli, A. (1979). The modal logic of programs. In *Proceedings of the 6th Colloquium on Automata, Languages and Programming (ICALP)*, pages 385–409.

Manna, Z. and Pnueli, A. (1995). *Temporal verification of reactive systems - safety*. Springer.

Medhat, R., Bonakdarpour, B., Kumar, D., and Fischmeister, S. (2015). Runtime monitoring of cyber-physical systems under timing and memory constraints. *ACM Transactions on Embedded Computing Systems*, **14**(4), 79:1–79:29.

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

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

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

Navabpour, S., Wu, C. W., Bonakdarpour, B., and Fischmeister, S. (2011). Efficient techniques for near-optimal instrumentation in time-triggered runtime verification. In *International Conference on Runtime Verification (RV)*. To appear.

Navabpour, S., Bonakdarpour, B., and Fischmeister, S. (2015). Time-triggered runtime verification of component-based multi-core systems. In *Proceedings of the 6th International Conference on Runtime Verification (RV)*, pages 153–168.

Nguyen, L. V., Kapinski, J., Jin, X., Deshmukh, J. V., and Johnson, T. T. (2017).

Hyperproperties of real-valued signals. In *Proceedings of the 15th ACM-IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*, pages 104–113.

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

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

Schneider, F. B. (2000). Enforceable security policies. *ACM Transactions on Information and System Security (TISSEC)*, **3**, 30–50.

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

Sen, K., Vardhan, A., Agha, G., and Rosu, G. (2006). Decentralized runtime analysis of multithreaded applications. In *20th International Parallel and Distributed Processing Symposium (IPDPS 2006), Proceedings, 25-29 April 2006, Rhodes Island, Greece.*

Stoller, S. D. and Schneider, F. B. (1995). Verifying programs that use causally-ordered message-passing. *Sci. Comput. Program.*, **24**(2), 105–128.

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

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