RUNTIME VERIFICATION OF DISTRIBUTED SYSTEMS

By

Ritam Ganguly

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

Computer Science - Doctor of Philosophy

2023

ABSTRACT

Given the broad scale of distribution and complexity of today's system, an exhaustive model-checking algorithm is computationally costly and testing is not exhaustive enough. Runtime Verification on the other hand analyzes a developing execution, be it online or offline, of the system in order to check for the health of the system with respect to some specification. Runtime verification of distributed systems with respect to temporal specification is both critical as well as a challenging task. It is critical because it ensures the reliability of the system by detecting violations of system requirements. To guarantee the lack of violations one has to analyze every possible ordering of system events which makes it computationally expensive and hence challenging. In this dissertation, we focus on a partially synchronous distributed system, where the various components of the distributed system do not share a common global clock and a clock synchronization algorithm limits the maximum clock skew among processes to a constant. Following listed are the main contributions of this dissertation,

- We introduce two monitoring techniques where the specification in the linear temporal logic (`LTL`) is either represented by a deterministic finite automaton, or, we use a progression-based formula re-witting technique to reduce the distributed runtime verification problem to an SMT problem.
- We introduce a progression-based formula rewriting scheme for monitoring metric temporal logic (`MTL`) specifications which employ SMT-solving techniques with probabilistic guarantees.
- We introduce an (offline) SMT-based monitor synthesis algorithm, which results in minimizing the size of monitoring messages for an automata-based synchronous monitoring algorithm that copes with up to t crash monitor failures.
- We extend the stream-based specification language Lola for monitoring partially-synchronous systems and develop an (online) SMT-based decentralized monitoring technique for the same.

- All of our techniques have been tested by both extensive synthetic experiments and real-life case studies, such as a distributed database, Cassandra; an Internet-of-Things dataset of an house, Orange4Home; an Ethereum-based smart contracts; Industrial Control Systems (ICS), Secure Water Treatment (SWaT), etc.

This dissertation is dedicated to my grandparents,
Rina Ganguly and Rama Prasad Ganguly

ACKNOWLEDGMENTS

First of all I would like to thank my advisor, Dr. Borzoo Bonakdarpour, for offering me technical, financial, and moral support during the four years of my research. He introduced me to the area of runtime verification of distributed systems. Much of the results reported in this dissertation is inspired by my discussion with him about our ideas and developing a general verification approach for a wide range of distributed systems with different system specifications. He helped me understand what research is and how to solve a problem.

My dissertation guidance committee comprising of Dr. Borzoo Bonakdarpour, Dr. Sandeep Kulkarni, Dr. Eric Torng and Dr. Shaunak D. Bopardikar has been of great help, guidance and encouragement. I would like to express gratitude to Dr. Sandeep Kulkarni and Dr. Gurpur Prabhu (from Iowa State University) for giving me the exposure and motivation behind taking teaching as a career.

It has been a great pleasure to work closely with Anik Momtaz (Michigan State University) and Yingjie Xue (Brown University). They co-authored multiple papers with me on runtime verification of distributed systems with respect to `LTL` and `MTL` specifications respectively. It is impossible to mention their innumerable contributions towards my work.

I would like to truly thank the Department of Computer Science and Engineering, College of Engineering at Michigan State University and the Department of Computer Science at Iowa State University for offering me financial support through teaching assistantship for several semesters and travel grants for conference travel and registration.

I would also like to thank my family, specially Ranjan Ganguly (*baba*), Molly Ganguly (*ma*), Ranjit Ganguly (*jethu*), and Rina Ganguly (*amma*) for their continuing encouragement and support. Additionally, the continuous encouragement from Saumitra Sinha (*sinha-jethu*) and Biman Ghosh (*biman-jethu*) has enabled me to not only have a pleasant stay but also to be inspired to travel to USA to pursue my PhD.

Special thanks goes out to my colleagues at Trustworthy and Reliable Technologies (TART) laboratory Anik Momtaz, Eshita Zaman, Tzu-Han Hsu, and Oyendrila Dobe for

proof reading my papers. Finally, I would like to thank my friends Puja Agarwal, Aniket Banerjee, Abhratanu Dutta, Saptaparni Ghosh, Sayantani Ghosh, Aishwarya Mazumdar, Debrudra Mitra, and Soham Vanage; because of them my PhD journey has been enjoyable and memorable.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

Chapter 1

Introduction

1.1 Motivation

As the world moves ahead, we find ourselves surrounded by technology. At the core of this technology today, lies several intelligent, automated programs as pointed out in [SSS16]. From self-driving cars to automated smart contracts for blockchain transactions, from keeping records efficiently in a data center to maneuvering aircraft in the sky, our health, safety, well-being, and finance is managed, directed and often controlled by these 'intelligent' software. But the thing that makes these pieces of software unbiased is the same thing that makes this software vulnerable to attacks of different kinds. Since these systems work without any intervention from humans, we must verify these systems before deploying them. Any slight error in the development/deployment of this software might be the reason behind multi-million dollar losses or even loss of human lives - the very lives it was built to protect and be beneficial to.

Multiple examples of these kinds of faults can be seen in our world. As pointed out by [EP18], the Parity Multisig Wallet smart contract [Tec17] version 1.5 included a vulnerability that led to the loss of 30 million US dollars. Thus, developing an effective, safe, and fault-tolerant system is both urgent and essential to protect against possible losses, both financial and human. Furthermore, critical infrastructure such as manufacturing and distribution of power, gas, water, etc. are often the site of these attacks, which makes the

company incur a loss of around $5 million and 50 days of system downtime on average. A recent report [SP18] pointed out that such an attack often compromises the integrity of the data generated and thereby making the operator vulnerable to making sound decisions. Moreover, as identified in [LLL+17, LLLG16, LHJ+14], distributed systems are prone to distributed concurrency (DC) bugs caused by non-deterministic timing of distributed events. The results show 63% of all DC bugs surface in the presence of hardware faults such as machine crashes, network delay, timeouts, and disk errors. Additionally, 53% of DC bugs lead to explicit local or global error in widely deployed cloud-based distributed systems, *Cassandra*, *Hadoop MapReduce*, *HBase* and *Zookeeper*.

In the past few decades, achieving system-wide dependability and reliability has substantially benefited by incorporating rigorous formal methods to verify and prove the correctness of safety-critical systems as pointed out in [Bow93]. In the aviation industry, formal methods have been used to develop standards and are accepted as a part of the certification process [RTC22]. Tools such as Astrée [CCF+05] and Frama-C [KKP+15] were successfully employed to formally analyze portions of the code for several aircraft models including the current largest passenger aircraft A380 [MLD+13, SWDD09]. In social media, Facebook internally runs the INFER tool to verify selected properties, such as memory safety errors and other common bugs of their mobile apps, used by over a billion people [CDD+15]. These are some of the success stories of verification in building reliable and dependable systems identified in [HGM20]. Amazon Web Services (AWS) has included a runtime threat detection coverage for Amazon Elastic Kubernetes Service (Amazon EKS) [Ser23a] nodes and containers within the AWS environment. EKS Runtime Monitoring uses a GuardDuty [Ser23b] security agent to add runtime visibility into individual EKS workloads, file access, process execution, and network connections.

Reliability and dependability are especially critical in the domain of distributed systems that inherently consist of complex algorithms and intertwined concurrent components. Given the complexity of today's computing systems, deploying exhaustive verification techniques

such as *model checking* and *theorem proving* come at a high cost in terms of time, resources, and expertise. In many cases, formal verification is hard to scale to a realistic size to analyze the system's correctness. Moreover, exhaustive verification techniques may overlook bugs due to unanticipated stimuli from the environment, internal bugs in virtual machines, or operating systems as well as hardware faults. On the other side of the spectrum, *testing* is a best-effort method to examine the correctness, which scrutinizes only a subset of behaviors of the system. Due to its under-approximate nature, testing often does not reveal obscure corner cases that complex systems may reach at run time. In a distributed setting, the inherent uncertainty about an exponential number of orderings of events makes testing techniques often blind to concurrency bugs.

Runtime verification (RV) is a lightweight popular technique, where a monitor or a set of monitors continually inspects the health of a system under consideration at run time with respect to a formally specified set of properties. The formal specification is normally in the form of some language with clear syntax and semantics, such as regular expressions or some form of temporal logic. RV acts as a crucial complement to costly model checking and non-exhaustive testing. It often acts as a crucial bridge between how a system was designed to perform versus how the system actually performs in the presence of various external environmental factors. Compared to *model checking* and *testing*, runtime verification stands out because of its ability to verify the actual execution of the system, along with the ability to be aware of any external stimuli of the environment affecting the working of the system.

As the scale and application of distributed systems are reaching new heights, so is the complexity of verifying the correctness of these systems. To add to this complexity, we find added challenges in the form of different clock synchronization schemes adopted by the distributed system. In other words, we can classify distributed systems according to the clock synchronization schemes they follow and are mainly of two types: synchronous and asynchronous. In a synchronous system, all components of the distributed system share a common global clock. Although verifying such a system is comparatively easier,

maintaining such a system is costly with synchronization messages required to be sent at very close intervals. On the other hand, asynchronous systems involve no synchronization messages. Although it is extremely cheap to maintain such a system, verifying such a system is extremely costly since it involves checking all possible interleavings of the events. An efficient yet effective solution involves the presence of a clock synchronization algorithm that sends out clock synchronization messages after a certain time instance. This limits the clock skew between all pairs of components to a constant and thereby limiting the number of interleavings needed to verify such a system.

Motivating Examples: Consider a large geographically separated distributed database consisting of two datasets, *Student*, containing details of the student enrolled in the university and *Enrollment*, that keeps a track of the classes each student has enrolled in. The distributed nature of the database makes maintaining a common global clock shared among all the components, a challenge. Moreover, the distributed database does not maintain the normalization of data. This makes the data stored in the database vulnerable to being replicated and also promotes unrelated data to be stored in the database. For example, an entry in the *Student* table reads, (1234, "*Leslie Lamport*", "126 Spartan Drive. East Lansing. MI 48800"). This represents a student with the name *Leslie Dijkstra* and student identification number 1234, living at the corresponding residential address. On the other side, an entry in the *Enrollment* table reads, (1234, "*Edsger Dijkstra*", "*CSE 260: Discrete Mathematics*". This represents a student with the name *Leslie Dijkstra* and student identification number 1234, enrolled in the corresponding course. As can be seen, although the student identification number matches, the names does not.

In another example, we see an entry in the *Enrollment* table that reads, (2345, "*Andrew Tanenbaum*", "*CSE 410: Distributed Systems*". This represents a student with the name *Andrew Tanenbaum* and student identification number 2345, enrolled in the corresponding course, but no such entry exists in the *Student* table with the respective

student identification number. Errors like these are often common and lead to violation of the ACID (*Atomicity*, *Consistency*, *Isolation*, and *Durability*) property.

Model checking of such a distributed database would entail a large state space consisting of all possible combinations of entries in each of the datasets, along with its time of occurrence. Although it would be exhaustive and would be successful in determining the faults, it would involve a huge cost and expertise, making it a non-preferred option. On the other hand, testing, although would be cheap, detection of such an error is not guaranteed. Additionally, considering the large size of the distributed database, the design of the test cases is a headache and would involve the skill of the tester. Runtime Verification allows for achieving a balance between guaranteeing the detection of such an error once it happens, and a light-weight technique, making it one of the most preferred options.

1.2 Technical Challenges of RV of Distributed System

Monitoring distributed systems and distributed monitoring has recently gained traction [CGNM13, BF12, CF16, SVAG04, Gar02, SS95, OG07, YNV+16, VYK+17, VKTA20, BKZ12, BKMZ15, BKMZ13], as a technique to discover latent bugs in concurrent settings. Most of the above-mentioned approaches have a common assumption for the system under inspection being synchronous. All processes in a synchronous distributed system share a common global clock. As such, there exists a total ordering of the events taking place in each process, and finding the ordered trace of events is comparatively easier. The time of occurrence of each event along with any message sent-receiving events leads us toward the totally ordered trace. To give a better understanding of the challenges faced in the verification of distributed systems, Figure 1.1 represents a distributed computation consisting of two processes, P_1 and P_2. Each change in the local computation is represented by an event. For example, the events $\{e_0^1, e_1^1, e_2^1, e_3^1, e_4^1\}$ (resp. $\{e_0^2, e_1^2, e_2^2, e_3^2, e_4^2\}$) are from the process P_1 (resp. P_2). Each event is either a message sent, a message received, or a local computation. A message-send event is represented by an outgoing arrow, whereas a message-receiving event

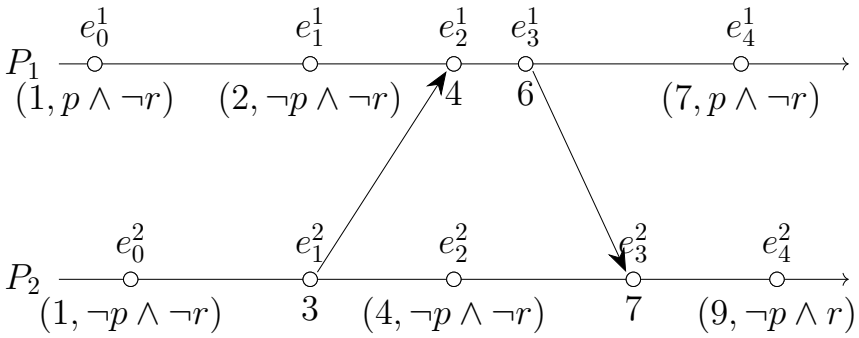
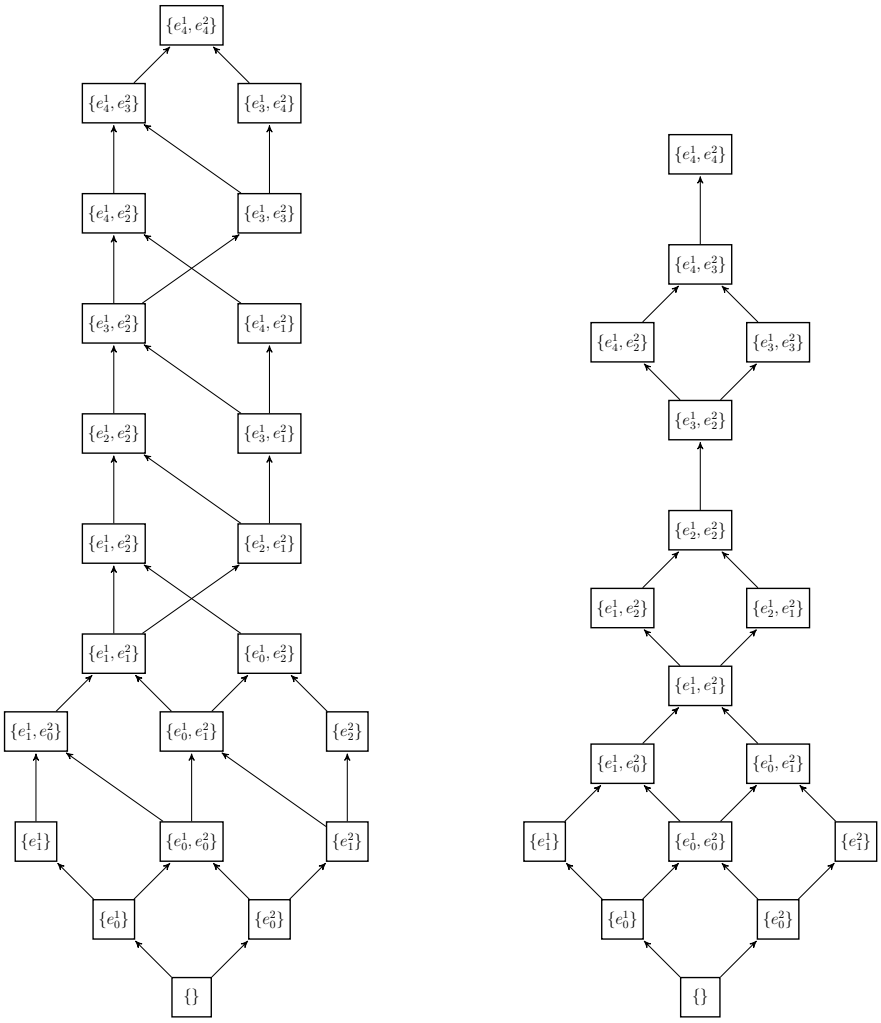


Figure 1.1: Distributed computation.

is represented by an incoming arrow. In Figure 1.1, event e_1^2 (resp. e_2^1) and event e_4^1 (resp. e_3^2) are the corresponding send (resp. receive) event. Additionally, each event is represented by a pair, consisting of the time of occurrence and the valuation of the atomic propositions, p and r. For example, event e_4^1 is represented by $(7, p \wedge \neg r)$, which denotes the time of occurrence of the event as time step 7 and the atomic proposition p is *true* whereas the atomic proposition r is *false*.

Given a distributed computation with a synchronous clock, we can form a totally ordered set by observing the time of occurrence of the events. For the events in Figure 1.1, we can order the events as $[\{e_0^1, e_0^2\}, \{e_1^1\}, \{e_1^2\}, \{e_2^1, e_2^2\}, \{e_3^1\}, \{e_3^2, e_4^1\}, \{e_4^2\}]$. An interesting observation is that since the time of occurrence of the events e_3^1 and e_3^2 is 6 and 7 respectively, we list the event e_3^1 as one that happened before the event e_3^2. This is also because e_3^1 is a sending event of a message of which e_3^2 is the receiving event and we know that a send operation strictly happens before the corresponding receiving event. Given this trace, the monitor checks for the satisfiability of the specification and generates the verdict for the given distributed computation.

With the size and complexity of distributed systems growing and with each component of a distributed system being often located at a different geographical location, maintaining a common global clock is difficult. As a result, we often find ourselves with an asynchronous distributed system, one where all the components have their local clock with no relation to each other.



(a) Considering Asynchronous System

(b) Considering Partially-Synchronous System ($\epsilon = 2$)

Figure 1.2: Computation Lattice.

Monitoring of asynchronous distributed systems, as seen in [MB15, BFR+16], does not scale well when verifying large systems. The lack of a global clock makes the time of occurrence of an event irrelevant in deciding the order of occurrence. Thus we are left with partially-ordered events. The possible number of traces of events that can be formed given the computation grows exponentially to the number of processes in the system. As can be understood, iterative monitoring of asynchronous distributed systems does not scale well.

As seen in Figure 1.2a, from the computation lattice we are able to generate

multiple traces and each trace can yield multiple verdicts. Given an LTL specification $\varphi = \bigcirc(\neg p \rightarrow (\neg p \ \mathcal{U} \ r))$ (read as 'from the next step no p should appear before an r), we can have both *true* and *false* verdict. For the traces that considers event e_0^2 appears before e_0^1, evaluate to *false*, because at event e_0^1, $\neg p$ evaluate to *false*, and we do not observe any r before that. Similarly for the traces which consider event e_0^1 happens before event e_0^2 and event e_4^2 happens before event e_4^1 satisfies the specification and there-by evaluating to a *true* verdict. This makes monitoring of asynchronous distributed systems an NP-Complete problem [Gar02] in the number of processes in the setting.

Thus, to come to a middle ground, we often find asynchronous systems use a clock synchronization algorithm (like NTP [Mil10]) that limits the maximum clock skew between any two processes in the system to a constant. This constant is known as the clock synchronization constant and is denoted by ϵ. There are mainly two main ways of synchronizing the clocks:

- **External clock synchronization:** It uses a centralized time source, such as a GPS receiver, to keep all clocks in sync. This is the most accurate way to synchronize clocks, but it requires all devices to have access to the same time source.
- **Internal clock synchronization:** It uses a peer-to-peer communication to adjust the clocks of each device relative to the others. This is less accurate than external clock synchronization, but it does not require all devices to have access to the same time source.

In this dissertation, we utilize an external clock synchronization, that limits the lattice blowup experienced when monitoring an asynchronous system to a bound. Any two events from different processes, more than ϵ time apart can be totally ordered using the time of occurrence. Any two events from different processes within ϵ time are still considered concurrent. This reduces the computation lattice by a considerable amount.

Figure 1.2b shows the computation lattice for the distributed computation in Figure 1.1 when considering partial synchrony for $\epsilon = 2$. As can be observed, the computation lattice

is considerably less. For the same LTL specification, $\varphi = \bigcirc(\neg p \rightarrow (\neg p \,\mathcal{U}\, r))$, the verdict of the monitor is *false*. For both the case, (1) event e_0^2 happened before event e_0^1 and (2) event e_0^1 happened before event e_0^2, we see that that time of occurrence of event e_4^1 and event e_4^2 dictates that event e_4^1 strictly happened before the event e_4^2 since the time of occurrence of this event is not less than ϵ. This makes the monitor compute a single verdict *false* for the given computation under partial synchrony.

1.2.1 Formal Specification

A Verification approach can only be as complete as the specification of the system properties. As identified in [Cli14], system specifications are needed to be mathematically precise and complete. Thus, we represent each event in the distributed computation by a set of predicates/propositions that reflects the values of the corresponding predicate/proposition in that event. In verification, we aim to check the conformance of these events against expected values. We express our expectations as specifications of the system. In [VYK+17, VKTA20], the authors propose a distributed predicate detection technique for partially-synchronous systems. Although predicate detection is useful to represent certain types of system specifications, it lacks the expressiveness that temporal logic offers.

Depending upon the type of system to be monitored, we decide on the logic of specification to be used. It can be selected from a wide variety of options. For example, when monitoring for mutual separation of autonomous drones or race conditions in distributed memory, the logic of choice is propositional logic. On the other hand, when monitoring more complex distributed systems such as read/write consistency in a database or priority-based train-platform allocation system, the regular predicate is of little use. We need more expressive Linear Temporal Logic (LTL) [Pnu77] in this case. Furthermore, when trying to monitor smart contracts involving a set of blockchains, transactions are usually time-bound. Such case studies require a time bounded logic, such as Metric Temporal Logic (MTL) [AH92, AH94] where each temporal operator has a time bound attached to it.

Additionally, Industrial Control Systems (ICS) require more expressive specification language that can handle aggregate functions like count, average, etc to make the Programmable Logic Controller (PLC) take well-informed, sound decisions. For monitoring such systems, we use a stream-based specification language Lola [DSS+05].

1.3 Thesis Statement

The approaches discussed above play a major role in verifying distributed systems. However, in the face of increasing size and complexity of distributed systems with evolving requirements, a real-time feasible runtime verification approach of a partially-synchronous distributed system is highly desirable. Additionally, since the verdict comes with a formal guarantee, lightweight compared to other formal verification approaches, and observant of dynamic changes in the environment affecting the working of the distributed system makes runtime verification an extremely desirable choice. However, current runtime verification approaches lack to make runtime verification appreciated in every distributed system application. We list the limitations of the present approaches and the corresponding approach we use to address them:

- Sharing a common global clock among different geographically separated components of a large distributed the system is not realistic. To limit the exponential blow-up of the computational space due to asynchrony, presence of a clock synchronization algorithm is practical: *We consider a partially-synchronous distributed system.*
- With changing requirements and a more versatile distributed system being developed, more expressive temporal logic is used to mention the specifications: *We consider specifications as temporal properties using* `LTL`*, time-bounded temporal properties using* `MTL`*, and stream-based specification language using* Lola.
- A robust monitoring approach which scales well with changing system properties: *We employ an SMT-based monitoring approach that encodes the distributed system to check for satisfaction and violation of the system property.*

- The monitoring approach should also be fault-tolerant, in other words, the verdict should be unaffected even if some components of the monitoring architecture behave faultily. *We study fault-tolerant monitoring for synchronous system.*
- The approach should finally be able to monitor the system at a similar pace as the events take place on the system under consideration: *We propose an online decentralized stream-based runtime verification approach where each monitor broadcasts a partially-evaluated* Lola *associated equation to all other monitors.*

With the above-mentioned motivation, we focus on developing a runtime verification approach that defends the following statement.

Runtime Verification of a partially-synchronous distributed system in real-time is feasible.

The contribution of our work validates the above statement. Briefly, based on the type of specification used to represent the distributed system and the type of monitoring architecture in use, we classify our contribution into four cases where (1) the specification of the system can be represented by `LTL` (2) the system is time sensitive and as a result we use `MTL` to represent the specification and (3) develop a fault-tolerant decentralized monitoring algorithm and (4) develop a decentralized runtime verification approach for Lola specification.

1.4 Contribution

We list the major contribution of our work below with the publications recorded in Table 1.1:

- **Runtime verification of partially-synchronous distributed system w.r.t. `LTL` specifications** We propose two sound and complete solutions to the problem of distributed runtime verification (RV) with respect to `LTL` formulas. Both of our solutions use a fault-proof central monitor, and in order to remedy the explosion of different interleavings, we make a practical assumption of the presence of a

clock synchronization algorithm. The first approach is based on constructing a $\texttt{LTL}_3$ automata of the LTL formula and constructing multiple SMT queries to determine which states of the monitor automaton are reachable for a given distributed computation. The other approach involves developing a formula progression technique. Specifically, given a finite trace α, and an LTL formula φ, we define a function Pr, such that $\mathsf{Pr}(\alpha, \varphi)$ characterizes the progression of φ and α. Progression is defined as the rewritten formula for future extensions of α depending on what has been observed thus far, which returns either *true*, *false*, or an LTL formula. We test our approach through not only a set of vigorous synthetic experiments but also by monitoring the same set of consistency conditions in Cassandra. We also put our approach to the test using a real-time airspace monitoring dataset (RACE) from NASA [MGS19].

- **Runtime verification of partially-synchronous distributed system w.r.t. MTL specifications** We propose a sound and complete solution to the problem of distributed runtime verification (RV) with respect to MTL formulas. We deploy a fault-proof central monitor, and in order to remedy the explosion of different interleavings, we again make a practical assumption of the presence of a clock synchronization algorithm. We introduce a progression-based formula rewriting technique that is reduced to an SMT encoding over distributed computations which takes into consideration the events observed thus far to rewrite the specifications for future extensions. Our monitoring algorithm accounts for all possible orderings of events without explicitly generating them when evaluating MTL formulas. We report on the results of rigorous experiments on monitoring synthetic data, using benchmarks in the tool UPPAAL [BDL04], as well as monitoring correctness, liveness, and conformance conditions for smart contracts on blockchains.
- **Crash-Resilient decentralized runtime verification of synchronous distributed system w.r.t. LTL specifications** We assume that a set of monitors, subject to crash failures, are distributed over a synchronous communication network.

Each monitor only has a partial view of the underlying system. In order to minimize the size of the transformed automaton, we formulate an offline optimization problem in the satisfiability modulo theory (SMT). This limits the size of the message to be $O(\log(|\mathcal{M}_3^{\varphi}|) \cdot |\mathsf{AP}|)$. We have evaluated our approach on a variety of LTL formulas for traces being generated using different random distributions as well as an IoT dataset, Orange4Home [CLRC17].

- **Decentralized stream-based runtime verification of partially-synchronous distributed system** We assume that a set of partially-synchronous set of monitors, are distributed over a partially-synchronous communication network. Each monitor only has a partial view of the entire system and utilize a message-passing based communication to share the locally computed results with other monitors. We first present a general technique for runtime monitoring of distributed applications whose behavior can be modeled as input/output *streams* with an internal computation module in the partially synchronous semantics, where an imperfect clock synchronization algorithm is assumed. Second, we propose a generalized stream-based decentralized runtime verification technique. We also rigorously evaluate our algorithm on extensive synthetic experiments and several Industrial Control Systems and aircraft SBS message datasets.

1.5 Organization

This report consists of 7 chapters. Each chapter addresses a separate aspect of runtime verification.

- We present the different preliminary concepts of a distributed system, linear temporal logic (LTL), metric temporal logic (MTL), etc. in Chapter 2.
- We introduce and discuss two solutions for monitoring partially synchronous distributed systems w.r.t. LTL specifications in Chapter 3.
- Next, we propose a monitoring solution with probabilistic guarantees for time-bounded

Chapter #	Distributed System(clock)	Specification	Monitor	Conference/Journal
3	Partially-Synchronous	LTL	Centralized	Published in OPODIS-2020
				Minor-revision in Springer-FMSD
4	Partially-Synchronous	MTL	Centralized	Published in IEEE ICDCS-2022
				Under review in Elsevier-JPDC
5	Synchronous	LTL	Decentralized	To appear in IEEE-TDSC
6	Partially-Synchronous	LOLA	Decentralized	Submitted in ACM EMSOFT-2023

Table 1.1: Summarized Publications.

temporal specifications in Chapter 4.

- In Chapter 5, we introduce a fault-tolerant decentralized monitoring approach for synchronous distributed system.
- In Chapter 6, we propose a decentralized stream-based runtime verification technique for partially-synchronous distributed systems.
- Finally, in Chapter 7 we present the related work in the literature of runtime verification of distributed systems followed by the conclusion and road map for future work in Chapter 8.

Chapter 2

Preliminary Concepts

In this chapter, we discuss and introduce the various preliminary concepts we use in the course of this report.

2.1 Distributed System

A distributed system is a computing environment where various components, often geographically separated, are spread across multiple computers (or other computing devices) on a network with the aim of achieving a common goal. These devices split up the work, coordinating their efforts to complete the job more efficiently than if a single device had been responsible for the same task.

In the scope of this report, we classify distributed system into two classes. One, where the components of the distributed system (processes) shared one common global clock, known as synchronous distributed system. Second, the components do not share a common global clock, but are synchronized by the help of a clock synchronization algorithm (eg. NTP [Mil10]), known as partially synchronous distributed system.

We assume a loosely coupled message passing system, consisting of n processes, denoted by $\mathcal{P} = \{P_1, P_2, \ldots, P_n\}$, without any shared memory. Channels are assumed to be FIFO, and lossless. In our model, each local state change is considered an event, and every message activity (send or receive) is also represented by a new event. Message transmission does

not change the local state of processes and the content of a message is immaterial to our purposes. We will need to refer to some global clock which acts as a '*real*' time keeper.

2.1.1 Synchronous Distributed System

In a synchronous distributed system, all the processes, share the global clock of the system. The *local clock* (or time) of a process P_i, is same as that of the global clock (or time) $\mathcal{G}$. Since all the processes share the global clock, a event in a process can be easily arranged by looking at the time of occurrence of the corresponding event. For any two events, e^i_σ (resp. $e^j_{\sigma'}$), occurring in process i (resp. j) at time σ (resp. σ') can be ordered using the Lamport's *happen-before relation* [Lam78] ($\rightsquigarrow$) as

$$(\sigma < \sigma') \leftrightarrow (e^i_\sigma \rightsquigarrow e^j_{\sigma'})$$

or

$$(\sigma' < \sigma) \leftrightarrow (e^j_{\sigma'} \rightsquigarrow e^i_\sigma)$$

Thus the events can be arranged in a unique ordering depending upon the time of occurrence, to form a trace, to be used for monitoring.

2.1.2 Partially-Synchronous Distributed System

A partially synchronous distributed system makes a practical assumption of *partial synchrony*. The *local clock* (or time) of a process P_i, where $i \in [1, n]$, can be represented as an increasing function $c_i : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$, where $c_i(\mathcal{G})$ is the value of the local clock at global time $\mathcal{G}$. Then, for any two processes P_i and P_j, we have $\forall \mathcal{G} \in \mathbb{R}_{\geq 0}.|c_i(\mathcal{G}) - c_j(\mathcal{G})| < \epsilon$, with $\epsilon > 0$ being the maximum *clock skew*. The value ϵ is assumed to be fixed and known by the monitor in the rest of this paper. In the sequel, we make it explicit when we refer to 'local' or 'global' time. This assumption is met by using a clock synchronization algorithm, like NTP [Mil10], to ensure bounded clock skew among all processes. It is to be understood, however, that this global clock is a theoretical object used in definitions, and is *not* available

to the processes.

An *event* in process P_i is of the form $e^i_{\tau,\sigma}$, where σ is *logical time* (i.e., a natural number) and τ is the local time at global time $\mathcal{G}$, that is, $\tau = c_i(\mathcal{G})$. We assume that for every two events $e^i_{\tau,\sigma}$ and $e^i_{\tau',\sigma'}$, we have $(\tau < \tau') \Leftrightarrow (\sigma < \sigma')$.

Definition 1. *A* distributed computation *on N processes is a tuple $(\mathcal{E}, \rightsquigarrow)$, where $\mathcal{E}$ is a set of events partially ordered by Lamport's* happened-before *($\rightsquigarrow$) relation [Lam78], subject to the partial synchrony assumption:*

- *In every process P_i, $1 \leq i \leq \mathbb{N}$, all events are totally ordered, that is,*

$$\forall \tau, \tau' \in \mathbb{R}_+ . \forall \sigma, \sigma' \in \mathbb{Z}_{\geq 0} . (\sigma < \sigma') \rightarrow (e^i_{\tau,\sigma} \rightsquigarrow e^i_{\tau',\sigma'}).$$

- *If e is a message send event in a process, and f is the corresponding receive event by another process, then we have $e \rightsquigarrow f$.*
- *For any two processes P_i and P_j, and any two events $e^i_{\tau,\sigma}, e^j_{\tau',\sigma'} \in \mathcal{E}$, if $\tau + \epsilon < \tau'$, then $e^i_{\tau,\sigma} \rightsquigarrow e^j_{\tau',\sigma'}$, where ϵ is the maximum clock skew.*
- *If $e \rightsquigarrow f$ and $f \rightsquigarrow g$, then $e \rightsquigarrow g$.* □

Definition 2. *Given a distributed computation $(\mathcal{E}, \rightsquigarrow)$, a subset of events $\mathcal{C} \subseteq \mathcal{E}$ is said to form a* consistent cut *iff when $\mathcal{C}$ contains an event e, then it contains all events that happened-before e. Formally, $\forall e \in \mathcal{E}.(e \in \mathcal{C}) \wedge (f \rightsquigarrow e) \rightarrow f \in \mathcal{C}$.* □

We represent the set of all consistent cut by $\mathbb{C}$. The *frontier* of a consistent cut $\mathcal{C}$, denoted $\mathsf{front}(\mathcal{C})$ is the set of events that happen last in the cut. $\mathsf{front}(\mathcal{C})$ is a set of e^i_{last} for each $i \in [1, |\mathcal{P}|]$ and $e^i_{last} \in \mathcal{C}$. We denote e^i_{last} as the last event in P_i such that $\forall e^i_{\tau,\sigma} \in \mathcal{E}.(e^i_{\tau,\sigma} \neq e^i_{last}) \rightarrow (e^i_{\tau,\sigma} \rightsquigarrow e^i_{last})$.

2.2 Linear Temporal Logics (LTL) for RV

Let AP be a set of *atomic propositions* and $\Sigma = 2^{\mathsf{AP}}$ be the *alphabet*. We call each element of Σ an *event*. For example, for $\mathsf{AP} = \{a, b\}$, event $s = \{\}$ means that both propositions a and b are not true in s and event $s' = \{a\}$ means that only proposition a is true in s'. A *trace* is a sequence $s_0 s_1 s_2 \cdots$, where $s_i \in \Sigma$, for every $i \geq 0$. The set of all finite (respectively, infinite) traces over Σ is denoted by Σ^* (respectively,

Σ^ω). Throughout the paper, we denote finite traces by the letter α, and infinite traces by the letter σ. For a finite trace $\alpha = s_0 s_1 \cdots s_n$, by α^i, we mean trace suffix $s_i s_{i+1} \cdots s_n$ of α.

2.2.1 Infinite-trace Semantics of LTL

The syntax and semantics of the *linear temporal logic* (LTL) [Pnu77, MP79] are defined for infinite traces. The syntax is defined by the following grammar:

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

where $\mathsf{p} \in \mathsf{AP}$, and where $\bigcirc$ and $\mathcal{U}$ are the 'next' and 'until' temporal operators respectively. We view other propositional and temporal operators as abbreviations, that is, $\mathit{true} = p \vee \neg p$, $\mathit{false} = \neg \mathit{true}$, $\varphi \rightarrow \psi = \neg\varphi \vee \psi$, $\varphi \wedge \psi = \neg(\neg\varphi \vee \neg\psi)$, $\Diamond\varphi = \mathit{true} \ \mathcal{U} \ \varphi$ (*eventually* φ), and $\Box\varphi = \neg\Diamond\neg\varphi$ (*always* φ). We denote the set of all LTL formulas by $\Phi_{\texttt{LTL}}$.

The infinite-trace semantics of LTL is defined as follows. Let $\sigma = s_0 s_1 s_2 \cdots \in \Sigma^\omega$, $i \geq 0$, and let $\models$ denote the *satisfaction* relation:

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

Also, $\sigma \models \varphi$ holds if and only if $\sigma, 0 \models \varphi$ holds.

2.2.2 Finite-trace Semantics of LTL

In the context of RV, the 3-valued LTL ($\texttt{LTL}_3$ for short) [BLS11] evaluates LTL formulas for *finite* traces, but with an eye on possible future extensions where as finite LTL, or FLTL [MP95] only takes into consideration the current trace with no eye towards the future.

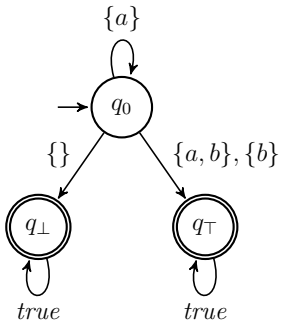


Figure 2.1: $\texttt{LTL}_3$ monitor for $\varphi = a\ \mathcal{U}\ b$.

In $\texttt{LTL}_3$, 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 finite trace extends, and '?' denotes an *unknown* verdict, i.e., there exists an extension that can violate the formula, and another extension that can satisfy the formula. Let $\alpha \in \Sigma^*$ be a non-empty finite trace. The truth value of an $\texttt{LTL}_3$ formula φ with respect to α, denoted by $[\alpha \models_3 \varphi]$, is defined as follows:

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

Definition 3. *The* $\texttt{LTL}_3$monitor *for a formula φ is the unique deterministic finite state machine $\mathcal{M}_\varphi = (\Sigma, Q, q_0, \delta, \lambda)$, where Q is the set of states, q_0 is the initial state, $\delta : Q \times \Sigma \to Q$ is the transition function, and $\lambda : Q \to \mathbb{B}_3$ is a function such that $\lambda\big(\delta(q_0, \alpha)\big) = [\alpha \models_3 \varphi]$, for every finite trace $\alpha \in \Sigma^*$.* □

For example, Fig. 2.1, shows the monitor automaton for formula $\varphi = a\ \mathcal{U}\ b$.

The syntax of FLTL is also identical to that of LTL, and the semantics is based on the truth values $\mathbb{B}_2 = \{\top, \bot\}$, where $\top$ (resp., $\bot$) denotes that the formula is satisfied (resp., violated) given the current finite trace. For atomic propositions and Boolean operators, the semantics of FLTL is identical to those of LTL. Let φ, φ_1, and φ_2 be LTL formulas, $\alpha = s_0 s_1 \dots s_n$ be a *non-empty* finite trace, and $\models_F$ denote the satisfaction relation in FLTL.

The semantics of FLTL for the temporal operators are as follows:

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

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

In order to further illustrate the difference between LTL and FLTL and LTL_3, consider formula $\varphi = \Box p$, and a finite trace $\alpha = s_0 s_1 \cdots s_n$. If $\mathsf{p} \notin s_i$ for some $i \in [0,n]$, then $[\alpha \models_3 \varphi] = \bot$, that is, the formula is permanently violated and so is the case in FLTL where, $[\alpha \models_F \varphi] = \bot$. Now, consider formula $\varphi = \Diamond p$. If $\mathsf{p} \notin s_i$ for all $i \in [0,n]$, then $[\alpha \models_3 \varphi] =?$. This is because there exist an infinite extension to α that can satisfy or violate φ in the infinite semantics of LTL. But, this is not the case in FLTL where $[\alpha \models_F \varphi] = \bot$ as it did not observe any p in the observed finite trace.

2.3 Metric Temporal Logic

Let $\mathbb{I}$ be a set of nonempty intervals over $\mathbb{Z}_{\geq 0}$. We define an interval, $\mathcal{I}$, to be

$$[start, end) \triangleq \{a \in \mathbb{Z}_{\geq 0} \mid start \leq a < end\}$$

where $start \in \mathbb{Z}_{\geq 0}$, $end \in \mathbb{Z}_{\geq 0} \cup \{\infty\}$ and $start < end$. We define AP as the set of all *atomic propositions*, and $\Sigma = 2^{\mathsf{AP}}$ as the set of all possible *states*. A *trace* is represented by a pair which consists of a sequence of states, denoted by $\alpha = s_0 s_1 \cdots$, where $s_i \in \Sigma$ for every $i > 0$ and a sequence of non-negative numbers, denoted by $\bar{\tau} = \tau_0 \tau_1 \cdots$, where $\tau_i \in \mathbb{Z}_{\geq 0}$ for all $i > 0$. We represent the set of all infinite traces by a pair of infinite sets, $(\Sigma^\omega, \mathbb{Z}_{\geq 0}^\omega)$. The trace $s_k s_{k+1} \cdots$ (resp. $\tau_k \tau_{k+1}$) is represented by α^k (resp. τ^k). For an infinite trace $\alpha = s_0 s_1 \cdots$ and $\bar{\tau} = \tau_0 \tau_1 \cdots$, $\bar{\tau}$ is an increasing sequence, meaning $\tau_{i+1} \geq \tau_i$, for all $i \geq 0$.

Syntax The syntax of metric temporal logic (MTL) [AH92, AH94] for infinite traces are defined by the following grammar:

$$\varphi ::= p \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \,\mathcal{U}_{\,\mathcal{I}} \varphi_2$$

where $p \in \mathsf{AP}$ and $\mathcal{U}_{\,\mathcal{I}}$ is the 'until' temporal operator with time bound $\mathcal{I}$. We also have $\mathit{true} = p \vee \neg p$, $\mathit{false} = \neg \mathit{true}$, $\varphi_1 \rightarrow \varphi_2 = \neg\varphi_1 \vee \varphi_2$, $\varphi_1 \wedge \varphi_2 = \neg(\neg\varphi_1 \vee \neg\varphi_2)$, $\diamondsuit_{\mathcal{I}}\, \varphi = \mathit{true}\, \mathcal{U}_{\,\mathcal{I}} \varphi$ ("eventually") and $\square_{\mathcal{I}}\, \varphi = \neg(\diamondsuit_{\mathcal{I}}\, \neg\varphi)$ ("always"). The set of all MTL formulas is denoted by $\Phi_{\texttt{MTL}}$.

Semantics The semantics of metric temporal logic (MTL) is defined over $\alpha = s_0 s_1 \cdots$ and $\bar{\tau} = \tau_0 \tau_1 \cdots$ as follows:

$$\begin{aligned}
(\alpha, \bar{\tau}, i) &\models p && \text{iff} \quad p \in s_i \\
(\alpha, \bar{\tau}, i) &\models \neg\varphi && \text{iff} \quad (\alpha, \bar{\tau}, i) \not\models \varphi \\
(\alpha, \bar{\tau}, i) &\models \varphi_1 \vee \varphi_2 && \text{iff} \quad (\alpha, \bar{\tau}, i) \models \varphi_1 \vee (\alpha, \bar{\tau}, i) \models \varphi_2 \\
(\alpha, \bar{\tau}, i) &\models \varphi_1 \,\mathcal{U}_{\,\mathcal{I}} \varphi_2 && \text{iff} \quad \exists j \geq i . \tau_j - \tau_i \in \mathcal{I} \wedge (\alpha, \bar{\tau}, j) \models \\
& && \qquad \varphi_2 \wedge \forall k \in [i, j), (\alpha, \bar{\tau}, k) \models \varphi_1
\end{aligned}$$

Also, $(\alpha, \bar{\tau}) \models \varphi$ holds if and only if $(\alpha, \bar{\tau}, 0) \models \varphi$.

In the context of RV, we introduce the notion of finite MTL. The truth values are represented by the set $\mathsf{B}_2 = \{\top, \bot\}$, where $\top$ (resp. $\bot$) represents a formula that is satisfied (resp. violated) given a finite trace. We represent the set of all finite traces by a pair of finite sets, $(\Sigma^*, \mathbb{Z}^*_{\geq 0})$. For a finite trace, $\alpha = s_0 s_1 \cdots s_n$ and $\bar{\tau} = \tau_0 \tau_1 \cdots \tau_n$ the only semantic that needs to be redefined is that of $\mathcal{U}$ ('until') and is as follows:

$$[(\alpha, \bar{\tau}, i) \models_F \varphi_1 \,\mathcal{U}_{\,\mathcal{I}} \varphi_2] = \begin{cases} \top & \text{if } \exists j \geq i . \tau_j - \tau_i \in \mathcal{I} ([\alpha^j \models_F \varphi_2] = \top) \wedge \\ & \quad \forall k \in [i, j) : ([\alpha^k \models_F \varphi_1] = \top) \\ \bot & \text{otherwise.} \end{cases}$$

In order to further illustrate the difference between MTL and finite MTL, consider formula $\varphi = \Diamond_{\mathcal{I}} p$ and a trace $\alpha = s_0 s_1 \cdots s_n$ and $\bar{\tau} = \tau_0 \tau_1 \cdots \tau_n$. We have $[(\alpha, \bar{\tau}) \models_F \varphi] = \top$ if for some $j \in [0, n]$, we have $\tau_j - \tau_0 \in \mathcal{I}$ and $p \in s_i$, otherwise $\bot$. Now, consider formula $\varphi = \Box_{\mathcal{I}} p$. We have $[(\alpha, \bar{\tau}) \models_F \varphi] = \bot$, if for some $j \in [0, n]$, we have $\tau_j - \tau_0 \in \mathcal{I}$ and $p \notin s_i$, otherwise $\top$.

2.4 Hybrid Logical Clocks

A *hybrid logical clock* (HLC) [KDM+14] is a tuple (τ, σ, ω) for detecting one-way causality, where τ is the local time, σ ensures the order of send and receive events between two processes, and ω indicates causality between events. Thus, in the sequel, we denote an event by $e^i_{\tau,\sigma,\omega}$. More specifically, for a set $\mathcal{E}$ of events:

- τ is the local clock value of events, where for any process P_i and two events $e^i_{\tau,\sigma,\omega}, e^i_{\tau',\sigma',\omega'} \in \mathcal{E}$, we have $\tau < \tau'$ iff $e^i_{\tau,\sigma,\omega} \rightsquigarrow e^i_{\tau',\sigma',\omega'}$.
- σ stipulates the logical time, where:
  - For any process P_i and any event $e^i_{\tau,\sigma,\omega} \in \mathcal{E}$, τ never exceeds σ, and their difference is bounded by ϵ (i.e, $\sigma - \tau \leq \epsilon$).
  - For any two processes P_i and P_j, and any two events $e^i_{\tau,\sigma,\omega}, e^j_{\tau',\sigma',\omega'} \in \mathcal{E}$, where event $e^i_{\tau,\sigma,\omega}$ receiving a message sent by event $e^j_{\tau',\sigma',\omega'}$, σ is updated to $\max\{\sigma, \sigma', \tau\}$. The maximum of the three values are chosen to ensure that σ remains updated with the largest τ observed so far. Observe that σ has similar behavior as τ, except the communication between processes has no impact on the value of τ for an event.
- $\omega : \mathcal{E} \to \mathbb{Z}_{\geq 0}$ is a function that maps each event in $\mathcal{E}$ to the causality updates, where:
  - For any process P_i and a send or local event $e^i_{\tau,\sigma,\omega} \in \mathcal{E}$, if $\tau < \sigma$, then ω is incremented. Otherwise, ω is reset to 0.
  - For any two processes P_i and P_j and any two events $e^i_{\tau,\sigma,\omega}, e^j_{\tau',\sigma',\omega'} \in \mathcal{E}$, where event $e^i_{\tau,\sigma,\omega}$ receiving a message sent by event $e^j_{\tau',\sigma',\omega'}$, $\omega(e^i_{\tau,\sigma,\omega})$ is updated based

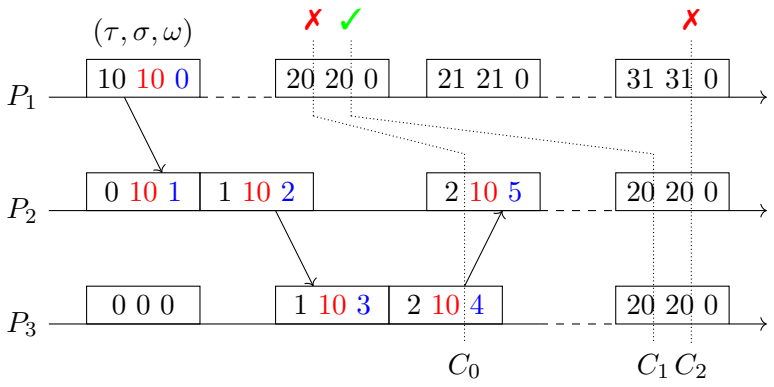


Figure 2.2: HLC example.

on $\max\{\sigma, \sigma', \tau\}$.

– For any two processes P_i and P_j, and any two events $e^i_{\tau,\sigma,\omega}, e^j_{\tau',\sigma',\omega'} \in \mathcal{E}$, $(\tau = \tau') \wedge (\omega < \omega') \rightarrow e^i_{\tau,\sigma,\omega} \rightsquigarrow e^j_{\tau',\sigma',\omega'}$.

In our implementation of HLC, we assume that it is fault-proof. Fig. 2.2 shows an HLC incorporated partially synchronous concurrent timelines of three processes with $\varepsilon = 10$. Observe that the local times of all events in $\mathsf{front}(C_1)$ are bounded by ε. Therefore, C_1 is a consistent cut, but C_0 and C_2 are not.

2.5 Stream-based Specification Lola

A Lola [DSS+05] specification describes the computation of output streams given a set of input streams. A *stream* α of type T is a finite sequence of values, $t \in \mathsf{T}$. Let $\alpha(i)$, where $i \geq 0$, denote the value of the stream at time stamp i. We denote a stream of finite length (resp. infinite length) by T^* (resp. T^ω).

Definition 4. *A* Lola *specification is a set of equations over typed stream variables of the form:*

$$\begin{aligned} s_1 &= e_1(t_1, \cdots, t_m, s_1, \cdots, s_n) \\ &\vdots \quad \vdots \\ s_n &= e_n(t_1, \cdots, t_m, s_1, \cdots, s_n) \end{aligned}$$

where $s_1, s_2, \cdots, s_n$ *are called the* dependent variables, $t_1, t_2, \cdots, t_m$ *are called*

the independent variables*, and* $e_1, e_2, \cdots, e_n$ *are the* stream expressions *over* $s_1, \cdots, s_n, t_1, \cdots, t_m$. □

Typically, *Input* streams are referred to as independent variables, whereas *output* streams are referred as dependent variable. A stream expression is constructed as follows:

- If c is a constant of type T, then c is an atomic stream expression of type T
- If s is a stream variable of type T, then s is an atomic stream expression of type T.
- If $f : \mathsf{T}_1 \times \mathsf{T}_2 \times \cdots \mathsf{T}_k \rightarrow \mathsf{T}$ is a k-ary operator and for $1 \leq i \leq k$, e_i is an expression of type T_i, then $f(e_1, e_2, \cdots, e_k)$ is a stream expression of type T
- If b is a stream expression of type *boolean* and e_1, e_2 are stream expressions of type T, then $ite(b, e_1, e_2)$ is a stream expression of type T, where *ite* is the abbreviated form of *if-then-else*.
- If e is a stream expression of type T, c is a constant of type T and i is an integer, then $e[i, c]$ is a stream expression of type T. $e[i, c]$ refers to the value of the expression e offset by i positions from the current position. In case the offset takes it beyond the end or before the beginning of the stream, then the *default* value is c.

For example, consider the following LOLA specification, where t_1 and t_2 are independent stream variables of type boolean and t_3 is an independent stream variable of type integer.

$$s_1 = \mathit{true}$$

$$s_2 = t_3$$

$$s_3 = t_1 \vee (t_3 \leq 1)$$

$$s_4 = ((t_3)^2 + 7) \mod 15$$

$$s_5 = ite(s_2, s_4, s_4 + 1)$$

$$s_6 = ite(t_1, t_3 \leq s_4, \neg s_3)$$

$$s_7 = t_1[+1, \mathit{false}]$$

$$s_8 = t_1[-1, \mathit{true}]$$

$$s_9 = s_9[-1, 0] + (t_3 \mod 2)$$

$$s_{10} = t_2 \vee (t_1 \wedge s_{10}[1, \mathit{true}])$$

where, *ite* is the abbreviated form of *if-then-else* and stream expressions s_7 and s_8 refers to the stream t_1 with an offset of $+1$ and -1, respectively.

Furthermore, LOLA can be used to compute *incremental statistics*, where a given a stream, α, a function, $f_\alpha(v, u)$, computes a measure, where u represents the measure thus far and v, the current value. Given a sequence of values, $v_1, v_2, \cdots, v_n$, with a default value d, the measure over the data is given as

$$u = f_\alpha(v_n, f_\alpha(v_{n-1}, \cdots, f_\alpha(v_1, d)))$$

Example of such functions include *count*, $f_{\mathit{count}}(v, u) = u + 1$, *sum*, $f_{\mathit{sum}}(v, u) = u + v$, *max*, $f_{\mathit{max}}(v, u) = \max\{v, u\}$, among others. Aggregate functions like *average*, can be defined using two incremental functions, *count* and *sum*.

The semantics of LOLA specifications is defined in terms of the evaluation model, which describes the relation between input and output streams.

Definition 5. *Given a* LOLA *specification φ over independent variables, $t_1, \cdots, t_m$, of type, $\mathsf{T}_1, \cdots, \mathsf{T}_m$, and dependent variables, $s_1, \cdots, s_n$ with type, $\mathsf{T}_{m+1}, \cdots, \mathsf{T}_{m+n}$, let $\tau_1, \cdots, \tau_m$ be the streams of length $N+1$, with τ_i of type T_i. The tuple $\langle \alpha_1, \cdots, \alpha_n \rangle$ of streams of length $N + 1$ is called the* evaluation model*, if for every equation in φ*

$$s_i = e_i(t_1, \cdots, t_m, s_1, \cdots, s_n)$$

$\langle \alpha_1, \cdots, \alpha_n \rangle$ satisfies the following associated equations:

$$\alpha_i(j) = \upsilon(e_i)(j) \qquad \textit{for } (1 \leq i \leq n) \wedge (0 \leq j \leq N)$$

where $\upsilon(e_i)(j)$ is defined as follows. For the base cases:

$$\upsilon(c)(j) = c$$

$$\upsilon(t_i)(j) = \tau_i(j)$$
$$\upsilon(s_i)(j) = \alpha_i(j)$$

For the inductive cases, where f is a function (e.g., arithmetic):

$$\upsilon\Big(f(e_1, \cdots, e_k)\Big)(j) = f\Big(\upsilon(e_1)(j), \cdots, \upsilon(e_k)(j)\Big)$$
$$\upsilon\Big(ite(b, e_1, e_2)\Big)(j) = \mathsf{if}\ \upsilon(b)(j)\ \mathsf{then}\ \upsilon(e_1)(j) \mathsf{else}\ \upsilon(e_2)(j)$$
$$\upsilon(e[k, c])(j) = \begin{cases} \upsilon(e)(j+k) & \text{if } 0 \leq j+k \leq N \\ c & \text{otherwise} \end{cases} \quad \blacksquare$$

The set of all equations associated with φ is noted by φ_α.

Definition 6. *A* dependency graph *for a* Lola *specification, φ is a weighted and directed graph $G = \langle V, E\rangle$, with vertex set $V = \{s_1, \cdots, s_n, t_1, \cdots, t_m\}$. An edge $e : \langle s_i, s_k, w\rangle$ (resp. $e : \langle s_i, t_k, w\rangle$) labeled with a weight w is in E iff the equation for $\alpha_i(j)$ in φ_α contains $\alpha_k(j+w)$ (resp. $\tau_k(j+w)$) as a subexpression. Intuitively, an edge records that s_i at a particular position depends on the value of s_k (resp. t_k), offset by w positions.*

Given a set of synchronous input streams $\{\alpha_1, \alpha_2, \cdots, \alpha_m\}$ of respective type $\mathbb{T} = \{\mathsf{T}_1, \mathsf{T}_2, \cdots, \mathsf{T}_m\}$ and a Lola specification, φ, we evaluate the Lola specification, given by:

$$(\alpha_1, \alpha_2, \cdots, \alpha_m) \models_S \varphi$$

given the above semantics, where $\models_S$ denotes the synchronous evaluation.

Chapter 3

Runtime Verification for Linear Temporal Specifications

3.1 Introduction

The main challenge with distributed monitoring lies within the fact that in the absence of a global clock, it is not always possible for the monitor to establish the correct order of occurrence of events across different processes. In fact, given the non-deterministic nature of distributed applications, it is perfectly foreseeable that a runtime monitor may produce different verdicts for the same distributed computation based on different ordering of events. In the case of complete asynchrony, this in turn results in a combinatorial blow-up of possibilities that the monitor must explore at run time, which in turn makes the problem computationally expensive. However, state-of-the-art networks, such as Google Spanner are augmented with clock synchronization techniques that result in partial-synchrony [CDE+13]. These clock synchronization techniques guarantee a maximum clock-skew of ε between any pair of processes. Having such a guarantee considerably limits the combinatorial blow-up,

(Published) Ritam Ganguly, Anik Momtaz, and Borzoo Bonakdarpour, Distributed Runtime Verification Under Partial Synchrony, 24th International Conference on Principles of Distributed Systems (OPODIS 2020).
(Under minor-review) Ritam Ganguly, Anik Momtaz, and Borzoo Bonakdarpour, Runtime Verification of Partially-Synchronous Distributed System, Springer Formal Methods in System Design.

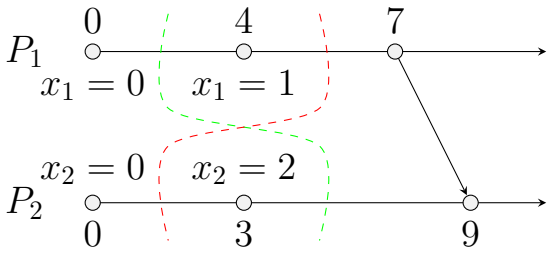


Figure 3.1: Distributed computation.

as events outside the window of ε can be ordered.

To give an example of the blow-up experienced by the monitor, consider Figure 3.1, where we have two processes P_1 and P_2 hosting two discrete variables x_1 and x_2, respectively. Let us also consider the linear temporal logic (LTL) property $\varphi = \bigcirc(x_2 > x_1)$ and a maximum clock-skew, also known as clock-synchronization constant, to be $\varepsilon = 2$. Events $x_1 = 1$ and $x_2 = 0$, as well as $x_1 = 0$ and $x_2 = 2$, are not considered concurrent, as the events in these event pairs are more than ε time apart. However, events $x_1 = 1$ and $x_2 = 2$ are considered concurrent, as these events occurred within ε time from one another. Therefore, it is not possible to determine the exact ordering of these events, without a global clock. Thus, the formula gets evaluated to both true and false, as both possible ordering of events must be taken into account. The number of different possible ordering of events can increase dramatically as more events and processes are introduced.

Handling concurrent events generally results in combinatorial enumeration of all possibilities and, hence, intractability of distributed RV. Existing distributed RV techniques operate in two extremes: they either assume a global clock [BF16b], which is unrealistic for large-scale distributed settings or assume complete asynchrony [OG07, MB15], which do not scale well. To further elaborate on our point, consider the processes P_1 and P_2 in Fig. 3.2, with events $\{e_0^1, e_1^1, e_2^1, e_3^1, e_4^1, e_5^1\}$ on process P_1, and events $\{e_0^2, e_1^2, e_2^2, e_3^2, e_4^2\}$ on process P_2 divided into two segments, seg_1 and seg_2, and a LTL formula,

$$\varphi = \bigcirc \Big(\Diamond r \rightarrow (\neg p \, \mathcal{U} \, r) \Big).$$

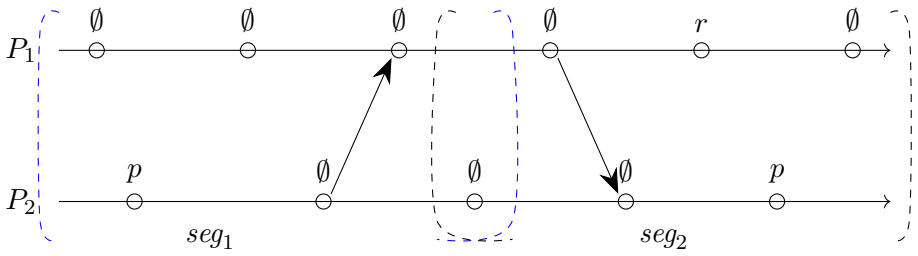


Figure 3.2: Distributed computation.

Observe that the predicate p (resp. r) is true at events e_0^2 and e_4^2 (resp. e_4^1), and in the rest of the events both predicates are false, denoted by $\emptyset$. The scenario where e_0^2 happens before e_0^1 and e_4^1 happens before e_4^2, the LTL property, φ, is satisfied. However, the scenario where e_0^1 happens before e_0^2 and e_4^1 happens after e_4^2, violates φ.

Thus, following the above example, the main research problem we aim to tackle in this paper is the following. Given a finite distributed computation and an LTL formula, our objective is to design efficient algorithms that determine whether or not the computation satisfies the formula. As shown above, the main obstacle is solving this problem is the explosion of interleavings at run time that need to be explored in order to monitor a computation.

Contributions In order to address the combinatorial explosion of various interleavings introduced by the absence of a global clock, our first design choice is a practical assumption, namely, a bounded skew of ε between local clocks of each pair of processes, which is guaranteed by a clock synchronization mechanism (e.g., NTP [Mil10]).

Our first technique is based on constructing the LTL_3 [BLS11] monitor automaton of an LTL formula and constructing multiple SMT queries to determine which states of the monitor automaton are reachable for a given distributed computation. For example, Fig. 3.3 shows the monitor automaton for formula φ mentioned earlier and one has to construct 4 different SMT queries to determine the set of all possible reachable states at the end of the computation in Fig. 3.2. We transform our monitoring decision problem into an SMT solving problem. The SMT instance includes constraints that encode (1) our monitoring algorithm

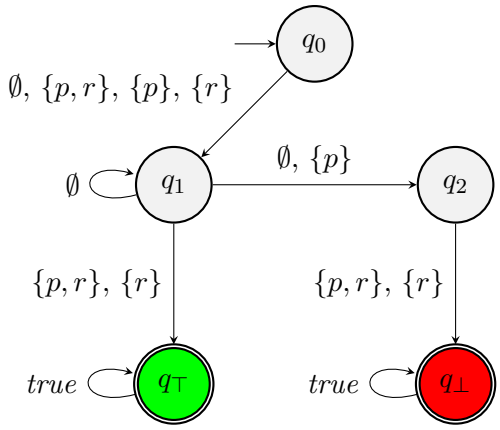


Figure 3.3: Monitor automaton for formula φ.

based on the 3-valued semantics of LTL [BLS11], (2) behavior of communicating processes and their local state changes in terms of a distributed computation, and (3) the happened-before relation subject to the ϵ clock skew assumption. Then, it attempts to concretize an uninterpreted function whose evaluation provides the possible verdicts of the monitor with respect to the given computation. In order to make the verification problem tractable, we chop a computation into multiple segments and effectively reduce the search space of each SMT query (see Fig. 3.4). Thus, the result of monitoring each segment (the possible $\mathtt{LTL}_3$ states) should be carried to the next segment. Furthermore, given the fact that distributed applications nowadays run on massive cloud services, we extend our solution to a parallel monitoring algorithm to utilize the available computing infrastructure and achieve better scalability.

The intuition behind our second monitoring technique is that since (in the first approach) running SMT queries to test whether each state of the $\mathtt{LTL}_3$ monitor automaton is reachable is excessive, it should be sufficient to test whether temporal sub-formulas of an LTL formula hold in a distributed computation. Similar to the first approach, we utilize segmentation, to break down the problem size. In the second, approach to carry the result of monitroing from one segment to the next, we also develop a *formula progression* technique. Specifically, given a finite trace α, and an LTL formula φ, we define a function Pr, such that $\mathsf{Pr}(\alpha, \varphi)$ characterizes the *progression* of φ and α. Progression is defined as the rewritten formula for

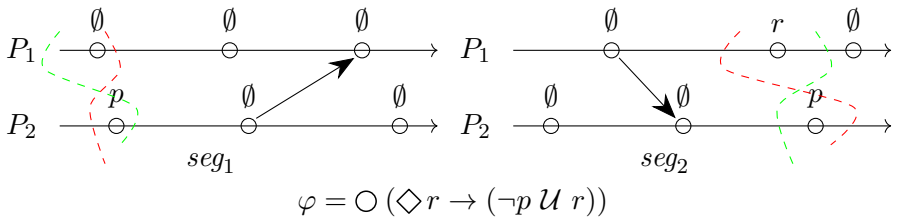


Figure 3.4: Progression and segmentation.

future extensions of α depending on what has been observed thus far, which returns either *true*, *false*, or an LTL formula. We emphasize that the main difference between our technique and the classic rewriting technique [HR01a] is that, function Pr takes a finite trace as input while the algorithm in [HR01a] rewrites the input LTL formula in a state-by-state manner. This means that in our setting, rewriting based on the fixed point representation of temporal operators is not possible. Our motivation is due to the fact that when a given distributed computation is chopped into a number of segments then a state-by-state rewriting approach would incur too many SMT queries, making it unscalable. For example, in Fig. 3.4 (which is the computation in Fig. 3.2 chopped to two segments), our progression-based approach needs the same 4 SMT queries for seg_1 (2 for each of the sub-formulas $\Diamond r$ and $\Box(\neg p)$). The evaluation yields formulas $\neg(\Diamond r)$ and $\Diamond r \rightarrow (\neg p \,\mathcal{U}\, r)$ as the possible formulas and as a result we only need to build 4 SMT queries in seg_2 compared to 5 for the automata-based approach.

Our method is fully implemented and the datasets generated during and/or analysed during the current study are available in *https://github.com/TART-MSU/dist-ltl-rv*. We make a detailed comparison between the proposed approaches in this paper through not only a set of vigorous synthetic experiments, but also monitoring the same set of consistency conditions in Cassandra. We also put our approach to test using a real-time airspace monitoring dataset (RACE) from NASA [MGS19]. Our experiments show that the progression-based approach has 35% reduced overhead (See Section 3.5 as compared to the automata-based approach.

In summary, the main contributions of this paper is as follows:

- We transform our monitoring decision problem into an SMT problem, to make for an efficient yet correct approach to consider different interleavings. Given an LTL formula, our solution provides all possible verdicts on a given computation.
- We present two monitoring approaches to address the challenges (mentioned earlier) of distributed runtime verification with regard to LTL formulas under a partially synchronous setting. In our first approach, we keep track of the observed events and the possible future outcomes by employing an automata-based technique. In our second approach, we employ a more efficient progression-based technique, where we rewrite the given LTL specifications based on the current observations. For both of our approaches, we consider a fault-proof central monitor.
- We divide a given computation into multiple segments in order to make the verification problem tractable, and as a result, significantly reduce the search space of each SMT query. Furthermore, we parallelize our monitoring technique in order to utilize the available computational resources and gain greater scalability.
- Finally, we explore and report on extensive comparisons between our automata-based approach and our progression-based approach in terms of runtime and complexity.

3.1.1 Problem Statement

Given a distributed computation $(\mathcal{E}, \rightsquigarrow)$, a *valid* sequence of consistent cuts is of the form $C_0C_1C_2\cdots$, where for all $i \geq 0$, (1) C_i denotes a set of events included in the consistent cut, (2) C_i is a subset of its succeeding consistent cut, C_{i+1}, that is, $C_i \subset C_{i+1}$, and (3) C_{i+1} has one additional event compared to its preceding consistent cut C_i, that is, $|C_i|+1 = |C_{i+1}|$. Let $\mathcal{C}$ denote the set of all valid sequences of consistent cuts. We define the set of all traces of $(\mathcal{E}, \rightsquigarrow)$ as follows:

$$\mathsf{Tr}(\mathcal{E}, \rightsquigarrow) = \Big\{\mathsf{front}(C_0)\mathsf{front}(C_1)\cdots \mid C_0C_1C_2\cdots \in \mathcal{C}\Big\}.$$

Now for our automata-based approach (resp. progression-based approach), the evaluation of the LTL formula φ with respect to $(\mathcal{E}, \rightsquigarrow)$ in the 3-valued semantics (resp. finite semantics) is the following:

$$[(\mathcal{E}, \rightsquigarrow) \models_3 \varphi] = \Big\{ \alpha \models_3 \varphi \mid \alpha \in \mathsf{Tr}(\mathcal{E}, \rightsquigarrow) \Big\}$$

and

$$[(\mathcal{E}, \rightsquigarrow) \models_F \varphi] = \Big\{ \alpha \models_F \varphi \mid \alpha \in \mathsf{Tr}(\mathcal{E}, \rightsquigarrow) \Big\}$$

respectively. This means evaluating a distributed computation with respect to a formula results in a *set* of verdicts, as a computation may involve several traces.

3.2 Formula Progression for LTL

In a synchronous system, verification on a computation can be performed in a state by state approach due to the existence of a total ordering of events [BF16a]. However, in a *partially* synchronous system, no such total ordering of events is possible. A distributed computation $(\mathcal{E}, \rightsquigarrow)$ may have different partial ordering of events dictated by different interleaving of events. Therefore, it is possible to obtain multiple verdicts on the same distributed computation $(\mathcal{E}, \rightsquigarrow)$. In order to explore these verdicts, we propose a monitoring approach based on *formula progression* that, if possible, *partially* evaluates a formula on the current computation, and based on the verdict, provides a *rewritten* formula that is to be evaluated on the extensions of the computation. As an example, let us consider the formula to be monitored as, $\varphi = \Diamond(a \rightarrow \Diamond b)$. Now, if in some trace in a computation, the monitor observes a, then for the extensions of computations, it is enough to monitor the rewritten formula, $\varphi' = \Diamond b$, as the final verdict is no longer dependent on the occurrence of a. We call this method of rewriting formula **Progression**, which we discuss in length in the following section.

Definition 7. *A* progression function $\mathsf{Pr} : \Sigma^* \times \Phi_{LTL} \rightarrow \Phi_{LTL}$ *is one that for all finite traces* $\alpha \in \Sigma^*$*, infinite traces* $\sigma \in \Sigma^\omega$*, and formulas* $\varphi \in \Phi_{LTL}$*, we have:* $\alpha\sigma \models \varphi$ *iff and only if* $\sigma \models \mathsf{Pr}(\alpha, \varphi)$. □

We emphasize that the main difference between our technique and the classic rewriting technique [HR01a] is that, function Pr takes a finite traces as input, while the algorithm in [HR01a] rewrite the input LTL formula in a state-by-state manner. This means that rewriting based on the fixed point representation of temporal operators is not possible. The motivation for our approach comes from the fact the a given distributed computation is chopped into a number of *segments*, and verification of each segment is handled by an SMT query. A state by state approach would incur too many SMT queries, making it unscalable.

Remark 1. *It is straightforward to see that for any $\alpha \in \Sigma^*$ and $\varphi \in \Phi$, if a progression function returns a non-trivial formula, which we denote by $\mathsf{Pr}(\alpha, \varphi) = \varphi'$ for some $\varphi' \in \Phi_{LTL}$, then the verdict of monitoring is* unknown.

Atomic propositions. Let $\varphi = p$ for some $p \in \mathsf{AP}$. The verdict is provided depending upon whether or not $p \in \alpha(0)$. This is the only case where the output of Pr cannot be a rewritten formula; the possible verdicts are either *true* or *false*:

$$\mathsf{Pr}(\alpha, \varphi) = \begin{cases} \mathit{true} & \text{if} \quad p \in \alpha(0) \\ \mathit{false} & \text{if} \quad p \notin \alpha(0) \end{cases}$$

Negation. Let $\varphi = \neg\phi$. We have $\mathsf{Pr}(\alpha, \varphi) = \neg\mathsf{Pr}(\alpha, \phi)$.

Disjunction. Let $\varphi = \varphi_1 \vee \varphi_2$. If either sub-formula φ_1 or φ_2 is evaluated to *false*, then the progression of φ becomes the other sub-formula φ_2 or φ_1 respectively, since that will be the only responsible sub-formula for the verdict of all future computations:

$$\mathsf{Pr}(\alpha, \varphi) = \begin{cases} \mathit{true} & \text{if} \quad \mathsf{Pr}(\alpha, \varphi_1) = \mathit{true} \ \vee \ \mathsf{Pr}(\alpha, \varphi_2) = \mathit{true} \\ \mathit{false} & \text{if} \quad \mathsf{Pr}(\alpha, \varphi_1) = \mathit{false} \ \wedge \ \mathsf{Pr}(\alpha, \varphi_2) = \mathit{false} \\ \varphi_2' & \text{if} \quad \mathsf{Pr}(\alpha, \varphi_1) = \mathit{false} \ \wedge \ \mathsf{Pr}(\alpha, \varphi_2) = \varphi_2' \\ \varphi_1' & \text{if} \quad \mathsf{Pr}(\alpha, \varphi_2) = \mathit{false} \ \wedge \ \mathsf{Pr}(\alpha, \varphi_1) = \varphi_1' \\ \varphi_1' \vee \varphi_2' & \text{if} \quad \mathsf{Pr}(\alpha, \varphi_1) = \varphi_1' \ \wedge \ \mathsf{Pr}(\alpha, \varphi_2) = \varphi_2' \end{cases}$$

Next operator. Let $\varphi = \bigcirc\phi$. The verdicts *true*, *false* and ϕ' can only be reached if α^1 is not an empty trace, that is, $\mid \alpha^1 \mid \neq 0$. Otherwise, if we are at the last event in the trace, then the progression of φ becomes ϕ; implying ϕ must hold at the beginning of the future extension:

$$\mathsf{Pr}(\alpha, \varphi) = \begin{cases} \mathit{true} & \text{if} \quad \mathsf{Pr}(\alpha^1, \phi) = \mathit{true} \wedge \mid \alpha^1 \mid \neq 0 \\ \mathit{false} & \text{if} \quad \mathsf{Pr}(\alpha^1, \phi) = \mathit{false} \wedge \mid \alpha^1 \mid \neq 0 \\ \phi' & \text{if} \quad \mathsf{Pr}(\alpha^1, \phi) = \phi' \wedge \mid \alpha^1 \mid \neq 0 \\ \phi & \text{if} \quad \mid \alpha^1 \mid = 0 \end{cases}$$

Always and eventually operators. Progression in the temporal operator 'always', $\Box$ (resp. 'eventually', $\Diamond$) may yield *false* (resp. *true*) or remain unchanged:

$$\mathsf{Pr}(\alpha, \varphi) = \begin{cases} \mathit{false} & \text{if} \quad [\alpha \models_F \varphi] = \bot \\ \Box\phi & \text{if} \quad \text{otherwise} \end{cases}$$

$$\mathsf{Pr}(\alpha, \varphi) = \begin{cases} \mathit{true} & \text{if} \quad [\alpha \models_F \varphi] = \top \\ \Diamond\phi & \text{if} \quad \text{otherwise} \end{cases}$$

Note that the semantics of FLTL is not frequently used, due to $\mathtt{LTL_3}$ being generally more expressive, as shown in [BLS10b]. However, the expressiveness of $\mathtt{LTL_3}$ would actually be an issue if it were used to construct the progression rules. To be more precise, the '?' (*unknown*) verdict in $\mathtt{LTL_3}$ semantics would raise additional and unnecessary complications in the progression rules, as this verdict does not provide any additional information as far as our progression-based approach is concerned. Therefore, we use FLTL for specifying the progression rules without any loss of generality as shown later in the proof of Lemma 1.

Until operator. Let $\varphi = \varphi_1 \; \mathcal{U} \; \varphi_2$. Recall that $\varphi_1 \; \mathcal{U} \; \varphi_2 = \varphi_2 \vee (\varphi_1 \wedge \bigcirc(\varphi_1 \; \mathcal{U} \; \varphi_2))$. We divide the $\mathcal{U}$ formula into two parts, one with globally ($\Box\varphi_1$) and the other eventuality ($\Diamond\varphi_2$). These sub-formulas are evaluated separately and the verdict of each of them is used

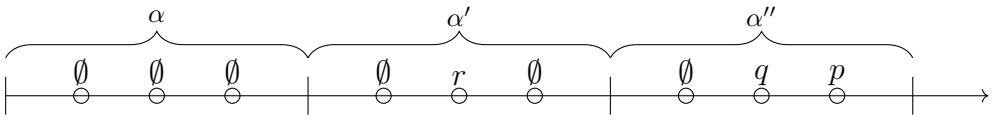


Figure 3.5: Progression example.

to define the progression for the $\mathcal{U}$ operator. However, for the case when both φ_1 and φ_2 occur in the same computation, we cannot come to a verdict without considering the order of satisfaction of these sub-formulas. That is, on a given finite trace α, if φ_2 holds in $\alpha(i)$ (denoted $\diamondsuit_i\, \varphi_2$) and φ_1 holds throughout in all states from $\alpha(0)$ to $\alpha(i-1)$ (denoted $\square_{i-1}\, \varphi_1$), then the progression of φ becomes *true*. If this is not the case, and $\square\varphi_1$ does not hold in α, the progression of φ becomes *false*, since this signifies a break from the streak of φ_1 required for φ to hold. If it is neither of the above two cases, and the evaluated verdict of $\diamondsuit \mathsf{Pr}(\alpha, \varphi_2)$ is $\top$, then this represents a case where we do not have enough information about φ_1 to evaluate $\varphi_1\, \mathcal{U}\, \varphi_2$. Thus, making the progression solely dependant on φ_1. The progression of φ remains unchanged if φ_1 holds throughout α, but φ_2 does not hold anywhere:

$$\mathsf{Pr}(\alpha, \varphi) = \begin{cases} \textit{true} & \text{if } \exists i \in [0, |\alpha| - 1]\ .\ [\alpha \models_F \diamondsuit_i\, \mathsf{Pr}(\alpha, \varphi_2)] = \top \\ & \quad \wedge\ [\alpha \models_F \square_{i-1}\, \mathsf{Pr}(\alpha, \varphi_1)] = \top \\ \textit{false} & \text{if } [\alpha \models_F \square\, \mathsf{Pr}(\alpha, \varphi_1)] = \bot\ \wedge \text{ not the first case} \\ \mathsf{Pr}(\alpha, \varphi_1) & \text{if } [\alpha \models_F \diamondsuit\, \mathsf{Pr}(\alpha, \varphi_2)] = \top\ \wedge \text{ not the second case} \\ \mathsf{Pr}(\alpha, \varphi_1)\ \mathcal{U}\ \mathsf{Pr}(\alpha, \varphi_2) & \text{if } [\alpha \models_F \square\, \mathsf{Pr}(\alpha, \varphi_1)] = \top\ \wedge\ [\alpha \models_F \diamondsuit\, \mathsf{Pr}(\alpha, \varphi_2)] = \bot \end{cases}$$

Example. Consider the formula, $\varphi = \diamondsuit r \rightarrow (\neg p\ \mathcal{U}\ q)$ with sub-formulas $\varphi_s = \{\diamondsuit r,\ q,\ \diamondsuit q,\ \square p\}$, according to our progression rules. Consider the trace in Fig. 3.5 divided into three segments. In the first segment α, neither p, q nor r are present, and as far as the laws of the progression function defined above, φ remains unchanged for the next segment; i.e., $\mathsf{Pr}(\alpha, \varphi) = \varphi$. In the second segment α', proposition r is observed, this satisfies sub-formula $\diamondsuit r$ the progressed formula becomes $\neg p\ \mathcal{U}\ q$; i.e., $\mathsf{Pr}(\alpha', \varphi) = \neg p\ \mathcal{U}\ q$. In the

next segment α'', proposition q occurs before p. This falls under the first case of the until progression operator. Since q happens after a streak of $\neg p$, we arrive at the verdict *true*; i.e., $\mathsf{Pr}(\alpha'', \neg p \,\mathcal{U}\, q) = \mathit{true}$. Put it another way, $\mathsf{Pr}(\alpha\alpha'\alpha'', \varphi) = \mathit{true}$.

Lemma 1. *Given an* `LTL` *formula* φ*, and a two finite traces* $\alpha, \sigma \in \Sigma^*$*, trace* $\alpha\sigma$ *satisfies* φ *if and only if* σ *satisfies* $\mathsf{Pr}(\alpha, \varphi)$*. Formally,*

$$[\alpha\sigma \models_F \varphi] \iff [\sigma \models_F \mathsf{Pr}(\alpha, \varphi)]$$

Proof. We distinguish the following cases:

Case 1: First, we consider the base case of this proof, where the formula is an atomic proposition, that is, $\varphi = \mathsf{p}$.

($\Rightarrow$) Let us first consider that p is observed on the first state of $\alpha\sigma$. This implies, $[\alpha\sigma \models_F \varphi]$ yields *true*, and $\mathsf{Pr}(\alpha, \varphi)$ yields $\top$. Therefore, $[\sigma \models_F \mathsf{Pr}(\alpha, \varphi)]$ must also yield *true*.

Now, let us consider that p is not observed on the first state of $\alpha\sigma$. This implies, $[\alpha\sigma \models_F \varphi]$ yields *false*, and $\mathsf{Pr}(\alpha, \varphi)$ yields $\bot$. Therefore, $[\sigma \models_F \mathsf{Pr}(\alpha, \varphi)]$ must also yield *false*.

($\Leftarrow$) Let us first consider that $[\sigma \models_F \mathsf{Pr}(\alpha, \varphi)]$ yields *true*. This implies, $\mathsf{Pr}(\alpha, \varphi)$ yields $\top$, and $[\alpha\sigma \models_F \varphi]$ yields *true*. Therefore, p must have been observed on the first state of $\alpha\sigma$.

Now, let us consider that $[\sigma \models_F \mathsf{Pr}(\alpha, \varphi)]$ yields *false*. This implies, $\mathsf{Pr}(\alpha, \varphi)$ yields $\bot$, and $[\alpha\sigma \models_F \varphi]$ yields *false*. Therefore, p must not have been observed on the first state of $\alpha\sigma$.

Case 2: Assume that the proof has been established for the case when the formula is $\varphi = \phi$. Now, we consider the case where the formula is $\varphi = \neg\phi$.

We can say $[\alpha\sigma \models_F \neg\phi]$ is equivalent to $\neg[\alpha\sigma \models_F \phi]$ according to the finite-trace semantics of `LTL`. We can also say $[\sigma \models_F \mathsf{Pr}(\alpha, \neg\phi)]$ is equivalent to $[\sigma \models_F \neg\mathsf{Pr}(\alpha, \phi)]$ since $\mathsf{Pr}(\alpha, \neg\phi) = \neg\mathsf{Pr}(\alpha, \phi)$ is defined as a progression rule. Furthermore, $[\sigma \models_F \neg\mathsf{Pr}(\alpha, \phi)]$ is equivalent to $\neg[\sigma \models_F \mathsf{Pr}(\alpha, \phi)]$ according to the finite-trace semantics of `LTL`.

Based on our assumption, the proof has already been established for $[\alpha\sigma \models_F \phi] \iff [\sigma \models_F \mathsf{Pr}(\alpha, \phi)]$. Therefore, $\neg[\alpha\sigma \models_F \phi] \iff \neg[\sigma \models_F \mathsf{Pr}(\alpha, \phi)]$, and by extension,

$[\alpha\sigma \models_F \neg\phi] \iff [\sigma \models_F \mathsf{Pr}(\alpha, \neg\phi)]$

Case 3: Assume that the proof has been established for the case when the formula is $\varphi = \phi$. Now, we consider the case where the formula is $\varphi = \bigcirc\phi$.

Let us first consider the case where the length of the trace α is 1, that is, $\mid \alpha \mid = 1$ and $\mid \alpha^1 \mid = 0$. In this particular case, $[\alpha\sigma \models_F \bigcirc\phi]$ is equivalent to $[\sigma \models_F \phi]$. Furthermore, $\mathsf{Pr}(\alpha, \bigcirc\phi) = \phi$; which implies, $[\sigma \models_F \mathsf{Pr}(\alpha, \bigcirc\phi)]$ is equivalent to $[\sigma \models_F \phi]$. Therefore, $[\alpha\sigma \models_F \bigcirc\phi] \iff [\sigma \models_F \mathsf{Pr}(\alpha, \bigcirc\phi)]$.

Now, let us consider the case where the length of the trace α is longer than 1, that is, $\mid \alpha \mid \geq 1$ and $\mid \alpha^1 \mid \geq 1$. In this case, $[\alpha\sigma \models_F \bigcirc\phi]$ is equivalent to $[\alpha^1\sigma \models_F \phi]$, and $[\sigma \models_F \mathsf{Pr}(\alpha, \bigcirc\phi)]$ is equivalent to $[\sigma \models_F \mathsf{Pr}(\alpha^1, \phi)]$.

Based on our assumption, the proof has already been established for $[\alpha^1\sigma \models_F \phi] \iff [\sigma \models_F \mathsf{Pr}(\alpha^1, \phi)]$. Therefore, $[\alpha\sigma \models_F \bigcirc\phi] \iff [\sigma \models_F \mathsf{Pr}(\alpha, \bigcirc\phi)]$.

Case 4: Assume that the proof has been established for the cases when the formulas are $\varphi = \varphi_1$ and $\varphi = \varphi_2$. Now, we consider the case where the formula is $\varphi = \varphi_1 \vee \varphi_2$.

Based on our assumption, the proof has already been established for $[\alpha\sigma \models_F \phi_1] \iff [\sigma \models_F \mathsf{Pr}(\alpha, \phi_1)]$ and $[\alpha\sigma \models_F \phi_2] \iff [\sigma \models_F \mathsf{Pr}(\alpha, \phi_2)]$. Therefore, we can derive the following:

$$\begin{aligned}
[\alpha\sigma \models_F (\varphi_1 \vee \varphi_2)] &\iff [\alpha\sigma \models_F \varphi_1 \vee \alpha\sigma \models_F \varphi_2] \\
&\iff [\sigma \models_F \mathsf{Pr}(\alpha, \varphi_1) \vee \sigma \models_F \mathsf{Pr}(\alpha, \varphi_2)] \\
&\iff [\sigma \models_F \varphi_1' \vee \varphi_2'] \\
&\iff [\sigma \models_F \mathsf{Pr}(\varphi_1 \vee \varphi_2)].
\end{aligned}$$

Case 5: Assume that the proof has been established for the cases when the formulas are $\varphi = \varphi_1$ and $\varphi = \varphi_2$. Now, we consider the case where the formula is $\varphi = \varphi_1 \,\mathcal{U}\, \varphi_2$.

First, we prove the equivalence between $\varphi = \varphi_1 \,\mathcal{U}\, \varphi_2$ and its corresponding SMT formula. That is,

$$[(\sigma, i) \models_F \varphi_1 \,\mathcal{U}\, \varphi_2] \iff [\exists k \geq i \;.\; \Diamond_k \varphi_2 \wedge \Box_{k-1} \varphi_1]$$

To this end, we have,

$$\begin{aligned}
&[(\sigma, i) \models_F \varphi_1 \,\mathcal{U}\, \varphi_2] \\
&\iff [(\sigma, i) \models_F \Big(\varphi_2 \vee \big(\varphi_1 \wedge \bigcirc(\varphi_1 \,\mathcal{U}\, \varphi_2)\big)\Big)] \\
&\iff [(\sigma, i) \models_F \varphi_2] \vee [(\sigma, i) \models_F \Big(\varphi_1 \wedge \bigcirc(\varphi_1 \,\mathcal{U}\, \varphi_2)\Big)] \\
&\iff [(\sigma, i) \models_F \varphi_2] \vee \Big([(\sigma, i) \models_F \varphi_1] \wedge [(\sigma, i+1) \models_F \varphi_1 \,\mathcal{U}\, \varphi_2\Big)] \\
&\iff [(\sigma, i) \models_F \varphi_2] \vee \Bigg([(\sigma, i) \models_F \varphi_1] \wedge [(\sigma, i+1) \models_F \Big(\varphi_2 \vee \big(\varphi_1 \wedge \bigcirc(\varphi_1 \,\mathcal{U}\, \varphi_2)\big)\Big)]\Bigg) \\
&\iff [(\sigma, i) \models_F \varphi_2] \vee \Big([(\sigma, i) \models_F \varphi_1] \wedge [(\sigma, i+1) \models_F \varphi_2]\Big) \vee \ldots \vee [(\sigma, i+k) \models_F \varphi_1 \,\mathcal{U}\, \varphi_2]
\end{aligned}$$

for some $k \geq 1$. Now, in order for $[(\sigma, i) \models_F \varphi_1 \,\mathcal{U}\, \varphi_2]$ to yield *true*, there must be a $k \geq 1$ such that $[(\sigma, i) \models_F \varphi_1 \wedge \ldots \wedge (\sigma, i+k-1) \models_F \varphi_1 \wedge (\sigma, i) \models_F \varphi_2]$, that is

$$\begin{aligned}
[(\sigma, i) \models_F \varphi_1 \,\mathcal{U}\, \varphi_2] \iff & [\exists k \geq 1 \;.\; (\sigma, i) \models_F \varphi_1 \wedge \ldots \wedge (\sigma, i+k-1) \models_F \varphi_1 \wedge \\
& (\sigma, k) \models_F \varphi_2] \\
\iff & [\exists k \geq 1 \;.\; (\sigma, i) \models_F \Diamond_k \varphi_2 \wedge (\sigma, i) \models_F \Box_{k-1} \varphi_1]
\end{aligned}$$

Now, we prove this case for the lemma as follows,

($\Rightarrow$) Let us assume $[\alpha\sigma \models_F \varphi_1 \,\mathcal{U}\, \varphi_2] = \top$ and $[\sigma \models_F \mathsf{Pr}(\alpha, \varphi_1 \,\mathcal{U}\, \varphi_2)] = \bot$. If $[\sigma \models_F \mathsf{Pr}(\alpha, \varphi_1 \,\mathcal{U}\, \varphi_2)] = \bot$, then that implies either $[\alpha \models_F \Box\varphi_1] = \bot$, or $[\alpha \models_F \Box\varphi_1] = \top \wedge [\alpha \models_F \Diamond\varphi_2] = \bot \wedge [\sigma \models_F \mathsf{Pr}(\alpha, \varphi_1) \,\mathcal{U}\, \mathsf{Pr}(\alpha, \varphi_2)] = \bot$. However, neither of these two cases are possible since in order for $[\alpha\sigma \models_F \varphi_1 \,\mathcal{U}\, \varphi_2] = \top$ to hold, φ_1 must hold until φ_2 in $\alpha\sigma$. Therefore, $[\sigma \models_F \mathsf{Pr}(\alpha, \varphi_1 \,\mathcal{U}\, \varphi_2)] = \top$.

($\Leftarrow$) Let us assume $[\alpha\sigma \models_F \varphi_1 \,\mathcal{U}\, \varphi_2] = \bot$ and $[\sigma \models_F \mathsf{Pr}(\alpha, \varphi_1 \,\mathcal{U}\, \varphi_2)] = \top$. If $[\sigma \models_F \mathsf{Pr}(\alpha, \varphi_1 \,\mathcal{U}\, \varphi_2)] = \top$, then that implies either $[\alpha \models_F \varphi_1 \,\mathcal{U}\, \varphi_2] = \top$ or $[\alpha \models_F \Box\varphi_1] = \top \wedge [\sigma \models_F \mathsf{Pr}(\alpha, \varphi_1) \,\mathcal{U}\, \mathsf{Pr}(\alpha, \varphi_2)] = \top$. However, neither of these two cases are possible since in order for $[\alpha\sigma \models_F \varphi_1 \,\mathcal{U}\, \varphi_2] = \bot$ to hold, either φ_1 must be violated on some state before φ_2 is observed, or φ_2 is never observed. Therefore,

$[\sigma \models_F \mathsf{Pr}(\alpha, \varphi_1 \mathcal{U} \varphi_2)] = \bot$.

□

3.3 SMT-based Solution

In this section, we elaborate on our solution for distributed monitoring using the two monitoring techniques mentioned before: (1) automata-based approach, and (2) progression-based approach.

3.3.1 Overall Idea

Automata-based approach. Recall from Section 3.1 (Fig. 3.4) that monitoring a distributed computation may result in multiple verdicts depending upon different ordering of events. In other words, given a distributed computation $(\mathcal{E}, \rightsquigarrow)$ and an LTL formula φ, different ordering of events may reach different states in the monitor automaton $\mathcal{M}_\varphi = (\Sigma, Q, q_0, \delta, \lambda)$ (as defined in Definition 3). In order to ensure that all possible verdicts are explored, we generate an SMT instance for (1) the distributed computation $(\mathcal{E}, \rightsquigarrow)$, and (2) each possible path in the $\mathtt{LTL_3}$ monitor. Thus, the corresponding decision problem is the following: given $(\mathcal{E}, \rightsquigarrow)$ and a monitor path $q_0 q_1 \cdots q_m$ in an $\mathtt{LTL_3}$ monitor, can $(\mathcal{E}, \rightsquigarrow)$ reach q_m? If the SMT instance is satisfiable, then $\lambda(q_m)$ is a possible verdict. For example, for the monitor in Fig. 2.1, we consider two paths $q_0^* q_\bot$ and $q_0^* q_\top$ (and, hence, two SMT instances). Thus, if both instances turn out to be unsatisfiable, then the resulting monitor state is q_0, where $\lambda(q_0) =?$.

We note that $\mathtt{LTL_3}$ monitors may contain non-self-loop cycles. In order to simplify the SMT instance creation process (for each possible path in the $\mathtt{LTL_3}$ monitor), we collapse each non-self-loop cycle into one state with a self-loop labeled by the sequence of events in the cycle using Algorithm 1. As an example, in Fig. 3.6, Algorithm 1 first takes an $\mathtt{LTL_3}$ monitor (Fig. 3.6a) and adds the necessary self-loops (Fig. 3.6b). Then it eliminates all

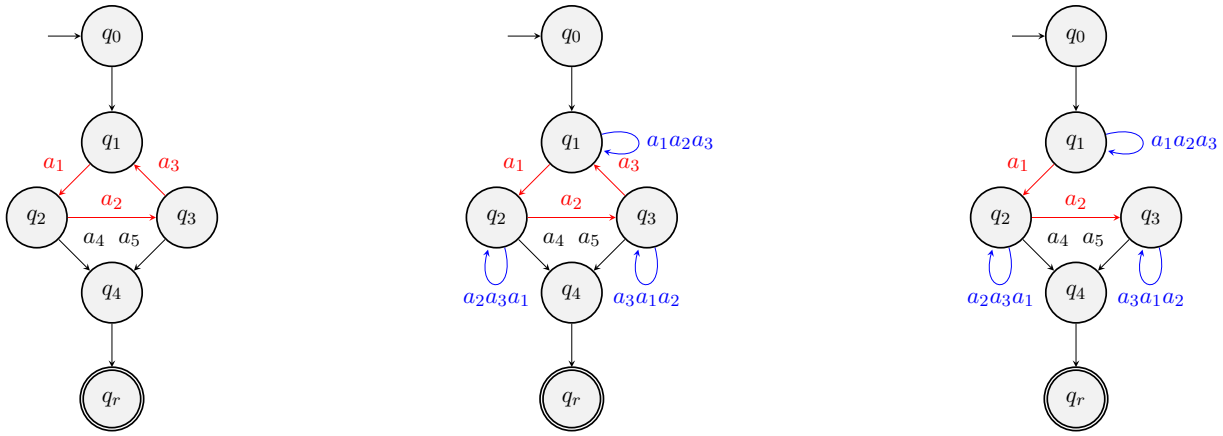


Figure 3.6: Removing non-loop cycles in an $\mathtt{LTL}_3$ Monitor.

non-self-loop cycles by removing transitions from states with higher identifiers to states with lower identifiers in cycles (Fig. 3.6c). The non-deterministic nature of the final automata ensure that all the transitions and the accepting language of the automata are preserved.

Algorithm 1: Non-Self Loop Cycle Removal Algorithm.

Input $\mathcal{M}_\varphi = (\Sigma, Q, q_0, \delta, \lambda)$
Output $\mathcal{M}'_\varphi = (\Sigma, Q, q_0, \delta', \lambda)$
Let CP be the set of all possible paths containing cycles
$\delta' \leftarrow \delta$
for each $q \in Q$ **do**
  for each $q \xrightarrow{s_m} \cdots \xrightarrow{s_n} q \in \mathsf{CP}$ **do**
    $\delta'(q, s_m \cdots s_n) \leftarrow q$
  end for
end for
for each $q_m \xrightarrow{s} q_n \in \{q_i \xrightarrow{s_k} q_j \mid q \xrightarrow{s_m} \cdots q_i \xrightarrow{s_k} q_j \cdots \xrightarrow{s_n} q \in \mathsf{CP}\}$ **do**
  if $m > n$ **then**
    $\delta'(q_m, s) \leftarrow \emptyset$
  end if
end for
return $\mathcal{M}_\varphi$

Lemma 2. *Let $\mathcal{M}_\varphi = (\Sigma, Q, q_0, \delta, \lambda)$ be the monitor automaton for LTL formula, φ, and $\mathcal{M}'_\varphi = (\Sigma, Q, q_0, \delta', \lambda)$ be the monitor automaton with no non-self loop cycles, obtained from applying Algorithm 1 on $\mathcal{M}_\varphi$. Given a finite trace, $\alpha = a_1 a_2 \cdots a_n$ and a initial state, $q \in Q$, we prove that $\lambda(\delta(q, \alpha)) = \lambda(\delta'(q, \alpha))$.*

Proof. We distinguish the following cases:

Case 1 ($\Rightarrow$):

First we show, $\lambda(\delta(q,\alpha)) \rightarrow \lambda(\delta'(q,\alpha))$, that is, $\forall\alpha, \forall q \in Q \;.\; \lambda(\delta(q,\alpha)) \implies \lambda(\delta'(q,\alpha))$ Let $\alpha = a_1a_2\cdots a_n$, where $\forall i \in [1,n].a_i \in \Sigma$. Algorithm 1 removes non-self loop cycles by removing a transition such that the corresponding transition of $\delta(q,a_i)$, $\delta'(q,a_i)$, where $i \in [1,m]$ does not exist. This is such that $\exists k \in [1,i] \;.\; q' \xrightarrow{a_{i-k}} \cdots q \xrightarrow{a_i} q'$. This transition is same as $\delta'(q', a_{i-k}\cdots a_i) = q'$ which was one of the added self-loops. The rest of the transitions are maintained such that $\delta(q,a_i) = \delta'(q,a_i)$, where $q \in Q$ and $i \in [1,m]$.

Case 2 ($\Leftarrow$):

Now, we show, $\lambda(\delta'(q,\alpha)) \rightarrow \lambda(\delta(q,\alpha))$, that is, $\forall\alpha, \forall q \in Q \;.\; \lambda(\delta'(q,\alpha)) \implies \lambda(\delta(q,\alpha))$ Let $\alpha = a_1a_1\cdots a_n$, where $\forall i \in [1,n].a_i \in \Sigma$. A self-loop in $\mathcal{M}'_\varphi$ can be represented by $\exists i \in [1,n], \exists k \in [1,n-i] \;.\; \delta'(q, a_ia_{i+1}\cdots a_{i+k}) = q$. In another words, there exists a path $q \xrightarrow{a_i} q' \xrightarrow{a_{i+1}} \cdots \xrightarrow{a_{i+k}} q$ in $\mathcal{M}_\varphi$. The rest of the non-self loop transitions are the same, such that $\delta'(q,a_i) = \delta(q,a_i)$, where $q \in Q$ and $i \in [1,m]$.

Thus, $\lambda(\delta(q,\alpha)) = \lambda(\delta'(q,\alpha))$ □

Progression-based approach. In a synchronous system, verification on a computation can be performed in a state by state approach due to the existence of a total ordering of events [BF16a]. However, in a *partially* synchronous system, no such ordering of events is possible. A distributed computation $(\mathcal{E}, \rightsquigarrow)$ may have different ordering of events dictated by different interleavings of events. Therefore, it is possible to obtain multiple verdicts on the same distributed computation $(\mathcal{E}, \rightsquigarrow)$. In order to explore these verdicts, we propose a monitoring approach based on *formula progression* that, if possible, *partially* evaluates a formula on the current computation, and based on the verdict, provides a *rewritten* formula that is to be evaluated on the extensions of the computation. As an example, let us consider the formula to be monitored as, $\varphi = \Diamond(a \rightarrow \Diamond b)$. Now, if in some trace in a computation, the monitor observes a, then for the extensions of computations, it is enough to monitor the rewritten formula, $\varphi' = \Diamond b$, as the final verdict is no longer dependent on the occurrence of

a. We call this method of rewriting formula **Progression**, which we discuss in length later on. In the next two subsections, we present the SMT entities and constraints with respect to *one* monitor path and a distributed computation.

3.3.2 SMT Entities

SMT entities represent the sub-formulas of an LTL formula and a distributed computation. After the verdicts from all the sub-formulas are generated, we construct our rewritten formula by attaching the said verdicts to their corresponding parent formulas in the parse tree and then performing an in-order traversal starting from the root of the parse tree. At the end of the traversal, the resulting formula is, in fact, the progression for the next computation. We now introduce the entities that represent a path in an $\mathtt{LTL_3}$ monitor $\mathcal{M}_\varphi = (\Sigma, Q, q_0, \delta, \lambda)$ for LTL formula φ and distributed computation $(\mathcal{E}, \rightsquigarrow)$. It should be noted that the SMT entities in this subsection are used in both the automata-based and the progression-based approaches.

Monitor automaton. Let $q_0 \xrightarrow{s_0} q_1 \xrightarrow{s_1} \cdots (q_j \xrightarrow{s_j} q_j)^* \cdots \xrightarrow{s_{m-1}} q_m$ be a path of monitor $\mathcal{M}_\varphi$, which may or may not include a self-loop. We include a non-negative integer variable k_i for each transition $q_i \xrightarrow{s_i} q_{i+1}$, where $i \in [0, m-1]$ and $s_i \in \Sigma$. This is also true for the self-loop $q_j \xrightarrow{s_j} q_j$, for which we include a non-negative interger k_j.

Distributed computation. In our SMT encoding, the set of events, $\mathcal{E}$ are represented by a bit vector, where each bit corresponds to an individual event in the distributed computation, $(\mathcal{E}, \rightsquigarrow)$. We conduct a *pre-processing* of the distributed computation, during which we create an $\mathcal{E} \times \mathcal{E}$ matrix, *hbSet* to incorporate the additional happen-before relations obtained by the clock-synchronization algorithm. Afterwards, we populate the *hbSet* with 0's and 1's, such that $hbSet[i][j] = 1$ if $\mathcal{E}[i] \rightsquigarrow \mathcal{E}[j]$, and $hbSet[i][j] = 0$ otherwise. We introduce a function $\mu : \mathcal{E} \times \mathsf{AP} \rightarrow \{true, false\}$ in order to establish a relation between each event and the atomic propositions in it. In the event that other variables or constants are used in defining the predicates (e.g. $x_1 + x_2 \geq 2$), μ is constructed accordingly. Finally, we introduce an

uninterpreted function $\rho : \mathbb{Z}_{\geq 0} \rightarrow 2^{\mathcal{E}}$ that identifies a sequence of consistent cuts from $\{\}$ to $\{\mathcal{E}\}$ for reaching a verdict, while satisfying a number of given constraints explained in 3.3.3.

3.3.3 SMT Constraints

Once we define the necessary SMT entities, we move onto the SMT constraints. We first define the common SMT constraints for consistent cuts that are enforced on both the automata-based and the progression-based approaches. Afterwards we define the SMT constraints that are more dependant on the methodology.

Consistent cut constraints over ρ. In order to ensure that the uninterpreted function ρ identifies a sequence of consistent cuts, we enforce certain consistent cut constraints. The first constraint enforces that each element in the range of ρ is in fact a consistent cut:

$$\forall i \in [0, m]. \forall e, e' \in \mathcal{E}. \Big((e' \rightsquigarrow e) \wedge (e \in \rho(i)) \Big) \rightarrow \Big(e' \in \rho(i) \Big)$$

Next, we enforce that the sequence of consistent cuts identified by ρ start from an empty set of events, and each successor cut of the sequence contains one more new event than its predecessor.

$$\forall i \in [0, m].\ |\rho(i+1)| \ = \ |\rho(i)| + 1$$

Finally, we ensure that each successive consistent cut is immediately reachable in $(\mathcal{E}, \rightsquigarrow)$ by enforcing a subset relation:

$$\forall i \in [0, m].\ \rho(i) \subseteq \rho(i+1)$$

Once a sequence of consistent cuts have been generated, we check if the sequence satisfies the specification. This is done using (1) progression-based approach, where the `LTL` formula is represented by a SMT constrain and (2) $\texttt{LTL}_3$ automata-based approach, where a path on the automata is represented as an SMT constraint. This is repeated for all sub-formulas of the original `LTL` formula and all paths in the $\texttt{LTL}_3$ automata respectively as discussed below.

Constraints for $\texttt{LTL}_3$ automata over ρ. These constraints are responsible for generating

a valid sequence of consistent cuts given a distributed computation $(\mathcal{E}, \rightsquigarrow)$ that runs on monitor path $q_1 \xrightarrow{s_1} q_2 \cdots q_j^* \cdots \xrightarrow{s_{m-1}} q_m$. We begin with interpreting $\rho(k_m)$ by requiring that running $(\mathcal{E}, \rightsquigarrow)$ ends in monitor state q_m. The corresponding SMT constraint is:

$$\mu(\mathsf{front}(\rho(k_m)), s_{m-1})$$

For every monitor state q_i, where $i \in [0, m-1]$, if q_i does not have a self-loop, the corresponding SMT constraint is:

$$\mu(\mathsf{front}(\rho(k_{i+1}-1)), s_i) \ \wedge \ (k_i = k_{i+1} - 1)$$

For every monitor state q_j, where $j \in [0, m-1]$, suppose q_j has a self-loop (recall that a cycle of r transitions in the monitor automaton is collapsed into a self-loop labeled by a sequence of r letters). Let us imagine that this self-loop executed z number of times for some $z \geq 0$. Furthermore, we denote the sequence of letters in the self-loop as $s_{j_1} s_{j_2} \cdots s_{j_r}$. The corresponding SMT constraint is:

$$\bigwedge_{i=1}^{z} \bigwedge_{n=1}^{r} \mu\Big(\mathsf{front}\big(\rho(k_j + r(i-1) + n)\big), s_{j_n}\Big)$$

Again, since z is a free variable in the above constraint, the solver will identify some value $z \geq 0$ which is exactly what we need. To ensure that the domain of ρ starts from the empty consistent cut (i.e., $\rho(0) = \emptyset$), we add:

$$k_0 = 0.$$

Finally, let C denote the conjunction of all the above constraints. Recall that this conjunction is with respect to only one monitor path from q_0 to q_m. Since there may be multiple paths in the monitor automaton that can reach q_m from q_0, we replicate the above constraints for each such path. Suppose there are n such paths and let $C_1, C_2, \ldots, C_n$ be the corresponding

SMT constraints for these n paths. We include the following constraint:

$$C_1 \vee C_2 \vee C_3 \vee \cdots \vee C_n$$

This means that if the SMT instance is satisfiable, then computation $(\mathcal{E}, \rightsquigarrow)$ can reach monitor state q_m from q_0.

Constraints for `LTL` progression over ρ. Given a distributed system $(\mathcal{E}, \rightsquigarrow)$, the aforementioned constraints may generate a valid sequence of consistent cuts that may yield different verdicts based on the ordering of the concurrent events. Therefore, in order to avoid false positives, all possible outcomes are explored when evaluating an `LTL` formula φ on $(\mathcal{E}, \rightsquigarrow)$. We achieve this by checking for both satisfaction and violation in the sequence of consistent cuts $C_0C_1C_2 \cdots C_m$ interpreted by the uninterpreted function $\rho(m)$. Note that monitoring any `LTL` formula using our progression rules will result in monitoring sub-formulas with only atomic propositions, globally and eventually temporal operators:

$$\begin{aligned}
\varphi = p \qquad & \mathsf{front}(\rho_i) \models p, \text{ for } p \in \mathsf{AP} \text{ (satisfaction, i.e.,} \top) \\
\varphi = \Box\phi \qquad & \exists i \in [0, m].\ \mathsf{front}(\rho_i) \not\models \phi \text{ (violation, i.e.,} \bot) \\
\varphi = \Diamond\phi \qquad & \exists i \in [0, m].\ \mathsf{front}(\rho_i) \models \phi \text{ (satisfaction, i.e.,} \top)
\end{aligned}$$

Opposite cases result in a rewritten formula that will progress to the next segment. In general, the verdict for any `LTL` formula will be derived using our progression rules in Section 3.2.

3.4 Optimization

3.4.1 Segmentation of Distributed Computation

RV is known to be an NP-complete problem in the number of processes in a distributed setting [Gar02]. The complexity exhibits even more exponential blowup during verifying formulas with nested temporal operators. In order to cope with this complexity, we divide

our computation into smaller *segments*, $(seg_1, \rightsquigarrow)(seg_2, \rightsquigarrow)\cdots(seg_{l/g}, \rightsquigarrow)$ to create smaller, albeit more SMT problems. Given a distributed computation $(\mathcal{E}, \rightsquigarrow)$ of length l, we divide it into $\frac{l}{g}$ smaller segments length g. The set of events in segment j, where $j \in [1, \frac{l}{g}]$, is the following:

$$seg_j = \left\{ e^n_{\tau,\sigma,\omega} | \sigma \in [\max\{0, (j-1) \times g - \epsilon\}, j \times g] \wedge n \in [1, |\mathcal{P}|] \right\}$$

Note that each segment (barring seg_0) has to be constructed starting at ϵ time units before the previous segments ending point. This creates an overlap of ϵ time units between each pair of adjacent segments. Doing so ensures that no pair of possible concurrent become non-concurrent due to the splits caused by segmentation. Therefore, dividing the actual computation into segments does not have any effect on the final verdict of the said computation. We also use parallelization to make our algorithm perform faster, while utilizing most of the computation power modern processors are capable of handling.

Lemma 3. *A distributed computation, $(\mathcal{E}, \rightsquigarrow)$, of length l satisfies an* LTL *formula, φ, if and only if the distributed computation, $(\mathcal{E}, \rightsquigarrow)$, is divided into $\frac{l}{g}$ segments of length g satisfies φ using the automata-based approach. That is,*

$$[(\mathcal{E}, \rightsquigarrow) \models_3 \varphi] \iff [(seg_1.seg_2.\cdots.seg_{\frac{l}{g}}, \rightsquigarrow) \models_3 \varphi]$$

Proof. Let us assume $[(\mathcal{E}, \rightsquigarrow) \models_3 \varphi] \neq [(seg_1.seg_2.\cdots.seg_{\frac{l}{g}}, \rightsquigarrow) \models_3 \varphi]$, that is, $\{\alpha \models_3 \varphi \mid \alpha \in \mathsf{Tr}(\mathcal{E}, \rightsquigarrow)\} \neq \{\alpha \models_3 \varphi \mid \alpha \in \mathsf{Tr}(seg_1.seg_2.\cdots.seg_{\frac{l}{g}}, \rightsquigarrow)\}$ (Recall Section 3.1.1).

($\Rightarrow$) Let C_k be a consistent cut such that C_k is in $\mathsf{Tr}(\mathcal{E}, \rightsquigarrow)$, but not in $\mathsf{Tr}(seg_1.seg_2.\cdots.seg_{\frac{l}{g}}, \rightsquigarrow)$ for some $k \in [0, |\mathcal{E}|]$. This implies that the frontier of C_k, $\mathsf{front}(C_k) \not\subseteq seg_1$ and $\mathsf{front}(C_k) \not\subseteq seg_2$ and $\cdots$ and $\mathsf{front}(C_k) \not\subseteq seg_{\frac{l}{g}}$. However, this is not possible, as according to the segmentation construction, there must be a seg_j where $1 \leq j \leq \frac{l}{g}$ such that $\mathsf{front}(C_k) \subseteq seg_j$. Therefore, such C_k cannot exist, and $\{\alpha \models_3 \varphi \mid \alpha \in \mathsf{Tr}(\mathcal{E}, \rightsquigarrow)\} \subseteq \{\alpha \models_3 \varphi \mid \alpha \in \mathsf{Tr}(seg_1.seg_2.\cdots.seg_{\frac{l}{g}}, \rightsquigarrow)\}$. By extension, $[(\mathcal{E}, \rightsquigarrow) \models_3 \varphi] \Rightarrow [(seg_1.seg_2.\cdots.seg_{\frac{l}{g}}, \rightsquigarrow) \models_3 \varphi]$

($\Leftarrow$) Let C_k be a consistent cut such that C_k is in $\mathsf{Tr}(seg_1.seg_2.\cdots.seg_{\frac{l}{g}}, \rightsquigarrow)$,

but not in $\mathsf{Tr}(\mathcal{E}, \rightsquigarrow)$ for some $k \in [0, |\mathcal{E}|]$. This implies, $\mathsf{front}(C_k) \subseteq seg_j$ and $\mathsf{front}(C_k) \not\subseteq \mathcal{E}$ for some $j \in [1, \frac{l}{g}]$. However, this is not possible due to the fact that $\forall j \in [1, \frac{l}{g}]$. $seg_j \subseteq \mathcal{E}$. Therefore, such C_k cannot exist, and $\{\alpha \models_3 \varphi \mid \alpha \in \mathsf{Tr}(seg_1.seg_2.\cdots.seg_{\frac{l}{g}}, \rightsquigarrow)\} \subseteq \{\alpha \models_3 \varphi \mid \alpha \in \mathsf{Tr}(\mathcal{E}, \rightsquigarrow)\}$. By extension, $[(seg_1.seg_2.\cdots.seg_{\frac{l}{g}}, \rightsquigarrow) \models_3 \varphi] \Rightarrow [(\mathcal{E}, \rightsquigarrow) \models_3 \varphi]$

Therefore, $[(\mathcal{E}, \rightsquigarrow) \models_3 \varphi] \iff [(seg_1.seg_2.\cdots.seg_{\frac{l}{g}}, \rightsquigarrow) \models_3 \varphi]$.

□

Lemma 4. *A distributed computation $(\mathcal{E}, \rightsquigarrow)$ of length l satisfies an LTL formula φ if and only if the distributed computation, $(\mathcal{E}, \rightsquigarrow)$, is divided into $\frac{l}{g}$ segments of length g satisfies φ using the progression-based approach. That is,*

$$[(\mathcal{E}, \rightsquigarrow) \models_F \varphi] \iff [(seg_1.seg_2.\cdots.seg_{\frac{l}{g}}, \rightsquigarrow) \models_F \varphi]$$

Proof. Using Lemma 1 and Lemma 3, we can trivially prove, $[(\mathcal{E}, \rightsquigarrow) \models_F \varphi] \iff [(seg_1.seg_2.\cdots.seg_{\frac{l}{g}}, \rightsquigarrow) \models_F \varphi]$. □

3.4.2 Parallelized Monitoring

Many cloud services use clusters of computers equipped with multiple processors and computing cores. This allows them to deal with high data rates and implement high-performance parallel/distributed applications. Monitoring such applications should also be able to exploit the massive infrastructure. To this end, we now discuss parallelization of our SMT-based monitoring technique.

Let G be a sequence of g segments $G = seg_1 seg_2 \cdots seg_g$. Our idea is to create a job queue for each available computing core, and then distribute the segments evenly across all the queues to be monitored by their respective cores independently. However, simply distributing all the segments across cores is not enough for obtaining a correct result. For example, consider formula $\varphi = a \,\mathcal{U}\, b$ and two segments, seg_1 and seg_2 across two cores, Cr_1 and Cr_2, respectively. In order for the monitor running on Cr_2 to give the correct verdict, it must know the result of the monitor running on Cr_1. In a scenario, where Cr_1 observes one or more $\neg a$ in seg_1, a violation must be reported even if Cr_2 does not observe b and no $\neg a$.

	seg_1			seg_2			seg_3			seg_4		
q_0	q_0	$q_\top$	$q_\bot$	q_0	$q_\top$	$q_\bot$	q_0	$q_\top$	$q_\bot$	q_0	$q_\top$	$q_\bot$
	T	F	F	T	T	F	T	T	T	T	T	T
$q_\top$	q_0	$q_\top$	$q_\bot$	q_0	$q_\top$	$q_\bot$	q_0	$q_\top$	$q_\bot$	q_0	$q_\top$	$q_\bot$
	F	F	F	F	T	F	F	T	F	F	T	F
$q_\bot$	q_0	$q_\top$	$q_\bot$	q_0	$q_\top$	$q_\bot$	q_0	$q_\top$	$q_\bot$	q_0	$q_\top$	$q_\bot$
	F	F	F	F	F	T	F	F	T	F	F	T

Figure 3.7: Reachability Matrix for $a \mathcal{U} b$.

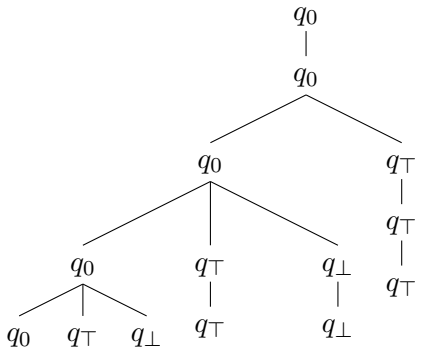


Figure 3.8: Reachability Tree for $a \mathcal{U} b$.

Generally speaking, the temporal order of events makes independent evaluation of segments impossible for LTL formulas. Of course, some formulas such as *safety* (e.g., $\Box p$) and *co-safety* (e.g., $\Diamond q$) properties are exceptions.

For our automata-based approach, we address this problem in two steps. Let $\mathcal{M}_\varphi = (\Sigma, Q, q_0, \delta, \lambda)$ be an $\mathtt{LTL}_3$ monitor. Our first step is to create a 3-dimensional reachability matrix RM by solving the following SMT decision problem: given a current monitor state $q_j \in Q$ and segment seg_i, can this segment reach monitor state $q_k \in Q$, for all $i \in [1, g]$, and $j, k \in [0, |Q| - 1]$. If the answer to the problem is affirmative, then we mark $RM[i][j][k]$ with *true*, otherwise with *false*. This is illustrated in Fig. 3.7 for the monitor shown in Fig. 2.1, where the grey cells are filled arbitrarily with the answer to the SMT problem. This step can be made embarrassingly parallel, where each element of RM can be computed independently by a different computing core. One can optimize the construction of RM by omitting

redundant SMT executions. For example, if $RM[i][j][\top] = \mathit{true}$, then $RM[i'][\top][\top] = \mathit{true}$ for all $i' \in [i, |Q| - 1]$. Likewise, if $RM[i][j][\bot] = \mathit{true}$, then $RM[i'][\bot][\bot] = \mathit{true}$ for all $i' \in [i, |Q| - 1]$.

The second step is to generate a verdict reachability tree from RM. The goal of the tree is to check if a monitor state $q_m \in Q$ can be reached from the initial monitor state q_0. This is achieved by setting q_0 as the root and generating all possible paths from q_0 using RM. That is, if $RM[i][k][j] = \mathit{true}$, then we create a tree node with label q_j and add it as a child of the node with the label q_k. Once the tree is generated, if q_m is one of the leaves, only then we can say q_m is reachable from q_0. In general, all leaves of the tree are possible monitoring verdicts. Note that creation of the tree is achieved using a sequential algorithm. For example, Fig.3.8 shows the verdict reachability tree generated from the matrix in Fig. 3.7.

For our progression-based approach, we adhere to a similar technique for parallelized monitoring as our automata-based approach. The key difference being, in the progression-based approach subformulas are used, whereas in the automata-based approach different states are used. As an example, the previous formula $\varphi = a \; \mathcal{U} \; b$ will be broken into two subformulas $\varphi_1 = \Box a$ and $\varphi_2 = \Diamond b$, before creating the reachibility matrix, and then generating the verdict for both these subformulas.

Lemma 5. *A distributed computation $(\mathcal{E}, \rightsquigarrow)$ of length l satisfies an* `LTL` *formula φ if and only if the parallelized monitoring technique satisfies φ. That is,*

$$\top \in [(\mathcal{E}, \rightsquigarrow) \models_3 \varphi] \iff \lambda(q) = \top$$

and,

$$\bot \in [(\mathcal{E}, \rightsquigarrow) \models_3 \varphi] \iff \lambda(q) = \bot$$

Where $q \in Q$ is some leaf node in the verdict reachability tree generated from RM during the parallelized monitoring process and λ is the labelling function in $\mathcal{M}_\varphi$.

Base Case: *Let us first consider the case where there is only one segment. That is, $l = g$.*

($\Rightarrow$) If $\top \in [(\mathcal{E}, \rightsquigarrow) \models_3 \varphi]$ (resp., $\bot \in [(\mathcal{E}, \rightsquigarrow) \models_3 \varphi]$), then according to the construction of the corresponding verdict reachability tree made from the RM, the root node

q_0 must have a child $q_\top$ (resp., $q_\bot$), such that, $\lambda(q_\top) = \top$ (resp., $\lambda(q_\bot) = \bot$). This child is also a leaf node, as the height of a verdict reachability tree is 2 when there is only one segment.

($\Leftarrow$) We can trivially show that if $\lambda(q_\top) = \top$ (resp., $\lambda(q_\bot) = \bot$), that is, if $q_\top$ (resp., $q_\bot$) is reachable from q_0, then $\top \in [(\mathcal{E}, \rightsquigarrow) \models_3 \varphi]$ (resp., $\bot \in [(\mathcal{E}, \rightsquigarrow) \models_3 \varphi]$).

Hypothesis: *Let us assume the proof as been established for $l = g \times k$. Now we consider $l = q \times (k + 1)$ as the segment length.*

($\Rightarrow$) If $\top \in [(\mathcal{E}, \rightsquigarrow) \models_3 \varphi]$ (resp., $\bot \in [(\mathcal{E}, \rightsquigarrow) \models_3 \varphi]$), then according to our assumption, there must be at least one node at height $k + 1$ (height of the leaf nodes where there are k segments), such that $\lambda(q_\top) = \top$ (resp., $\lambda(q_\bot) = \bot$). Now for $k + 1$ number of segments, according to the construction of the corresponding verdict reachability tree made from the RM, the node $q_\top$ (resp., $q_\bot$) can only have the child $q_\top$ (resp., $q_\bot$). Therefore, there must be at least one node at height $k + 2$ (height of the leaf nodes when there are $k + 1$ segments), such that $\lambda(q_\top) = \top$ (resp., $\lambda(q_\bot) = \bot$).

($\Leftarrow$) We can trivially show that if $\lambda(q_\top) = \top$ (resp., $\lambda(q_\bot) = \bot$), that is, if $q_\top$ (resp., $q_\bot$) is reachable from q_0, then $\top \in [(\mathcal{E}, \rightsquigarrow) \models_3 \varphi]$ (resp., $\bot \in [(\mathcal{E}, \rightsquigarrow) \models_3 \varphi]$).

3.5 Case Studies and Evaluation

In this section, we emphasize on analyzing our SMT-based solution without digressing into analyzing other dimensions such as instrumentation, data collection, data transfer, monitoring, etc., as given the distributed setting, runtime will be the dominant factor over any other kind of overhead. We evaluate our proposed technique using synthetic experiments, Cassandra (a distributed database), and the RACE dataset from NASA [MGS19].

3.5.1 Implementation and Experimental Setup

Each experiment can be divided into three phases: (1) data generation, (2) data collection and (3) data verification. For data-generation, we develop a synthetic program that randomly generates a distributed computation (i.e., the behavior of a set of programs in terms of their local computations and inter-process communication). Generating synthetic

experimental data offer benefits that enable us to draw comparison between different parameters and their effect on the approach. For example, generating data for different values of ε is beneficial to study its effect on the runtime and the number of false warning verdicts of our approach.

When developing the synthetic distributed system as part of our experiment, we ensure a partially-synchronous setting by including an HLC implementation. We use a uniform distribution $(0, 2)$ to define the type of event (local computation, send and receive message) and a flip-coin distribution for computing the atomic propositions that are true at each local computation event. Although the events in our synthetic experiments in Section 3.5.2 are uniformly distributed over the length of the trace, the event distribution as part of the Cassandra experiments in Section 3.5.3 are affected by the network latency and other external factors. In addition, we assume that that there is an external data collection program which keeps track of the data/states of the system under verification. It generates the trace logs which is used by the monitoring program to verify against the given `LTL` specifications mentioned in Figure 3.9b.

For data verification, we consider the following parameters: (1) number of processes ($|\mathcal{P}|$), (2) computation duration (l secs), (3) segment length (g), (4) event rate (r events/process/sec), (5) maximum clock skew (ϵ), (6) depth of the automaton (d) and number of nested temporal operators ($|\phi|$) for the `LTL` formula under monitoring. The main *metric* is to measure the runtime of SMT solving for each configuration of the parameters. Note that the time axis is shown in log-scale in all the plots presented in this section. When we analyze the effects of one parameter by holding the value of all the other parameters at a relevant constant value. In all the graphs, we compare the runtime of our automata-based approach against the progression based approach. We use a MacBook Pro with Intel i7-7567U(3.5Ghz) processor, 16GB RAM, 512 SSD and g++ Apple clang version 12.0.5 (clang-1205.0.22.9) interface to the Z3 SMT-solver [dMB08] to generate the traces. To evaluate our parallel algorithm, we use a server with 2x Intel Xeon Platinum 8180 (2.5GHz) processor, 768GB

RAM, 112 vcores and g++(GCC) 9.3.1 interface to the Z3 SMT-solver [dMB08]. Unless specified otherwise, the system under consideration has $|\mathcal{P}| = 2$, $l = 2$ sec, $g = 250ms$, $r = 10$ events/process/sec, $\epsilon = 250ms$ and $d = 3$.

3.5.2 Analysis of Results – Synthetic Experiments

In this set of experiments, we exhaust all the available parameters and note how it affects SMT solving. We test each parameter individually to study its effect on runtime. As our generated synthetic data does not depend on any external factors, we induce a delay to not only limit the number of events happening at every time unit, but also to ensure uniform distribution of events over the execution of each process. We use a uniform distribution $(0, |\Sigma|)$ to assign a value to each local computation event in each process. We only use one CPU core for the following experimental results.

Overall, we notice an improvement of around 35% when the progression based technique is compared to the other automata based approach. This improvement in performance owes to two main reasons: (1) compared to the automata-based approach, the `LTL` constrains in our progression-based approach is less demanding in terms of computational complexity. Each sub-formula consists of mostly one atomic proposition as opposed to multiple atomic propositions in each path of the automaton, which in turn speeds up the overall verification process, and (2) the total number of SMT-instances needed is fewer due to the less number of sub-formulas compared to automaton paths given the same specification. We now analyze the results in detail.

Impact of predicate structure. In this experiment (Figure 3.9a), we consider different predicate distribution over AP for the formula, φ_1, i.e., how many processes are involved with a particular predicate. We consider different predicate structures: $O(1)$, $O(n)$, $O(n^2)$ and $O(n^3)$ which signifies the order of the number of SMT-encodings that need to be generated for the given distribution of predicates. As can be seen, the progression based technique outperforms the automata-based technique overall by 35% on average.

Having said that, during our experiments when comparing the runtime of our monitoring approach for increasing number of sub-formulas, we observe a slight decrease in the overall efficiency in runtime when using the progression-based approach compared to the automata-based approach. Since the progression-based approach is based on evaluating each sub-formula, there exists an LTL formula where the number of sub-formulas is more than the number of paths in the corresponding automata, and thus, the the progression-based approach might not be as efficient as the automata-based approach in such a scenario.

For example, consider a formula, $\varphi = \Diamond a \vee \Diamond b \vee \Diamond c$, where the automata has two states, which makes the number of paths to be 2. However, the progression involves 3 sub-formulas, which makes the progression based approach less efficient than its automata counterpart. We would like to point out that the formula can be rewritten as $\Diamond(a \vee b \vee c)$, which makes both the approaches yield similar results. Thus we hypothesize that for all LTL formulas, the progression-based approach will be more (if not equally) efficient to that of the automata-based approach.

Impact of LTL formula. Given an LTL formula, the depth of nested temporal operators plays an important role as suggested by Fig. 3.9b. We experimental with the following LTL formula and the progression based technique achieved an average improvement of 32.8% compared to the automata-based one.

$$\varphi_1 = \Box p \qquad d = 2 \qquad |\phi| = 1$$

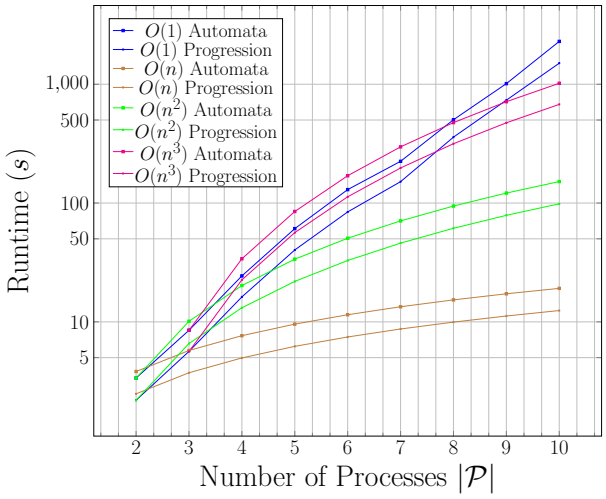
$$\varphi_2 = \Box(q \rightarrow \Box p) \qquad d = 3 \qquad |\phi| = 2$$

$$\varphi_3 = \Box((q \wedge \Diamond r) \rightarrow (\neg p \,\mathcal{U}\, r)) \qquad d = 4 \qquad |\phi| = 3$$

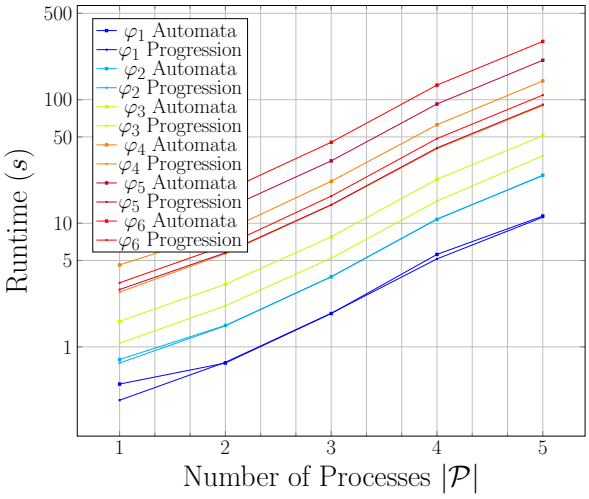
$$\varphi_4 = \Box((q \wedge \Diamond r) \rightarrow (\neg p \,\mathcal{U}\, (r \vee (s \wedge \neg p \wedge \bigcirc(\neg p \,\mathcal{U}\, t))))) \qquad d = 5 \qquad |\phi| = 8$$

$$\varphi_5 = \Diamond r \rightarrow (s \wedge \bigcirc(\neg r \,\mathcal{U}\, t) \rightarrow \bigcirc(\neg r \,\mathcal{U}\, (t \wedge \Diamond p))) \qquad d = 6 \qquad |\phi| = 8$$

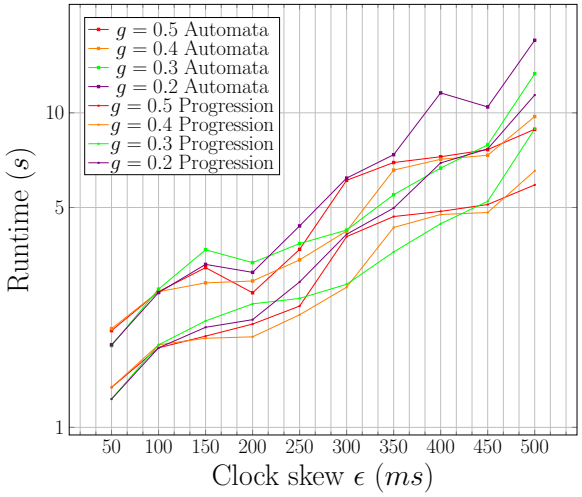
$$\varphi_6 = \Box((q \wedge \Diamond r) \rightarrow (s \wedge \bigcirc(\neg r \,\mathcal{U}\, t) \rightarrow \bigcirc(\neg r \,\mathcal{U}\, (t \wedge \Diamond p))) \,\mathcal{U}\, r) \qquad d = 7 \qquad |\phi| = 9$$



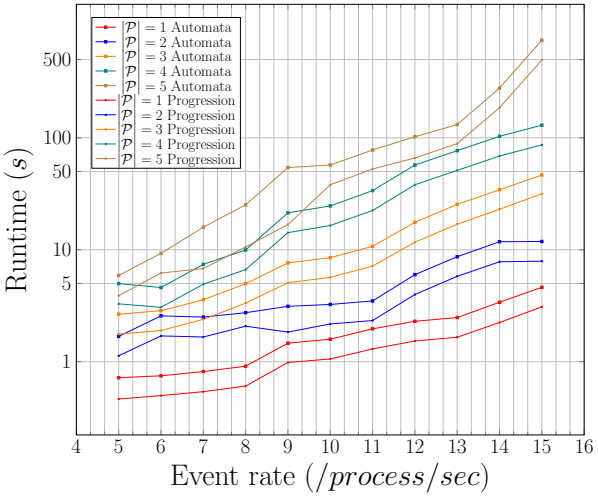
(a) Predicate Structure



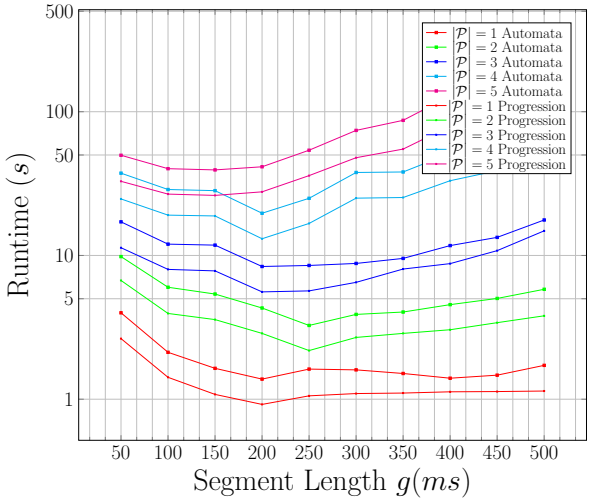
(b) LTL Formula



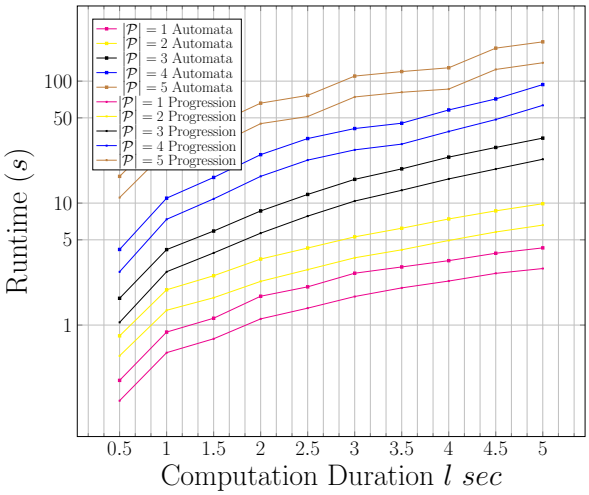
(c) Epsilon



(d) Event Rate



(e) Segment Length



(f) Computation Duration

Figure 3.9: Synthetic experiments – impact of different parameters.

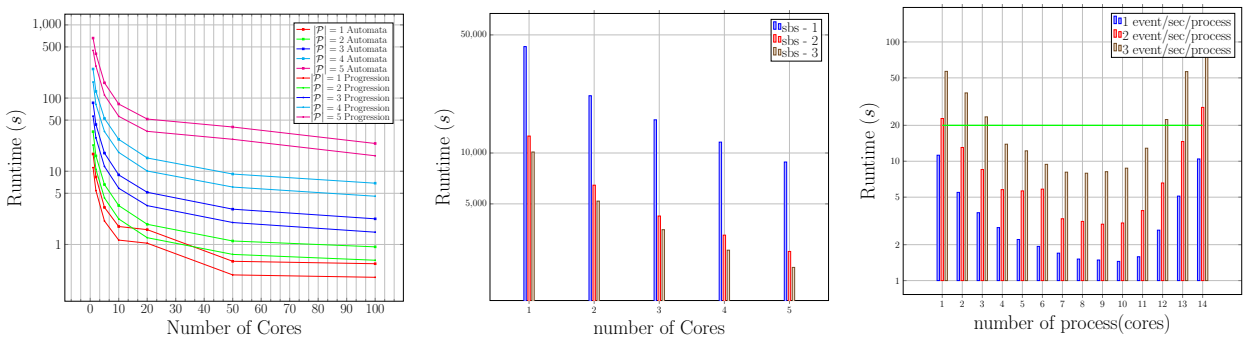
Impact of partial synchrony. Figure 3.9c shows an expected result where increasing clock skew ϵ results in greater runtime as the number of possible concurrent events across processes increases exponentially. When comparing with the automata-based approach , the progression-based technique yields us an improvement of 33.36%.

Impact of event rate. Figure. 3.9d shows that our approach breaks even with the computation duration for $|\mathcal{P}| = 3$ for an event rate of $5 events/process/sec$. However, increasing the event rate increases the search space for the SMT solver. Overall we improve by 34.4% by using the progression-based technique compared to the automata-based technique.

Impact of segment count. Increasing the segment length increases the number of events to be worked with, and therefore, exponentially increasing the runtime of our approach. In Fig. 3.9e, we do not see much improvement for $|\mathcal{P}| = 1, 2$, since the number of events is not large enough to make an impact. However, we see better performance with low segment length for higher number of processes. Note, the runtime increases for very small segment length, since the time taken to generate a higher number of SMT encodings outweigh the performance gain from smaller segments. Here too, we notice an improvement of 32.6% for the progression-based technique over the automata-based technique.

Impact of computation duration. In this experiment (Fig. 3.9f), we increase computation duration and measure its effect on runtime. With increasing computation duration, the number of segments needed to verify the longer computation increases, and thereby resulting in a linear increase of the runtime. The progression-based approach improves the runtime by 33.1% when compared to the automata-based approach.

Impact of parallelization. Distributing the verification among multiple cores improves the performance of the approach by a considerable amount. As seen in Figure 3.10a, increasing the number of cores from 1 to 10 improves the performance by a huge margin. However, increasing it further shows little improvement, as the time taken for generating the SMT encodings starts to dominate the time taken to solve it. An improvement of 33.8% is



(a) Synthetic Data (b) SBS Data (c) Google Data

Figure 3.10: Impact of parallelization on different data.

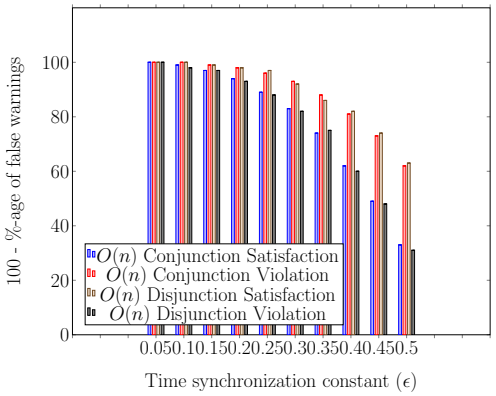


Figure 3.11: False Warnings for Synthetic Data.

achieved for progression-based approach when compared to automata-based approach.

Impact of ϵ on false warnings. As discussed in Section 2.4, the monitor does not have access to the global clock, it can report events as concurrent, when in reality, one happened before the other in the system under observation. However, during this experiment, we keep track of the global clock values separately, which gives us full knowledge over the total ordering of all events. Thus, allowing us to study and report the *real verdicts* alongside the *reported verdicts*. We observe that the monitor sometimes report *false warnings*, that is, it reports both verdicts (satisfaction and violation), when in reality, only one has occurred. Note that the monitor never fails to report real verdicts. However, it may report false warnings alongside real verdicts on some occasions. Although this does not change the correctness of the approach, it may still include false warnings as part of the set of evaluated results.

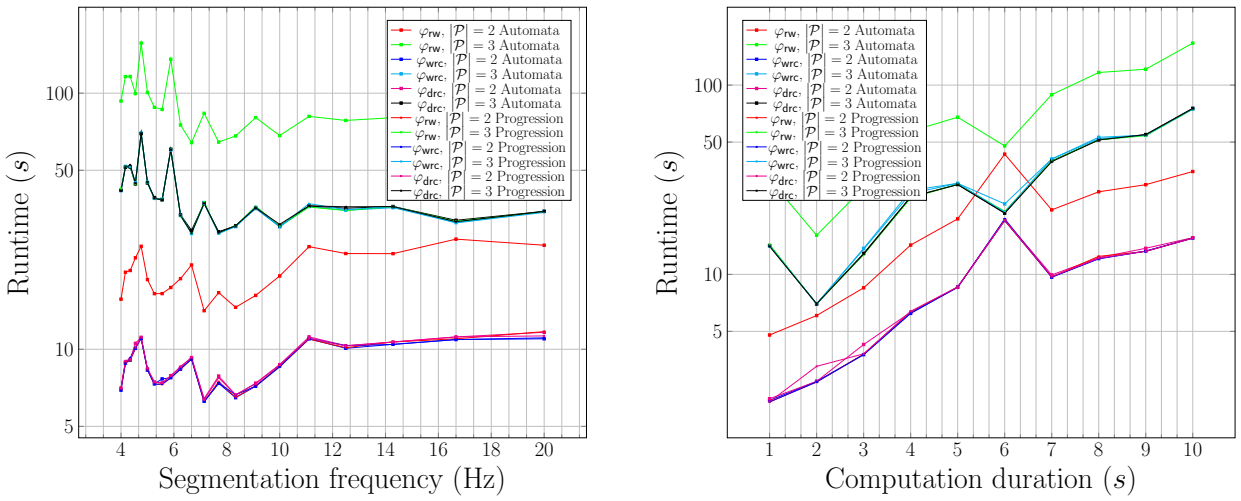
In Figure 3.11, we observe that with the increase of the maximum clock skew ϵ, the number of false warnings increases. The increase in false warnings is attributed to the fact that as the value of ϵ increases, so does the number of events considered as concurrent by the monitor.

Additionally, we observe that the number of false warning is greatly influenced by the predicate structure of the `LTL` formula, as evident from Figure 3.11. For $O(n)$ conjunctive satisfaction formula monitoring and $O(n)$ disjunctive violation formula monitoring, false warnings might occur if any one of the n sub-formulas are violated or satisfied, respectively. Therefore, we see a higher number of false warnings. Similarly, for $O(n)$ disjunctive satisfaction formula monitoring and $O(n)$ conjunctive violation formula monitoring, false warnings might occur if all of the n sub-formulas are violated or satisfied, respectively. Therefore, we see a lower number of false warnings.

3.5.3 Case Study 1: Cassandra

Cassandra [LM10] is a No-SQL distributed database management system. We simulate a distributed database with two data-centers: one cluster consisting of 4 nodes, and the other cluster consisting of 3 nodes, with one node from each cluster serving as the seed node. All data is replicated among every node in both the clusters. Each node runs on *Red Hat OpenStack Platform* using 4 VCPUs, 4GB RAM, Ubuntu 1804, Cassandra 3.11.6, and Java 1.8.0_252. We have also simulated a system of multiple processes where each process is responsible for the basic database operations (read, write and update). These processes are also capable of inter-process communication that allows for informing other processes in case of a write of a new entry to the database.

To make our simulated database realistic, we tested the latency of our system to the ones offered by Google Cloud, Microsoft Azure and Amazon Web Services. The fastest response was clocked at $41ms$ compared to $100ms$ from our system. The reason behind such a high latency when compared to the industry standard owes to the slow bandwidth and



(a) Segment Length

(b) Computation Duration

Figure 3.12: Cassandra experiments.

infrastructure differences. We consider a latency of $100ms$ for all our experiments, and fix maximum clock skew ϵ at $250ms$.

We design the processes such that each process is capable of reading, writing, or updating the entries in the database. We use a $(0, 2)$ uniform distribution to select the type of the operation that is to be performed by the process. Once there is any kind of addition from the write operation, the change is notified to the other processes using the inter-process communications. We consider no loss of messages in transmission and all messages are read by the receiving process immediately once they are received.

In a database, consistency level helps maintain the minimum of replications that needs to be performed on an operation in order to consider the operation to be successfully executed. According to the recommendations from Cassandra the sum of read and write consistency should be more than the replication factor so as to remove any chances of read or write anomaly in the database. We aim to monitor and identify *read/write anomalies* in the database using runtime monitoring techniques. The corresponding `LTL` specification becomes:

$$\varphi_{rw} = \bigwedge_{i=0}^{n} \Big(write(i) \rightarrow \Diamond read(i) \Big)$$

where n is the number of read/write operations.

One of the challenges for using a distributed database such as Cassandra is the lack of normalization (of database) capabilities. Therefore, we aim to monitor *write reference check* and *delete reference check*. We introduce two tables:

$$\textsc{Student}(id, name) \qquad \textsc{Enrollment}(id, course)$$

We enforce the write and delete reference check on the tables above. For a write in the ENROLLMENT table, it should always be preceded by a write in the STUDENT table with the same *id*. Similarly, for a delete from the STUDENT table, it should always be preceded by a delete from the ENROLLMENT table with the same *id*. These enforces no insertion and deletion anomaly, and therefore, leads to the following LTL specification:

$$\varphi_{wrc} = \neg\Big(\neg write(\textsc{Student}.id)\ \mathcal{U}\ write(\textsc{Enrollment}.id)\Big)$$

$$\varphi_{drc} = \neg\Big(\neg delete(\textsc{Enrollment}.id)\ \mathcal{U}\ delete(\textsc{Student}.id)\Big)$$

Extreme load scenario. Figure 3.12b and 3.12a plot runtime vs computation duration and runtime vs segmentation frequency respectively, under full read/write load allowed by our network. When compared with the results from that of the synthetic experiments, these results are slightly noisier. This owes to the fact that in the synthetic experiments, the events were evenly spread over the entire computation duration, whereas here they are not uniform. Database operations involving network communications (read, write and update) takes an average of 100ms, however sending and receiving of messages are inter-process communication, and takes about 10-15ms, making the overall event distribution non-uniform. When comparing with the automata-based approach, we do not see much improvement when monitoring φ_{wrc} or φ_{drc} using progression based approach. However, when monitoring φ_{rw}, we observe an average improvement of 55.53%.

Moderate load scenario. In Figure 3.12b, we were able to make even for number of processes as low as 2. Now, to look for a real-life example with moderate database operations

we consider Google Sheets API, which allows a maximum of 500 requests per 100 seconds per project and a 100 requests per second per user, i.e., on an average 5 events/sec per project and a user can only generate 1 event/sec. To evaluate how our approach performs in such a scenario, we increase the number of processes and the number of cores available to monitor such a system to study the time taken to verify the trace generated by such a system. We plot our findings in Fig. 3.10c, and notice that we break even for an event rate of 3 events/sec/user considering the progression-based approach. This is a significant improvement over the automata-based approach, where we could only break even for an event rate of 2 events/sec/user. Our algorithm performs wells when the number of processes are 7, 8 and 9 which is much more than what is permitted by Google. This allows for us to be confident that our approach can pave way for implementation in a real-life settings.

3.5.4 Case Study 2: RACE

Runtime for Airspace Concept Evaluation (RACE) [MGS19] is a framework developed by NASA that is used to build an event based, reactive airspace simulation. We use a dataset developed using this RACE framework (*https://github.com/NASARace/race-data*). This dataset contains three sets of data collected on three different days. Each set was recorded at around 37° N Latitude and 121° W Longitude. The dataset includes all 8 types of messages being sent by the SBS unit by using a Telnet application to listen to port 30003, but we only use the messages with ID $MSG - 3$ which is the Airborne Position Message and includes a flight's latitude, longitude and altitude using which we verify the mutual separation of all pairs of aircraft.

On analyzing the dataset, we observe that the time difference between the time message was generated to the time message was logged is usually less than a second apart, thus we considered an $\epsilon = 1s$ over the time message was generated. Furthermore, calculating the distance between two coordinates is computationally expensive, as we need to factor in parameters such as curvature of earth. In order to speed up distance related calculations, we

consider a constant latitude of 111.2km and longitude of 87.62km, at the cost of a negligible error margin. We use these as constants and multiply them by the difference in latitude and longitude, and factor in the altitude to get the distance between two aircrafts. We verify *mutual separation* by considering the minimum separation between every pair of aircrafts to be 500m. From the dataset, we observe that each aircraft generates a message on at least 1 sec intervals. There are 3 separate datasets: sbs-1 consists of 293 aircrafts, 168,283 messages spread over 3 hours and 28 minutes and 58seconds; sbs-2 consists of 110 aircrafts, 64,218 messages spread over 1 hour 1 minute and 46 seconds; sbs-3 consists of 97 aircrafts, 64,162 messages spread over 49 minutes and 42 seconds.

In Fig. 3.10b, we compare our achieved runtime against the three datasets available from RACE (labelled sbs-1, sbs-2 and sbs-3). We monitor the data in *real time*, with 10s long segments and ϵ of 1s. We test our approach using the parallelization technique introduced in 3.4.2 by using more number of cores on the processor and utilize all available cores. Our results break even for 4 cores. This makes our approach desirable for aircraft monitoring and similar systems such as IoT.

3.6 Summary and Limitation

In this chapter, we propose two monitoring technique which takes an `LTL` formula and a distributed computation as input. We apply a automata-based and a progression-based formula rewriting monitoring algorithm implemented as an SMT decision problem in order to verify the correctness of the distributed system with respect to the formula. We also conduct extensive synthetic experiments along with monitoring traces by Cassandra and RACE dataset by NASA.

The monitoring approach takes an `LTL` formula as specification which is not very expressive in the sense that it fails to express specifications for systems with time bounded execution. Additionally, as discussed in Section 3.5, the approach does not scale well when considering larger distributed system. Currently, the monitoring runtime increases

exponentially with increase in the number of processes or events being monitored. This is a big limiting factor when designing a verification approach which can work in real time.

Chapter 4

Runtime Verification for Time-bounded Temporal Specifications

4.1 Introduction

In this chapter, we advocate for a *runtime verification* (RV) approach, to monitor the behavior of a system of blockchains with respect to a set of temporal logic formulas. Applying RV to deal with multiple blockchains can be reduced to *distributed RV*, where a centralized or decentralized monitor observes the behavior of a distributed system in which processes do not share a global clock. Although RV deals with finite executions, the lack of a common global clock prohibits it from having a total unique ordering of events in a distributed setting. Put it another way, the monitor can only form a partial order of event which may result in different verification verdicts. Enumerating all possible partial ordering of events at run time

(Published) Ritam Ganguly, Yingjie Xue, Aaron Jonckheere, Parker Ljung, Benjamin Schornstein, Borzoo Bonakdarpour, and Maurice Herlihy, Distributed Runtime Verification of Metric Temporal Properties for Cross-Chain Protocols, IEEE 42nd International Conference on Distributed Computing Systems (ICDCS 2022).
(Under review) Ritam Ganguly, Yingjie Xue, Aaron Jonckheere, Parker Ljung, Benjamin Schornstein, Borzoo Bonakdarpour, and Maurice Herlihy, Distributed Runtime Verification of Metric Temporal Properties, Elsevier Journal of Parallel and Distributed Computing.

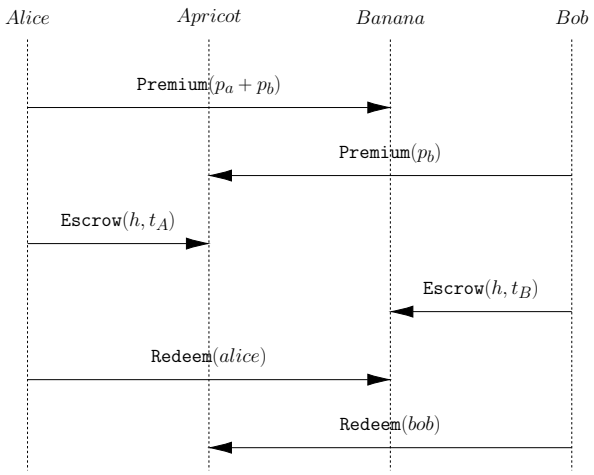


Figure 4.1: Hedged Two-party Swap.

incurs in an exponential blow up, making the approach not scalable. To add to this already complex task, most specifications for verifying blockchain smart contracts, come with a time bound. This means, not only the partial ordering of the events are at play when verifying, but also the actual physical time of occurrence of the events dictates the verification verdict.

In this chapter, we propose an effective, sound and complete solution to distributed RV for timed specifications expressed in the *metric temporal logic* (MTL) [Koy90]. To present a high-level view of MTL, consider the *two-party swap* protocol [XH21] shown in Fig 4.1. Alice and Bob, each in possession of Apricot and Banana blockchain assets respectively, wants to swap their assets between each other without being a victim of a sore loser attack [XH21] (A sore loser attack is a type of attack in cross-blockchain commerce. It occurs when one party decides to halt participation partway through, leaving other parties' assets locked up for a long duration). There is a number of requirements that should be followed by the conforming parties to discourage any attack on themselves. We use *metric temporal logic* (MTL) [Koy90] to express such requirements. One such requirement is, where Bob should not be able to redeem his asset before Alice redeems hers within eight time units can be represented by the MTL formula:

$$\varphi_{\mathsf{spec}} = \neg Apr.Redeem(bob)\ \mathcal{U}_{\ [0,8)}\ Ban.Redeem(alice).$$

We consider a fault proof central monitor which has the complete view of the system but

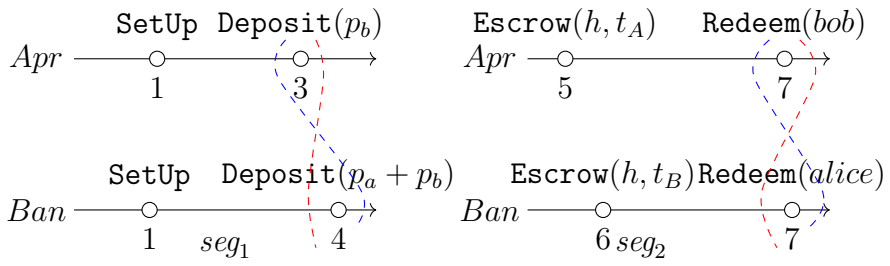


Figure 4.2: Progression Example.

has no access to a global clock. In order to limit the blow-up of states posed by the absence of a global clock, we make a practical assumption about the presence of a *bounded clock skew* ϵ between the local clocks of every pair of processes, guaranteed by a clock synchronization algorithm (e.g. NTP [Mil10]). This setting is known to be partially synchronous when we do not assume the presence of a global clock and limit the impact of asynchrony within clock drifts. Such an assumption limits the window of partial orders of events only within ϵ time units and significantly reduces the combinatorial blow-up caused by nondeterminism due to concurrency. Existing distributed RV techniques either assume a global clock when working with time sensitive specifications [BKMZ15, WOH19] or use untimed specifications when assuming partial synchrony [GMB21, MBAB21].

As often observed, the real clock skew between two processes is less than the maximum clock skew that is allowed by the system. Here, as a part of the monitoring scheme we want to take that into consideration when monitoring the distributed computation. We study the observed clock skew between every pair of processes and estimate the cumulative density function (cdf) that the clock skew follows. Based on our estimated cdf, we quantify the time of occurrence of each event in the distributed system and is able to calculate the probabilistic guarantee for the verdict of the monitor.

We introduce an SMT-based *progression-based* formula rewriting technique over distributed computations which takes into consideration the events observed thus far to rewrite the specifications for future extensions. Our monitoring algorithm accounts for all possible orderings of events without explicitly generating them when evaluating `MTL` formulas.

For example, in Fig. 4.2, we see the events and the time of occurrence in the two blockchains, Apricot(Apr) and Banana(Ban) divided into two segments, seg_1 and seg_2 for computational purposes. Considering maximum clock skew $\epsilon = 2$ and the clock skew cdf function return $0.25, 0.75, 1$ for observed clock skew $-1, 0, 1$ respectively for the specification φ_{spec}, at the end of the first segment, we have three possible rewritten formulas for the next segment along with the statistical guarantee of each of them:

$$\varphi_{\mathsf{spec}_1} = \neg Apr.Redeem(bob) \qquad \mathcal{U}_{[0,5)} Ban.Redeem(alice); pr = 0.1875$$

$$\varphi_{\mathsf{spec}_2} = \neg Apr.Redeem(bob) \qquad \mathcal{U}_{[0,4)} Ban.Redeem(alice); pr = 0.5625$$

$$\varphi_{\mathsf{spec}_3} = \neg Apr.Redeem(bob) \qquad \mathcal{U}_{[0,3)} Ban.Redeem(alice); pr = 0.15$$

This is possible due to the different ordering and different time of occurrence of the events $Deposit(p_b)$ and $Deposit(p_a + p_b)$. In other words, the possible time of occurrence of the event $Deposit(p_b)$ (resp. $Deposit(p_a + p_b)$) is either 2, 3 or 4 (resp. 3, 4, or 5) due to the maximum clock skew of 2. The probabilistic guarantee is calculated by the possible time of occurrence and the probability of the corresponding time of occurrence. To calculate the statistical guarantee of a verdict we see how was it reached. Here for, $\varphi_{\mathsf{spec}_1}$, we see that it can be reached by the time of occurrence of $Deposit(p_b)$ (resp. $Deposit(p_a + p_b)$) being either 2 or 3 (resp. 3). Thereby making the probability, $0.25 \times 0.25 + 0.5 \times 0.25 = 0.1875$. Similarly, we calculate the statistical guarantee of the other verdicts.

Likewise, at the end of seg_2, we have $\varphi_{\mathsf{spec}_1}$ and $\varphi_{\mathsf{spec}_2}$ evaluate to *true* where as $\varphi_{\mathsf{spec}_3}$ evaluate to *false*. This is because, even if we consider the scenario when *Ban.Redeem(alice)* occurs before *Apr.Redeem(bob)*, a possible time of occurrence of *Ban.Redeem(alice)* is 8 (resp. 6) which makes $\varphi_{\mathsf{spec}_3}$ (resp. $\varphi_{\mathsf{spec}_1}$ and $\varphi_{\mathsf{spec}_2}$) evaluate to *false* (resp. *true*). The statistical guarantee of the verdicts, *true* and *false* being 0.6875 and 0.6875. An interesting note here is that the sum of guarantee for *true* and *false* is not 1. This is because of the case, where both verdicts were equally likely. In Fig. 4.2, when the time of occurrence of

both *Ban.Redeem(alice)* and *Apr.Redeem(bob)* is 6, either order of occurrence is possible. Thereby making, both verdicts, equally likely.

We have fully implemented our technique (*https://github.com/TART-MSU/rv-mtl-blockc*) and report the results of rigorous experiments on monitoring synthetic data, using benchmarks in the tool UPPAAL [LPY97], as well as monitoring correctness, liveness and conformance conditions for smart contracts on blockchains. We put our monitoring algorithm to test, studying the effect of different parameters on the runtime and report on each of them.

4.1.1 Estimating Offset distribution

We run a number of diagnostic tests on every pair of processes in the distributed computation. Since the distributed system considered allows message passing, tests include sending and receiving of messages. A client process sends a dummy message to a server processes and once a the server process receives a message, it replies the client. Using the timestamps of the messages, we calculate the offset

$$\Theta = \frac{(t_1 - t_0) + (t_2 - t_3)}{2}$$

and the round trip delay

$$\delta = (t_3 - t_0) - (t_2 - t_1)$$

where, t_0 is the client's timestamp of the requested packet transmission, t_1 is the server's timestamp of the requested packet reception, t_2 is the server's timestamp of the response packet transmission and t_3 is the client's timestamp of the response packet reception. We derive the expression for the offset, for the request packet (resp. response packet),

$$t_0 + \Theta + \delta/2 = t_1 \qquad t_3 + \Theta - \delta/2 = t_2$$

Solving for Θ yields the time offset. This procedure is repeated for n times for each pair of processes and a vector of offsets is collected that defines the system, $(x_1, x_2, \cdots, x_n)$.

This vector of independent, identical distributed bounded random numbers constitute our sample. We assume that it follows a common cumulative distribution function $F(x)$. Then the empirical distribution function is defined by

$$\hat{F}(x) = \frac{number\ of\ elements\ in\ the\ sample \leq x}{n} = \frac{1}{n}\sum_{i=1}^{n} 1_{x_i \leq x}$$

where 1_a is the indicator of event a. This makes, $\hat{F}(x)$ an unbiased estimator of $F(x)$. Since the data in our setting is bounded by $(-\epsilon, \epsilon)$ where $\hat{F}(-\epsilon) = 0$ and $\hat{F}(\epsilon) = 1$. We decide to break the entire range into h steps where each step is of length $2\epsilon/h$. Thus, the estimated probability of a time offset, t is given by $p(t) = \hat{F}(t) - \hat{F}(t - 2\epsilon/h)$.

For example, for a vector of observed offsets, $(-4, -3, -3, -2, -2, -1, -1, -1, 0, 0, 0, 0, 0, 1, 1, 1, 2, 2, 3, 4)$, and $\epsilon = 5$, the estimated distribution function can be graphically represented by Figure 4.3 where $h = 5$. Thus we calculate the estimated probability of a time offset, $p(t = 0) = \hat{F}(0) - \hat{F}(-2) = 0.9 - 0.65 = 0.25$.

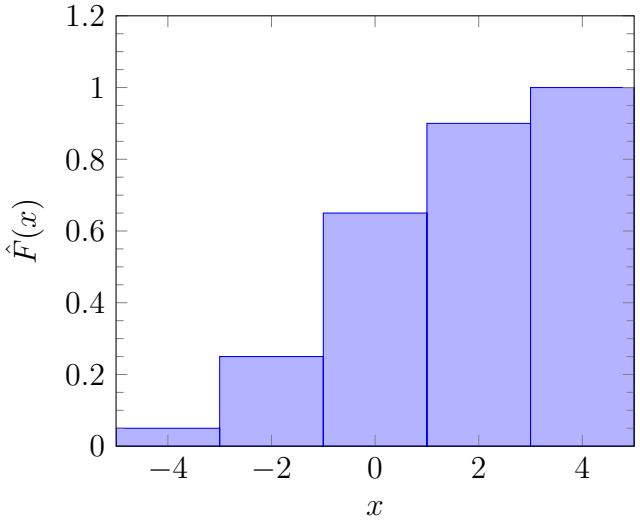


Figure 4.3: Example of a Cumulative Density Function.

For a better understanding as to how close our estimated distribution function, $\hat{F}(t)$, is to the real distribution, $F(t)$, we run a hypothesis test for the mean and standard deviation with a p-value of ≤ 0.05. It is also to be noted that any other non-parametric density

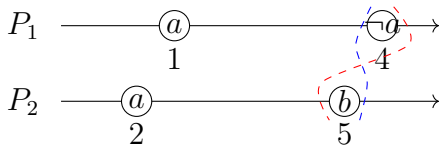


Figure 4.4: Differrent time interleaving of events.

estimation method can be used, eg. kernel density estimator, spectral density estimator, etc.

4.1.2 Formal Problem Statement

In a partially synchronous system, there are different possible ordering of events and each unique *ordering* of events [BF12] might evaluate to different RV verdicts. Let $(\mathcal{E}, \rightsquigarrow)$ be a distributed computation. A sequence of consistent cuts is of the form $\mathcal{C}_0\mathcal{C}_1\mathcal{C}_2\cdots$, where for all $i \geq 0$, we have (1) $\mathcal{C}_i \subset \mathcal{C}_{i+1}$ and (2) $|\mathcal{C}_i| + 1 = |\mathcal{C}_{i+1}|$, and (3) $\mathcal{C}_0 = \emptyset$. The set of all sequences of consistent cuts is denoted by $\mathbb{C}$. We note that in our view, the time interval $\mathcal{I}$ in the syntax of `MTL` represents the physical (global) time $\mathcal{G}$. Thus, when deriving all the possible traces given the distributed computation $(\mathcal{E}, \rightsquigarrow)$, we account for all different orders in which the events could possibly occur with respect to $\mathcal{G}$. This involves replacing the local time of occurrence of an event, e^i_σ with the set of events $\{e^i_{\sigma'} \mid \sigma' \in [\max\{0, \sigma - \epsilon + 1\}, \sigma + \epsilon)\}$. This is to account for the maximum clock drift that is possible on the local clock of a process when compared to the global clock. For example, given the computation in Fig. 4.4, a maximum clock skew $\epsilon = 2$ and a `MTL` formula, $\varphi = a\ \mathcal{U}_{\ [0,6)} b$, one has to consider all possible traces including $(a,1)(a,2)(b,4)(\neg a,5) \models \varphi$ and $(a,1)(a,2)(\neg a,4)(b,5) \not\models \varphi$.

In any typical system, the observed clock skew between any pair of processes is much less than the maximum clock skew that is allowed. To get a better understanding of the clock skew, we run some diagnostic tests (explained in Section 4.1.1) to help estimate the probability density function (pdf) defining the clock skew. Here, lets assume that the pdf is represented by a function $P(e^i_\sigma, \pi)$, such that given an event, e^i_σ and the function gives us the probability that the event took place at global time π.

Given a sequence of consistent cuts, it is evident that for all $j > 0$, $|\mathcal{C}_j - \mathcal{C}_{j-1}| = 1$ and

event $\mathcal{C}_j - \mathcal{C}_{j-1}$ is the last event that was added onto the cut $\mathcal{C}_j$. To translate monitoring of a distributed system into monitoring a trace with guarantees, we define a sequence of natural numbers as $\bar{\pi} = \pi_0\pi_1\cdots$, where $\pi_0 = 0$ and for each $j \geq 1$, we have $\pi_j = \sigma$, such that $\mathsf{front}(\mathcal{C}_j) - \mathsf{front}(\mathcal{C}_{j-1}) = \{e^i_\sigma\}$. To maintain time monotonicity, we only consider sequences where for all $i \geq 0$, $\pi_{i+1} \geq \pi_i$.

The set of all traces that can be formed from $(\mathcal{E}, \rightsquigarrow)$ is defined as:

$$\mathsf{Tr}(\mathcal{E}, \rightsquigarrow) = \Big\{ \mathsf{front}(\mathcal{C}_0)\mathsf{front}(\mathcal{C}_1)\cdots \mid \mathcal{C}_0\mathcal{C}_1\cdots \in \mathbb{C} \Big\}$$

In the sequel, we assume that every sequence α of frontiers in $\mathsf{Tr}(\mathcal{E}, \rightsquigarrow)$ is associated with a sequence $\bar{\pi}$. Thus, to comply with the semantics of MTL, we refer to the elements of $\mathsf{Tr}(\mathcal{E}, \rightsquigarrow)$ by pairs $(\alpha, \bar{\pi})$. The statistical guarantee associated with a verdict is calculated as the product of the probability of each event occurring at the time considered when generating the trace.

$$\mathcal{P}r(\alpha, \bar{\pi}) = \prod_{\forall j. \mathcal{C}_j - \mathcal{C}_{j-1} = \{e^i_\sigma\}} P(e^i_\sigma, \sigma')$$

Thus, we evaluate an MTL formula φ with respect to a computation $(\mathcal{E}, \rightsquigarrow)$ as follows:

$$[(\mathcal{E}, \rightsquigarrow) \models_F \varphi] = \Big\{ (\alpha, \bar{\pi}, 0) \models_F \varphi \mid (\alpha, \bar{\pi}) \in \mathsf{Tr}(\mathcal{E}, \rightsquigarrow) \Big\}$$

This boils down to having a set of verdicts and the corresponding probability of generating it, since a distributed computation may involve several traces and each trace might evaluate to a different verdict.

Overall idea of our solution To solve the above problem (evaluating all possible verdicts), we propose a monitoring approach based on formula-rewriting (Section 4.2) and SMT solving (Section 4.3). Our approach involves iteratively (1) chopping a distributed computation into a sequence of smaller segments to reduce the problem size, (2) progress the MTL formula for each segment for the next segment, which results in a new MTL formula by invoking an SMT solver and (3) calculate the sum of the probability of each trace that

yield the same MTL formula. Since each computation/segment corresponds to a set of possible traces due to partial synchrony, each invocation of the SMT solver may result in a different verdict.

4.2 Formula Progression for MTL

We start describing our solution by explaining the formula progression technique.

Definition 8. *A* progression function *is of the form* $\mathsf{Pr} : \Sigma^* \times \mathbb{Z}^*_{\geq 0} \times \Phi_{MTL} \rightarrow \Phi_{MTL}$ *and is defined for all finite traces* $(\alpha, \bar{\tau}) \in (\Sigma^*, \mathbb{Z}^*_{\geq 0})$*, infinite traces* $(\alpha', \bar{\tau}') \in (\Sigma^\omega, \mathbb{Z}^\omega_{\geq 0})$ *and* MTL *formulas* $\varphi \in \Phi_{MTL}$*, such that* $(\alpha.\alpha', \bar{\tau}.\bar{\tau}') \models \varphi$ *if and only if* $(\alpha', \bar{\tau}') \models \mathsf{Pr}(\alpha, \bar{\tau}, \varphi)$. □

Compared to the classic formula rewriting technique in [HR01b], here the function Pr takes a finite trace as input, while the algorithm in [HR01b] rewrites the formula after every observed state. When monitoring a partially synchronous distributed system, multiple verdicts are possible as a result of no unique ordering of events, as a result the classical state-by-state formula rewriting technique is of little use. The motivation of our approach comes from the fact that for computation reasons, we chop the computation into smaller segments and the verification of each segment is done through an SMT query. A state-by-state approach would incur in a huge number of SMT queries being generated.

Let $\mathcal{I} = [start, end)$ denote an interval. By $\mathcal{I} - \tau$, we mean the interval $\mathcal{I}' = [start', end')$, where $start' = \max\{0, start - \tau\}$ and $end' = \max\{0, end - \tau\}$. Also, for two time instances τ_i and τ_0, we let InInt(i) return *true* or *false* depending upon whether $\tau_i - \tau_0 \in \mathcal{I}$.

Progressing atomic propositions. For an MTL formula of the form $\varphi = p$, where $p \in \mathsf{AP}$, the result depends on whether or not $p \in \alpha(0)$. This marks as our base case for the other temporal and logical operators:

$$\mathsf{Pr}(\alpha, \bar{\tau}, \varphi) = \begin{cases} \mathit{true} & \text{if } \ p \in \alpha(0) \\ \mathit{false} & \text{if } \ p \notin \alpha(0) \end{cases}$$

Progressing negation. For an MTL formula of the form $\varphi = \neg\phi$, we have:

$$\mathsf{Pr}(\alpha, \bar{\tau}, \varphi) = \neg\mathsf{Pr}(\alpha, \bar{\tau}, \phi).$$

Progressing disjunction. Let $\varphi = \varphi_1 \vee \varphi_2$. Apart from the trivial cases, the result of progression of $\varphi_1 \vee \varphi_2$ is based on progression of φ_1 and/or progression of φ_2:

$$\mathsf{Pr}(\alpha, \bar{\tau}, \varphi) = \begin{cases} \mathit{true} & \text{if } \mathsf{Pr}(\alpha, \bar{\tau}, \varphi_1) = \mathit{true} \ \vee \ \mathsf{Pr}(\alpha, \bar{\tau}, \varphi_2) = \mathit{true} \\ \mathit{false} & \text{if } \mathsf{Pr}(\alpha, \bar{\tau}, \varphi_1) = \mathit{false} \ \wedge \ \mathsf{Pr}(\alpha, \bar{\tau}, \varphi_2) = \mathit{false} \\ \varphi_2' & \text{if } \mathsf{Pr}(\alpha, \bar{\tau}, \varphi_1) = \mathit{false} \ \wedge \ \mathsf{Pr}(\alpha, \bar{\tau}, \varphi_2) = \varphi_2' \\ \varphi_1' & \text{if } \mathsf{Pr}(\alpha, \bar{\tau}, \varphi_2) = \mathit{false} \ \wedge \ \mathsf{Pr}(\alpha, \bar{\tau}, \varphi_1) = \varphi_1' \\ \varphi_1' \vee \varphi_2' & \text{if } \mathsf{Pr}(\alpha, \bar{\tau}, \varphi_1) = \varphi_1' \ \wedge \ \mathsf{Pr}(\alpha, \bar{\tau}, \varphi_2) = \varphi_2' \end{cases}$$

Always and eventually operators. As shown in Algorithms 2 and 3, the progression for 'always', $(\Box_{\mathcal{I}}\,\varphi)$ and 'eventually', $(\Diamond_{\mathcal{I}}\,\varphi)$ depends on the value of $\mathsf{InInt}(\mathsf{i})$ and the progression of the inner formula φ. In Algorithms 2 and 3, we divide the algorithm into three cases: (1) line 4, corresponds to if $\mathcal{I}$ is within the sequence $\bar{\tau}$; (2) line 6, corresponds to where $\mathcal{I}$ starts in the current trace but its end is beyond the boundary of the sequence $\bar{\tau}$, and (3) line 9, corresponds to if the entire interval $\mathcal{I}$ is beyond the boundary of sequence $\bar{\tau}$. In Algorithm 2, we are only concerned about the progression of φ on the suffix $(\alpha^i, \bar{\tau}^i)$ if $\mathsf{InInt}(\mathsf{i}) = \mathit{true}$. In case, $\mathsf{InInt}(\mathsf{i}) = \mathit{false}$ the consequent drops and the entire condition equates to *true*. In other words, equating over all $i \in [0, |\alpha|]$, we are only left with conjunction of $\mathsf{Pr}(\alpha^i, \bar{\tau}^i, \varphi)$ where $\mathsf{InInt}(\mathsf{i}) = \mathit{true}$. In addition to this, we add the initial formula with updated interval for the next trace. Similarly, in Algorithm 3, equating over all $i \in [0, |\alpha|]$, if $\mathsf{InInt}(\mathsf{i}) = \mathit{false}$ the corresponding $\mathsf{Pr}(\alpha^i, \bar{\tau}^i, \varphi)$ is disregarded and the final formula is a disjunction of $\mathsf{Pr}(\alpha^i, \bar{\tau}^i, \varphi)$ with $\mathsf{InInt}(\mathsf{i}) = \mathit{true}$.

Progressing the until operator. Let the formula be of the form $\varphi_1 \ \mathcal{U}_{\ \mathcal{I}} \varphi_2$. According to the semantics of until, φ_1 should be evaluated to true in all states leading up to some

Algorithm 2: Always.

1: **function** $\mathsf{Pr}(\alpha, \bar{\tau}, \Box_{\mathcal{I}}\, \varphi)$
2: **if** $\mathcal{I}_{start} \leq \tau_{|\alpha|} - \tau_0$ **then**
3: **if** $\mathcal{I}_{end} \leq \tau_{|\alpha|} - \tau_0$ **then**
4: **return** $\bigwedge_{i\in[0,|\alpha|]} \left(\mathsf{InInt(i)} \rightarrow \mathsf{Pr}(\alpha^i, \bar{\tau}^i, \varphi)\right)$
5: **else**
6: **return** $\bigwedge_{i\in[0,|\alpha|]} \left(\mathsf{InInt(i)} \rightarrow \mathsf{Pr}(\alpha^i, \bar{\tau}^i, \varphi)\right) \wedge \Box_{[\mathcal{I}-(\tau_{|\alpha|}-\tau_0))}\, \varphi$
7: **end if**
8: **else**
9: **return** $\Box_{[\mathcal{I}-(\tau_{|\alpha|}-\tau_0))}\, \varphi$
10: **end if**
11: **end function**

Algorithm 3: Eventually.

1: **function** $\mathsf{Pr}(\alpha, \bar{\tau}, \Diamond_{\mathcal{I}}\, \varphi)$
2: **if** $\mathcal{I}_{start} \leq \tau_{|\alpha|} - \tau_0$ **then**
3: **if** $\mathcal{I}_{end} \leq \tau_{|\alpha|} - \tau_0$ **then**
4: **return** $\bigvee_{i\in[0,|\alpha|]} \left(\mathsf{InInt(i)} \wedge \mathsf{Pr}(\alpha^i, \bar{\tau}^i, \varphi)\right)$
5: **else**
6: **return** $\bigvee_{i\in[0,|\alpha|]} \left(\mathsf{InInt(i)} \wedge \mathsf{Pr}(\alpha^i, \bar{\tau}^i, \phi)\right) \vee \Diamond_{[\mathcal{I}-(\tau_{|\alpha|}-\tau_0))}\, \varphi$
7: **end if**
8: **else**
9: **return** $\Diamond_{[\mathcal{I}-(\tau_{|\alpha|}-\tau_0))}\, \varphi$
10: **end if**
11: **end function**

Algorithm 4: Until.

1: **function** $\mathsf{Pr}(\alpha, \bar{\tau}, \varphi_1\, \mathcal{U}_{\,\mathcal{I}} \varphi_2)$
2: **if** $\mathcal{I}_{start} \leq \tau_{|\alpha|} - \tau_0$ **then**
3: **if** $\mathcal{I}_{end} \leq \tau_{|\alpha|} - \tau_0$ **then**
4: **return** $\left(\bigwedge_{i\in[0,|\alpha|]} \left((\tau_i < \mathcal{I}_{start} + \tau_0) \rightarrow \mathsf{Pr}(\alpha^i, \bar{\tau}^i, \varphi_1)\right)\right) \wedge \left(\bigvee_{j\in[0,|\alpha|]} \left(\mathsf{InInt(j)} \wedge \mathsf{Pr}(\alpha, \bar{\tau}, \Box_{[0,\tau_j-\tau_0)}\, \varphi_1) \wedge \mathsf{Pr}(\alpha^j, \bar{\tau}^j, \varphi_2)\right)\right)$
5: **else**
6: **return** $\left(\bigwedge_{i\in[0,|\alpha|]} \left((\tau_i < \mathcal{I}_{start} + \tau_0) \rightarrow \mathsf{Pr}(\alpha^i, \bar{\tau}^i, \varphi_1)\right)\right) \wedge \left(\bigvee_{j\in[0,|\alpha|]} \left(\mathsf{InInt(j)} \wedge \mathsf{Pr}(\alpha, \bar{\tau}, \Box_{[0,\tau_j-\tau_0)}\, \varphi_1) \wedge \mathsf{Pr}(\alpha^j, \bar{\tau}^j, \varphi_2)\right) \vee \varphi_1\, \mathcal{U}_{\,(\mathcal{I}-(\tau_{|\alpha|}-\tau_0)\varphi_2}\right)$
7: **end if**
8: **else**
9: **return** $\left(\bigwedge_{i\in[0,|\alpha|]} \mathsf{Pr}(\alpha^i, \bar{\tau}^i, \varphi_1)\right) \wedge \varphi_1\, \mathcal{U}_{\,(\mathcal{I}-(\tau_{|\alpha|}-\tau_0)\varphi_2}$
10: **end if**
11: **end function**

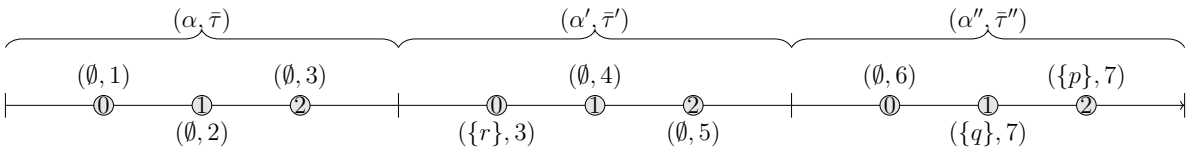


Figure 4.5: A trace example divided into three segments.

$i \in \mathcal{I}$, where φ_2 evaluates to true. We start by progressing φ_1 (resp. φ_2) as $\Box_{[0,\tau_i-\tau_0)}\,\varphi_1$ (resp. $\Diamond_{[\tau_i,\tau_i+1)}\,\varphi_2$) for some $i \in \mathcal{I}$. Since, we are only verifying the sub-formula, $\Diamond_{[\tau_i,\tau_i+1)}\,\varphi_2$, on the trace sequence $(\alpha, \bar{\tau})$, it is equivalent to verifying the sub-formula $\Diamond_{[0,1)}\,\varphi_2 \equiv \varphi_2$ over the trace sequence $(\alpha^i, \bar{\tau}^i)$. Similar to Algorithms 2 and 3, in Algorithm 4 we need to consider three cases. In lines 4, 6 and 9, following the semantics of until operator, we make sure for all $i \in [0, |\alpha|]$, if $\tau_i < \mathcal{I}_{start} + \tau_0$, φ_1 is satisfied in the suffix $(\alpha^i, \bar{\tau}^i)$. In addition to this there should be some $j \in [0, |\alpha|]$ for which if $\mathsf{InInt(j)} = \mathit{true}$, then the trace satisfies the sub-formula $\Box_{[0,\tau_j-\tau_0)}\,\varphi_1$ and $\Diamond_{[\tau_j,\tau_j+1)}\,\varphi_2$). In lines 6 and 9, we also accommodate for future traces satisfying the formula $\varphi_1\,\mathcal{U}_{\mathcal{I}}\varphi_2$ with updated intervals.

Example In Fig. 4.5, the time line shows propositions and their time of occurrence, for formula $\Diamond_{[0,6)}\,r \rightarrow (\neg p\,\mathcal{U}_{[2,9)}q)$. The entire computation has been divided into 3 segments, $(\alpha, \bar{\tau})$, $(\alpha', \bar{\tau}')$, and $(\alpha'', \bar{\tau}'')$ and each state has been represented by (s, τ):

- We start with segment $(\alpha, \bar{\tau})$. First we evaluate $\Diamond_{[0,6)}\,r$, which requires evaluating $\mathsf{Pr}(\alpha^i, \bar{\tau}^i, r)$ for $i \in \{0, 1, 2\}$, all of which returns the verdict *false* and there by rewriting the sub-formula as $\Diamond_{[0,4)}\,r$. Next, to evaluate the sub-formula $\neg p\,\mathcal{U}_{[2,9)}q$, we need to evaluate (1) $\mathsf{Pr}(\alpha^i, \bar{\tau}^i, \neg p)$ for $i \in \{0, 1\}$ since $\tau_i - \tau_0 < 2$ and both evaluates to *true*, (2) $\mathsf{Pr}(\alpha, \bar{\tau}, \Box_{[0,2)}\,\neg p)$ which also evaluates to *true* and (3) $\mathsf{Pr}(\alpha^2, \bar{\tau}^2, q)$ which evaluates as *false*. Thereby, the rewritten formula after observing $(\alpha, \bar{\tau})$ is $\Diamond_{[0,3)}\,r \rightarrow (\neg p\,\mathcal{U}_{[0,6)}q)$.
- Similarly, we evaluate the formula now with respect to $(\alpha', \bar{\tau}')$, which makes the sub-formula $\Diamond_{[0,3)}\,r$ evaluate to *true* at $\tau = 3$ and the sub-formula $\neg p\,\mathcal{U}_{[0,6)}q$ (there is no such $i \in \{0, 1, 2\}$ where $\tau_i - \tau_0 < 0$ and for all $j \in \{0, 1, 2\}$, $\mathsf{Pr}(\alpha'^j, \bar{\tau}'^j, q) = \mathit{false}$) is rewritten as $\neg p\,\mathcal{U}_{[0,4)}q$.

- In $(\alpha'', \bar{\tau}'')$, for $j = 1$, $\mathsf{Pr}(\alpha'', \bar{\tau}'', \Box_{[0,2)} \neg p) = \mathit{true}$ and $\mathsf{Pr}(\alpha''^j, \bar{\tau}''^j, q) = \mathit{true}$, and thereby rewriting the entire formula as *true*.

4.3 SMT-based Solution

4.3.1 SMT Entities

SMT entities represent the variables used to represent the distributed computation. After we have the verdicts for each of the individual sub-formulas, we use the progression laws discussed in Section 4.2 to construct the formula for the future computations.

Distributed Computation We represent a distributed computation $(\mathcal{E}, \rightsquigarrow)$ by a function $f : \mathcal{E} \rightarrow \{0, 1, \ldots, |\mathcal{E}| - 1\}$. To represent the happen-before relation, we define a $\mathcal{E} \times \mathcal{E}$ matrix called hbSet where $\mathsf{hbSet}[e^i_\sigma][e^j_{\sigma'}] = 1$ represents $e^i_\sigma \rightsquigarrow e^j_{\sigma'}$ for $e^i_\sigma, e^j_{\sigma'} \in \mathcal{E}$. Also, if $|\sigma - \sigma'| \geq \epsilon$ then $\mathsf{hbSet}[e^i_\sigma][e^j_{\sigma'}] = 1$, else $\mathsf{hbSet}[e^i_\sigma][e^j_{\sigma'}] = 0$. This is done in the pre-processing phase of the algorithm and in the rest of the chapter, we represent events by the set $\mathcal{E}$ and a happen-before relation by $\rightsquigarrow$ for simplicity.

In order to represent the possible time of occurrence of an event, we define a function $\delta : \mathcal{E} \rightarrow \mathbb{Z}_{\geq 0}$, where

$$\forall e^i_\sigma \in \mathcal{E}. \exists \sigma' \in [\max\{0, \sigma - \epsilon + 1\}, \sigma + \epsilon - 1]. \delta(e^i_\sigma) = \sigma'$$

Given an event, we map each possible time of occurrence of the vent with the respective probability using a function $p : \mathcal{E} \times \mathbb{Z}_{\geq 0} \rightarrow [0, 1]$, where $p(e^i_\sigma, \delta(e^i_\sigma))$ is some real number in the range $[0, 1]$ such that

$$\forall e^i_\sigma \in \mathcal{E}, \forall \sigma_1, \sigma_2 \in [\max\{0, \sigma - \epsilon + 1\}, \sigma + \epsilon - 1]. (\sigma_1 < \sigma_2) \rightarrow p(e^i_\sigma, \sigma_1) \leq p(e^i_\sigma, \sigma_1)$$

and

$$\forall e^i_\sigma \in \mathcal{E}. p(e^i_\sigma, \sigma + \mathit{epsilon} - 1) = 1; p(e^i_\sigma, \sigma - \mathit{epsilon} + 1) = 0$$

To connect events, $\mathcal{E}$, and propositions, AP, on which the `MTL` formula φ is constructed, we

define a boolean function $\mu : \mathsf{AP} \times \mathcal{E} \rightarrow \{true, false\}$. For formulas involving non-boolean variables (e.g., $x_1 + x_2 \leq 7$), we can update the function μ accordingly. We represent a sequence of consistent cuts that start from $\{\}$ and end in $\mathcal{E}$, we introduce an *uninterpreted function* $\rho : \mathbb{Z}_{\geq 0} \rightarrow 2^{\mathcal{E}}$ to reach a verdict, given it satisfies all the constrains explained in 4.3.2. Lastly, to represent the sequence of time associated with the sequence of consistent cuts, we introduce a function $\tau : \mathbb{Z}_{\geq 0} \rightarrow \mathbb{Z}_{\geq 0}$.

4.3.2 SMT Constraints

Once we have the necessary SMT entities, we move onto including the constraints for both generating a sequence of consecutive cuts and also representing the `MTL` formula as a SMT constraint.

Consistent cut constraints over ρ: In order to make sure the sequence of cuts represented by the uninterpreted function ρ, is a sequence of consistent cuts, i.e., they follow the happen-before relations between events in the distributed system:

$$\forall i \in [0, |\mathcal{E}|].\forall e, e' \in \mathcal{E}.\Big((e' \rightsquigarrow e) \wedge \big(e \in \rho(i)\big)\Big) \rightarrow \big(e' \in \rho(i)\big)$$

Next, we make sure that in the sequence of consistent cuts, the number of events present in a consistent cut is one more than the number of events that were present in the consistent cut before it:

$$\forall i \in [0, |\mathcal{E}|). \mid \rho(i+1) \mid = \mid \rho(i) \mid +1$$

Next, we make sure than in the sequence of consistent cuts, each consistent cut includes all the events that were present in the consistent cut before it, i.e, it is a subset of the consistent cut prior in the sequence.

$$\forall i \in [0, |\mathcal{E}|].\rho(i) \subset \rho(i+1)$$

The sequence of consistent cuts starts from $\{\}$ and ends at $\mathcal{E}$.

$$\rho(0) = \emptyset;\ \rho(|\mathcal{E}|) = \mathcal{E}$$

The sequence of time reflects the time of occurrence of the event that has just been added to the sequence of consistent cut:

$$\forall i \geq 1.\tau(i) = \delta(e^i_\sigma), \text{ such that } \rho(i) - \rho(i-1) = \{e^i_\sigma\}$$

We make sure the monotonicity of time is maintained in the sequence of time

$$\forall i \in [0, |\mathcal{E}|).\tau(i+1) \geq \tau(i)$$

Calculating statistical guarantee over ρ: The statistical guarantee of a verdict is the same as the probability of generating the corresponding trace which yielded the respective verdict. To avoid a iterative process of generating all possible traces, we use a consolidated method which limits the number of traces to be verified. Forall $i \geq 1$, if $\rho(i) - \rho(i-1) = \{e^i\sigma\}$, then we define two entities such that

$$\sigma_{start} = max\{\tau(i-1), \sigma - \epsilon + 1\}$$

and

$$\sigma_{end} = \delta(e^i_\sigma)$$

We define a function $P : \mathcal{E} \times \mathbb{Z}_{\geq 0} \times \mathbb{Z}_{\geq 0} \rightarrow [0, 1]$ which calculates the probability for the range of time of occurrence of the event given by $[\sigma_{start}, \sigma_{end}]$ as

$$P(e^i_\sigma, \sigma_{start}, \sigma_{end}) = p(e^i_\sigma, \sigma_{end}) - \mathsf{p}(e^i_\sigma, \sigma_{start})$$

The probability of generating the corresponding trace is given by

$$\mathcal{P}r(\rho, \tau) = \prod_{\forall i \in [0, |\mathcal{E}|].max\{\tau(i)\}} P(e^i_\sigma, \sigma_{start}, \sigma_{end})$$

where we aim to maximize the each $\tau(i)$

Constraints for `MTL` formulas over ρ: These constraints will make sure that ρ will not only represent a valid sequence of consistent cuts but also makes sure that the sequence

of consistent cuts satisfy the MTL formula. As is evident, a distributed computation can often yield two contradicting evaluation. Thus, we need to check for both satisfaction and violation for all the sub-formulas in the MTL formula provided. Note that monitoring any MTL formula using our progression rules will result in monitoring sub-formulas which are atomic propositions, eventually and globally temporal operators. Below we mention the SMT constrain for each of the different sub-formula. Violation (resp. satisfaction) for atomic proposition and eventually (resp. globally) constrain will be the negation of the one mentioned.

$$\begin{aligned}
\varphi = \mathsf{p} \qquad & \bigvee_{e \in \mathsf{front}(\rho(0))} \mu[\mathsf{p}, e] = \mathit{true}, \text{for } \mathsf{p} \in \mathsf{AP} \quad \text{(satisfaction, i.e., } \top\text{)} \\
\varphi = \Box_{\mathcal{I}} \varphi \qquad & \exists i \in [0, |\mathcal{E}|].\tau(i) - \tau(0) \in \mathcal{I} \wedge \rho(i) \not\models \varphi \quad \text{(violation, i.e., } \bot\text{)} \\
\varphi = \Diamond_{\mathcal{I}} \varphi \qquad & \exists i \in [0, |\mathcal{E}|].\tau(i) - \tau(0) \in \mathcal{I} \wedge \rho(i) \models \varphi \quad \text{(satisfaction, i.e., } \top\text{)}
\end{aligned}$$

A satisfiable SMT instance denotes that the uninterpreted function was not only able to generate a valid sequence of consistent cuts but also that the sequence satisfies the MTL formula given the computation. This result is then fed to the progression cases to generate the final verdict.

4.3.3 Segmentation of a Distributed Computation

We know that predicate detection, let alone runtime verification, is NP-complete [Gar02] in the size of the system (number of processes). This complexity grows to higher classes when working with nested temporal operators. To make the problem computationally viable, we aim to chop the computation, $(\mathcal{E}, \rightsquigarrow)$ into g segments, $(seg_1, \rightsquigarrow), (seg_2, \rightsquigarrow), \cdots, (seg_g, \rightsquigarrow)$. This involves creating small SMT-instances for each of the segments which improves the runtime of the overall problem. In a computation of length l, if we were to chop it into g segments, each segment would of the length $\frac{l}{g} + \epsilon$ and the set of events included in it can be given by:

$$seg_j = \Big\{ e^i_\sigma \mid \sigma \in \Big[max\big(0, \frac{(j-1)\times l}{g} - \epsilon\big), \frac{j \times l}{g} \Big] \wedge i \in [1, \mid \mathcal{P} \mid] \Big\}$$

Note that monitoring of a segment should include the events that happened within ϵ time of the segment actually starting since it might include events that are concurrent with some other events in the system not accounted for in the previous segment.

4.4 Case Study and Evaluation

In this section, we analyze our SMT-based solution. We note that we are not concerned about data collections, data transfer, etc, as given a distributed setting, the runtime of the actual SMT encoding will be the most dominating aspect of the monitoring process. We evaluate our proposed solution using traces collected from benchmarks of the tool UPPAAL [LPY97] (UPPAAL is a model checker for a network of timed automata. The tool-set is accompanied by a set of benchmarks for real-time systems. Here, we assume that the components of the network are partially synchronized.) models (Section 4.4.1) and a case study involving smart contracts over multiple blockchains (Section 4.4.2).

4.4.1 UPPAAL Benchmarks

Setup

Below we explain in details how each of the UPPAAL models work. In respect to our monitoring algorithm, we consider multiple instances of each of the models as different processes. Each event consists of the action that was taken along with the time of occurrence of the event. In addition to this, we assume a unique clock for each instance, synchronized by the presence of a clock synchronization algorithm with a maximum clock skew of ϵ.

The Train-Gate It models a railway control system which controls access to a bridge for several trains. The bridge can be considered as a shared resource and can be accessed by one train at a time. Each train is identified by a unique *id* and whenever a new train appears in

the system, it sends a *appr* message along with it's *id*. The Gate controller has two options: (1) send a *stop* message and keep the train in waiting state or (2) let the train cross the bridge. Once the train crosses the bridge, it sends a *leave* message signifying the bridge is free for any other train waiting to cross.

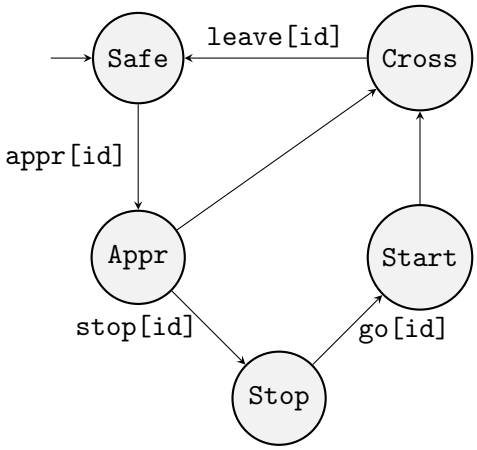


Figure 4.6: Train model.

The gate keeps track of the state of the bridge, in other words the gate acts as the controller of the bridge for the trains. If the bridge is currently not being used, the gate immediately offers any train appearing to go ahead, otherwise it sends a *stop* message. Once the gate is free again from a train leaving the bridge, it sends out a *go* message to any train that had appeared in the mean time and was waiting in the queue.

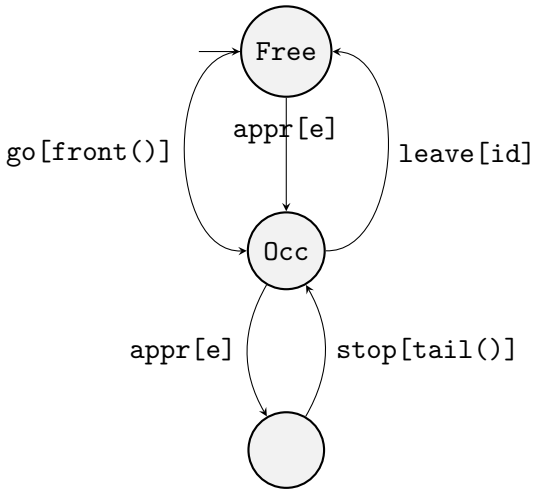


Figure 4.7: Gate model.

$$\varphi_1 = (\bigwedge_{i \in \mathcal{P}} \neg \mathit{Train[i].Cross}) \ \ \mathcal{U} \ \ \mathit{Train[1].Cross}$$

$$\varphi_2 = \bigwedge_{i \in \mathcal{P}} \Box \big(\mathit{Train[i].Appr} \rightarrow \Diamond (\mathit{Gate.Occ} \ \ \mathcal{U} \ \ \mathit{Train[i].Cross}) \big)$$

where $\mathcal{P}$ is the set of trains.

The Fischer's Protocol It is a mutual exclusion protocol designed for n processes. A process always sends in a request to enter the critical section (cs). On receiving the request, a unique pid is generated and the process moves to a *wait* state. A process can only enter into the critical section when it has the correct id. Upon exiting the critical section, the process resets the id which enables other processes to enter the cs

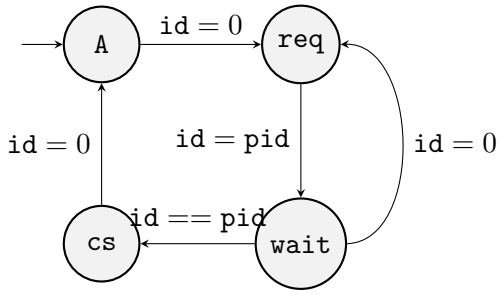


Figure 4.8: Fischer model.

$$\varphi_3 = \Box (\sum_{i \in \mathcal{P}} \mathit{P[i].cs} \leq 1)$$

$$\varphi_4 = \Box (\bigwedge_{i \in \mathcal{P}} \mathit{P[i].req} \rightarrow \Diamond_{\mathcal{I}} \mathit{P[i].cs})$$

The Gossiping People The model consists of n people, each having a private secret they wish to share with each other. Each person can *Call* another person and after a conversation, both person mutually knows about all their secrets. With respect to our monitoring problem, we make sure that each person generates a new secret that needs to be shared among others

infinitely often.

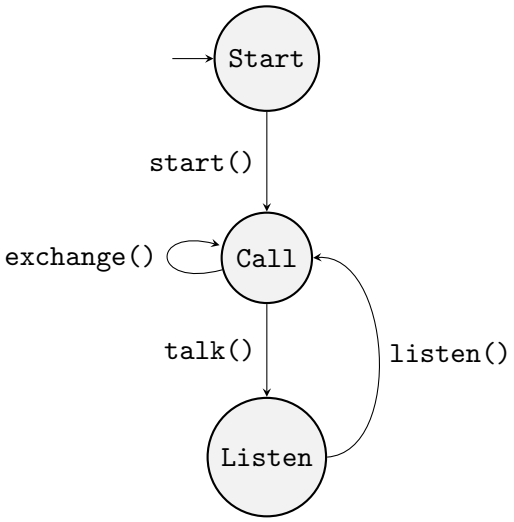


Figure 4.9: Gossiping people model.

$$\varphi_5 = \Diamond_{\mathcal{I}}(\bigwedge_{i,j \in \mathcal{P}} (i \neq j) \rightarrow \mathit{Person[i].secret[j]})$$

$$\varphi_6 = \bigwedge_{i \in \mathcal{P}} \Box(\Diamond_{\mathcal{I}}\, \mathit{Person[i].secrets})$$

Each experiment involves three steps: (1) offset calculation of the given distributed system, (2) distributed computation/trace generation and (3) trace verification. As stated earlier, the value of the offset ranges from $(-\epsilon, \epsilon)$ with 0 signifying that there is no skew between the two processes. To study as to how the offset distribution effects the statistical guarantee of a verdict, we make use of five different distribution.

- A truncated normal distribution, $TX_1 : (\mu = 0, \sigma = \frac{\epsilon}{1.5})$
- A truncated normal distribution $TX_2 : (\mu = 0, \sigma = \frac{\epsilon}{5})$
- A uniform distribution $U_1 : U(-\epsilon, \epsilon)$
- A uniform distribution $U_2 : U(-\frac{\epsilon}{2}, \frac{\epsilon}{2})$
- A sum of two truncated normal distribution, TX_3, with $(\mu = -\epsilon/2, \sigma = \frac{\epsilon}{3})$ and $(\mu = \epsilon/2, \sigma = \frac{\epsilon}{3})$.

The truncated normal distribution has limits of $(-\epsilon, \epsilon)$. For each UPPAAL model, we consider each pair of consecutive events are $0.1s$ apart, i.e., there are 10 events per second per process. For our verification step, our monitoring algorithm executes on the generated computation and verifies it against an MTL specification. We consider the following parameters (1) primary which includes time synchronization constant (ϵ), (2) MTL formula under monitoring, (3) number of segments (g), (4) computation length (l), (5) number of processes in the system ($\mathcal{P}$), (6) event rate and (7) offset distribution. We study the runtime of our monitoring algorithm against each of these parameters. We use a machine with $2x$ Intel Xeon Platinum 8180 (2.5 Ghz) processor, 768 GB of RAM, 112 vcores with gcc version 9.3.1.

Analysis : Runtime

We study each of the parameters individually and analyze how it effects the runtime of our monitoring approach. All results correspond to $\epsilon = 15ms$, $|\mathcal{P}| = 2$, $g = 15$, $l = 2sec$, an event rate of $10events/sec$, φ_4 as the MTL specification and U_1 as the offset distribution unless mentioned otherwise. We vary the number of processes in the system from 2 to 4, since in most cross-chain transactions the number of blockchains involved is small.

Impact of different formula. Fig. 4.10a shows that runtime of the monitor depends on two factors: the number of sub-formulas and the depth of nested temporal operators. Comparing φ_3 and φ_6, both of which consists of the same number of predicates but since φ_6 has recursive temporal operators, it takes more time to verify and the runtime is comparable to φ_1, which consists of two sub-formulas. This is because verification of the inner temporal formula often requires observing states in the next segment in order to come to the final verdict. This accounts for more runtime for the monitor.

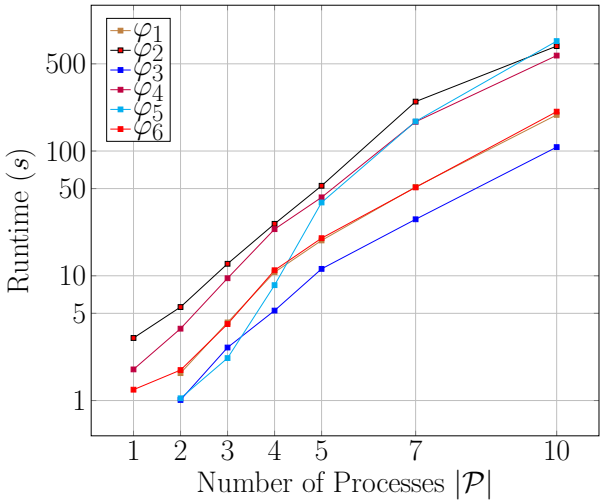
Impact of epsilon. Increasing the value of time synchronization constant (ϵ), increases the possible number of concurrent events that needs to be considered. This increases the complexity of verifying the computation and there-by increasing the runtime of the algorithm. In addition to this, higher values of ϵ also correspond to more number of possible traces that

are possible and should be taken into consideration. We observe that the runtime increases exponentially with increasing the value of time synchronization constant in Fig. 4.10b. An interesting observation is that, with longer segment length, the runtime increases at a higher rate than with shorter segment length. This is because with longer segment length and higher ϵ, it equates to a larger number of possible traces that the monitoring algorithm needs to take into consideration. This increases the overall runtime of the verification algorithm by a considerable amount and at a higher pace.

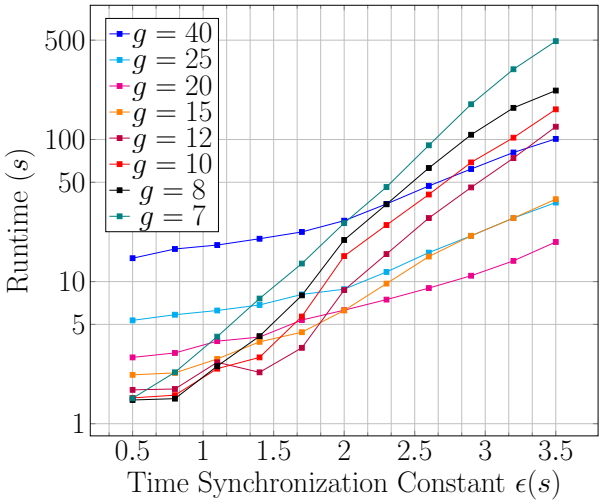
Impact of segment frequency. Increasing the segment frequency makes the length of each segment lower and thus verifying each segment involves a lower number of events. We observe the effect of segment frequency on the runtime of our verification algorithm in Fig. 4.10c. With increasing the segment frequency, the runtime decreases unless it reaches a certain value (here it is ≈ 0.6) after which the benefit of working with a lower number of events is overcast by the time required to setup each SMT instances. Working with higher number of segments equates to solving more number of SMT problem for the same computation length. Setting up the SMT problem requires a considerable amount of time which is seen by the slight increase in runtime for higher values of segment frequency.

Impact of computation length. As it can be inferred from the previous results, the runtime of our verification algorithm is majorly dictated by the number of events in the computation. Thus, when working with a longer computation, keeping the maximum clock skew and the number of segments constant, we should see a longer verification time as well. Results in Fig. 4.10d supports the above claim.

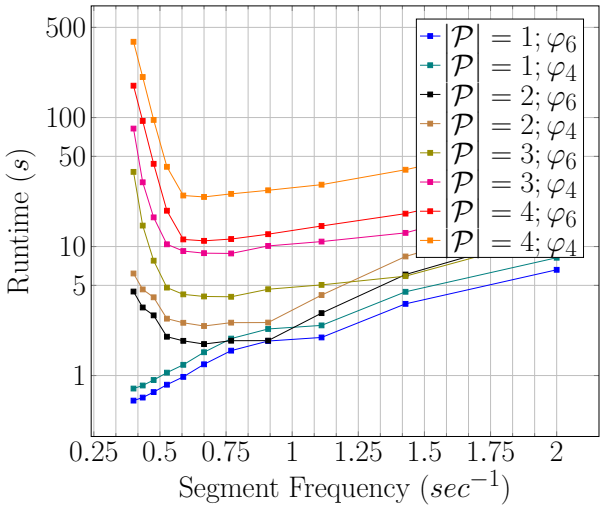
Impact of number of truth values per segment. In order to take into consideration all possible truth values of a computation, we execute the SMT problem multiple times, with the verdict of all previous executions being added to the SMT problem such that no two verdict is repeated. Here in Fig. 4.10e we see that the runtime is linearly effected by increasing number of distinct verdicts. This is because, the complexity of the problem that



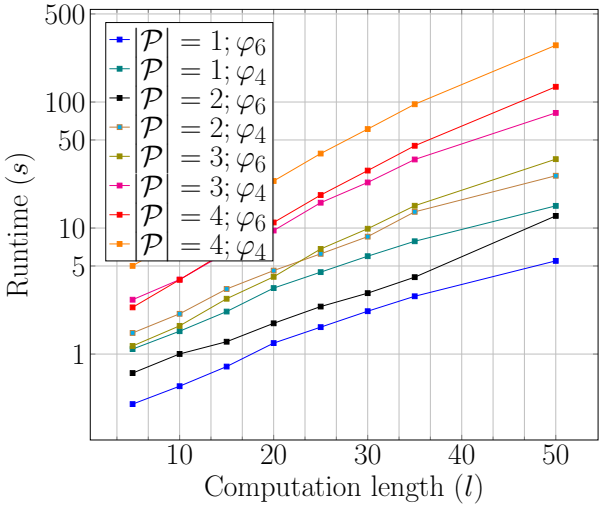
(a) Different Formula



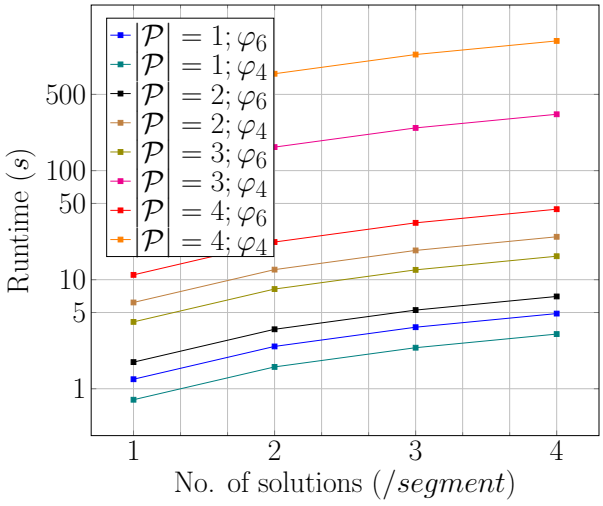
(b) Epsilon



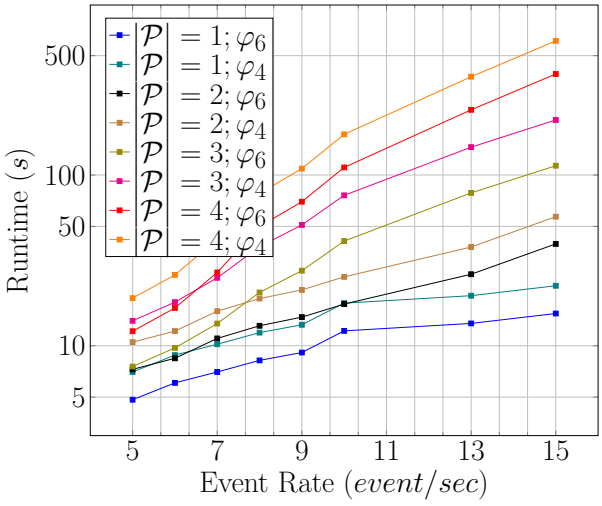
(c) Segment Frequency



(d) Computation Length

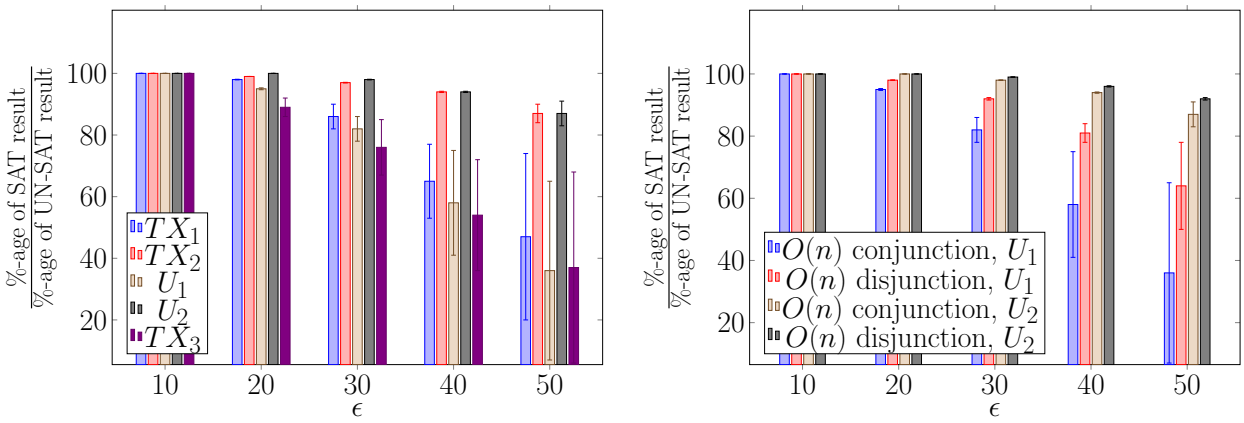


(e) Number of Process



(f) Event Rate

Figure 4.10: Different parameter's impact on runtime for synthetic data.



(a) Epsilon (b) Predicate Structure

Figure 4.11: Different parameter's impact on statistical guarantee for synthetic data.

the SMT is trying to solve does not change when trying to evaluate to a different solution.

Impact of event-rate. Increasing the event rate involves more number of events that needs to be processed by our verification algorithm per segment and thereby increasing the runtime at an exponential rate as seen in Fig. 4.10f. We also observe that with higher number of processes, the rate at which the runtime of our algorithm increases is higher for the same increase in event rate.

Analysis : Statistical Guarantee

Next, we study the effect of different parameters on the statistical guarantee of the verdict computed by the monitor. All results correspond to $\epsilon = 20ms$, $|\mathcal{P}| = 2$, $g = 15$, $l = 2sec$, an event rate of $10events/sec$ and φ_4 as the `MTL` specification unless mentioned otherwise.

Impact of epsilon. As can be imagined, impact of larger clock skew has an negative impact on the verification result of the system. Larger clock skew leads to more number of events being considered as concurrent and that leads to more number of traces that is possible with the correct order of events being compromised. This leads to a lower statistical guarantee associated with system with larger ϵ. In our case, as seen in Figure 4.11a, we receive perfect score when $\epsilon = 10ms$, since this makes all the event perfectly ordered. Moreover, the

guarantee slides uniformly with increasing value of ϵ.

The other observation from Figure 4.11a is how the guarantee is effected by the different offset distribution. The less is the standard deviation of the distribution, the closer to the global clock is the time of occurrence of the event. This makes the statistical guarantee of yielding a satisfiable result more than compared to when the time of occurrence is far from the global clock. Thus TX_2 yields higher percentage of satisfiable result when compared to TX_1, which has a larger standard deviation.

Impact of type of logical operator. Here, we compare how the the type of logical operator effects the probabilistic guarantee of a verdict. As can be seen in Figure 4.11b, formulas separated with a disjunction has a higher probabilistic percentage than the formulas separated by a conjunction. This can be explained by how a formula separated with conjunction is evaluated compared to the one with disjunction. In case of disjunction, any one sub-formula evaluated to be *true* rewrites the entire formula to be *true*, where-as in case of conjunction, all the sub-formulas need to be evaluated to be *true* and only then we come to a verdict of *true*. This marks the satisfiability percentage difference between the formulas separated by conjunction and disjunction.

4.4.2 Blockchain

Setup

We implemented the following cross-chain protocols from [XH21]: two-party swap, multi-party swap, and auction. The protocols are written as smart contracts in Solidity and tested using Ganache, a tool that creates mocked Ethereum blockchains. Using a single mocked chain, we mimicked cross-chain protocols via several (discrete) tokens and smart contracts, which do not communicate with each other.

Two Party Swap Protocol: We use the hedged two-party swap example from [XH21] to describe our experiments. The implementation of the other two protocols are similar. Suppose Alice would like to exchange her apricot tokens with Bob's banana tokens, using

the hedged two-party swap protocol shown in Fig. 4.1. This protocol provides protection for parties compared to a standard two-party swap protocol [Nol13], in that if one party locks their assets to exchange which is refunded later, this party gets a premium as compensation for locking their assets. The protocol consists of six steps to be executed by Alice and Bob in turn. In our example, we let the amount of tokens they are exchanging be 100 ERC20 tokens and the premium p_b be 1 token and $p_a + p_b$ be 2 tokens. We deploy two contracts on both apricot blockchain(the contract is denoted as *ApricotSwap*) and banana blockchain (denoted as *BananaSwap*) by mimicking the two blockchains on Ethereum. Denote the time that they reach an agreement of the swap as $startTime$. Δ is the maximum time for parties to observe the state change of contracts by others and take a step to make changes on contracts. In our experiment, $\Delta = 500$ milliseconds. By the definition of the protocol, the execution should be:

- Step 1. Alice deposits 2 tokens as premium in *BananaSwap* before Δ elapses after $startTime$.
- Step 2. Bob should deposit 1 token as premium in *ApricotSwap* before 2Δ elapses after $startTime$.
- Step 3. Alice escrows her 100 ERC20 tokens to *ApricotSwap* before 3Δ elapses after $startTime$.
- Step 4. Bob escrows her 100 ERC20 tokens to *BananaSwap* before 4Δ elapses after $startTime$.
- Step 5. Alice sends the preimage of the hashlock to *BananaSwap* to redeem Bob's 100 tokens before 5Δ elapses after $startTime$. Premium is refunded.
- Step 6. Bob sends the preimage of the hashlock to *ApricotSwap* to redeem Alice's 100 tokens before 6Δ elapses after $startTime$. Premium is refunded.

If all parties are conforming, the protocol is executed as above. Otherwise, some asset refund and premium redeem events is triggered to resolve the case where some party deviates. To avoid distraction, we do not provide details here.

Each smart contract provides functions to let parties deposit premiums *DepositPremium()*, escrow an asset *EscrowAsset()*, send a secret to redeem assets *RedeemAsset()*, refund the asset if it is not redeemed after timeout, *RefundAsset()*, and counterparts for premiums *RedeemPremium()* and *RefundPremium()*. Whenever a function is called successfully (meaning the transaction sent to the blockchain is included in a block), the blockchain emits an event that we then capture and log. The event interface is provided by the Solidity language. For example, when a party successfully calls *DepositPremium()*, the *PremiumDeposited* event emits on the blockchain. We then capture and log this event, allowing us to view the values of *PremiumDeposited*'s declared fields: the time when it emits, the party that initiated *DepositPremium()*, and the amount of premium sent. Those values are later used in the monitor to check against the specification.

Three Party Swap Protocol: The three-party swap example we implemented can be described as a digraph where there are directed edges between Alice, Bob and Carol. For simplicity, we consider each party transfers 100 assets. Transfer between Alice and Bob is called *ApricotSwap*, meaning Alice proposes to transfer 100 apricot tokens to Bob, transfer between Bob and Carol called *BananaSwap*, meaning Bob proposes to transfer 100 banana tokens to Carol, transfer between Carol and Alice, called *CherrySwap*, meaning Carol proposes to transfer 100 cherry tokens to Alice. Different tokens are managed by different blockchains (Apricot, Banana and Cherry respectively).

We denote the time they reach an agreement of the swap as $startTime$. Δ is the maximum time for parties to observe the state change of contracts by others and take a step to make changes on contracts. According of the protocol, the execution should follow the following steps:

- Step 1. Alice deposits 3 tokens as $escrow\_premium$ in *ApricotSwap* before Δ elapses after $startTime$.
- Step 2. Bob deposits 3 tokens as $escrow\_premium$ in *BananaSwap* before 2Δ elapses after $startTime$.

- Step 3. Carol deposits 3 tokens as *escrow_premium* in *CherrySwap* before 3Δ elapses after *startTime*.
- Step 4. Alice deposits 3 tokens as *redemption_premium* in *CherrySwap* before 4Δ elapses after *startTime*.
- Step 5. Carol deposits 2 tokens as *redemption_premium* in *BananaSwap* before 5Δ elapses after *startTime* .
- Step 6. Bob deposits 1 token as *redemption_premium* in *ApricotSwap* before 6Δ elapses after *startTime*.
- Step 7. Alice escrows 100 ERC20 tokens to *ApricotSwap* before 7Δ elapses after *startTime*.
- Step 8. Bob escrows 100 ERC20 tokens to *BananaSwap* before 8Δ elapses after *startTime*.
- Step 9. Carol escrows 100 ERC20 tokens to *CherrySwap* before 9Δ elapses after *startTime*.
- Step 10. Alice sends the preimage of the hashlock to *CherrySwap* to redeem Carol's 100 tokens before 10Δ elapses after *startTime*.
- Step 11. Carol sends the preimage of the hashlock to *BananaSwap* to redeem Bob's 100 tokens before 11Δ elapses after *startTime*.
- Step 12. Bob sends the preimage of the hashlock to *ApricotSwap* to redeem Alice's 100 tokens before 12Δ elapses after *startTime*.

If all parties are conforming, the protocol is executed as above. Otherwise, some asset refund and premium redeem events will be triggered to resolve the case where some party deviates. To avoid distraction, we do not provide details here.

Liveness: A liveness property of a program is that it asserts that something good eventually happens. In other words, a liveness property describes something that must happen during an execution. Below shows the specification to liveness, i.e., if all the steps of the protocol

has been taken:

$$\varphi_{liveness} = \Diamond_{[0,\Delta)}\, apr.depositEscrowPr(alice) \wedge \Diamond_{[0,2\Delta)}\, ban.depositEscrowPr(bob)$$

$$\wedge \Diamond_{[0,3\Delta)}\, che.depositEscrowPr(carol) \wedge \Diamond_{[0,4\Delta)}\, che.depositRedemptionPr(alice)$$

$$\wedge \Diamond_{[0,5\Delta)}\, ban.depositRedemptionPr(carol) \wedge \Diamond_{[0,6\Delta)}\, apr.depositRedemptionPr(bob)$$

$$\wedge \Diamond_{[0,7\Delta)}\, apr.assetEscrowed(alice) \wedge \Diamond_{[0,8\Delta)}\, ban.assetEscrowed(bob)$$

$$\wedge \Diamond_{[0,9\Delta)}\, che.assetEscrowed(carol) \wedge \Diamond_{[0,10\Delta)}\, che.hashlockUnlocked(alice)$$

$$\wedge \Diamond_{[0,11\Delta)}\, ban.hashlockUnlocked(carol) \wedge \Diamond_{[0,12\Delta)}\, apr.hashlockUnlocked(bob)$$

$$\wedge \Diamond\, assetRedeemed(alice) \wedge \Diamond\, assetRedeemed(bob) \wedge \Diamond\, assetRedeemed(carol)$$

$$\wedge \Diamond\, EscrowPremiumRefunded(alice) \wedge \Diamond\, EscrowPremiumRefunded(bob)$$

$$\wedge \Diamond\, EscrowPremiumRefunded(carol)$$

$$\wedge \Diamond\, RedemptionPremiumRefunded(alice)$$

$$\wedge \Diamond\, RedemptionPremiumRefunded(bob)$$

$$\wedge \Diamond\, RedemptionPremiumRefunded(carol)$$

Safety A safety property of a program is that it asserts that nothing bad happens during execution. In other words, a safety property describes something that must not happen during an execution. Below shows the specification to check if an individual party is conforming. If a party is found to be conforming we ensure that there is no negative payoff

for the corresponding party. Specification to check Alice is conforming:

$$
\begin{aligned}
&\varphi_{alice\_conf} = \Diamond_{[0,\Delta)}\ apr.depositEscrowPr(alice) \wedge \\
&\big(\Diamond_{[0,3\Delta)}\ che.depositEscrowPr(carol) \rightarrow \Diamond_{[0,4\Delta)}\ che.depositRedemptionPr(alice)\big) \wedge \\
&\big(\neg che.depositRedemptionPr(alice)\ \mathcal{U}\ che.depositEscrowPr(carol)\big) \wedge \\
&\big(\Diamond_{[0,6\Delta)}\ apr.depositRedemptionPr(bob) \rightarrow \Diamond_{[0,7\Delta)}\ apr.assetEscrowed(alice)\big) \wedge \\
&\big(\neg apr.assetEscrowed(alice)\ \mathcal{U}\ apr.depositRedemptionPr(bob)\big) \wedge \\
&\big(\Diamond_{[0,9\Delta)}\ che.assetEscrowed(carol) \rightarrow \Diamond_{[0,10\Delta)}\ che.hashlockUnlocked(alice)\big) \wedge \\
&\big(\neg che.hashlockUnlocked(alice)\ \mathcal{U}\ che.assetEscrowed(carol)\big) \wedge \\
&\big(\neg ban.hashlockUnlocked(carol)\ \mathcal{U}\ che.hashlockUnlocked(alice)\big) \wedge \\
&\big(\neg apr.hashlockUnlocked(bob)\ \mathcal{U}\ che.hashlockUnlocked(alice)\big)
\end{aligned}
$$

Specification to check conforming Alice does not have a negative payoff:

$$\varphi_{alice\_safety} = \varphi_{alice\_conform} \rightarrow \Big(\sum_{TransTo\ =\ alice} amount \quad \geq \sum_{TransFrom\ =\ alice} amount \Big)$$

Hedged Below shows the specification to check that, if a party is conforming and its escrowed asset is refunded, then it gets a premium as compensation.

$$
\begin{aligned}
\varphi_{alice\_hedged} = &\Diamond \big(\varphi_{alice\_conform} \wedge apr.assetEscrowed(alice)\big) \rightarrow \\
\Diamond \Big(\sum_{TransTo\ =\ alice} amount \geq &\sum_{TransFrom\ =\ alice} amount + apr.redemptionPremium.amount \Big)
\end{aligned}
$$

Auction Protocol: In the auction example, we consider Alice to be the auctioneer who would like to sell a ticket (worth 100 ERC20 tokens) on the ticket (*tckt*) blockchain, and Bob and Carol bid on the *coin* blockchain and the winner should get the ticket and pay for the auctioneer what they bid, and the loser will get refunded. We denote the time that they reach an agreement of the auction as $startTime$. Δ is the maximum time for parties to observe the state change of contracts by others and take a step to make changes on

contracts. Let $TicketAuction$ be a contract managing the "ticket" on the ticket blockchain, and $CoinAuction$ be a contract managing the bids on the coin blockchain. The protocol is briefed as follows.

- Setup. Alice generates two hashes $h(s_b)$ and $h(s_c)$. $h(s_b)$ is assigned to Bob and $h(s_c)$ is assigned to Carol. If Bob is the winner, then Alice releases s_b. If Carol is the winner, then Alice releases s_c. If both s_b and s_c are released in $TicketAuction$, then the ticket is refunded. If both s_b and s_c are released in $CoinAuction$, then all coins are refunded. In addition, Alice escrows her ticket as 100 ERC20 tokens in $TicketAuction$ and deposits 2 tokens as premiums in $CoinAuction$.
- Step 1 (Bidding). Bob and Carol bids before Δ elapses after $startTime$.
- Step 2 (Declaration). Alice sends the winner's secret to both chains to declare a winner before 2Δ elapses after $startTime$.
- Step 3 (Challenge). Bob and Carol challenges if they see two secrets or one secret missing, i.e. Alice cheats, before 4Δ elapses after $startTime$. They challenge by forwarding the secret released by Alice using a path signature scheme [Her18].
- Step 4 (Settle). After 4Δ elapses after $startTime$, on the $CoinAuction$, if only the hashlock corresponding to the actual winner is unlocked, then the winner's bid goes to Alice. Otherwise, the winner's bid is refunded. Loser's bid is always refunded. If the winner's bid is refunded, all bidders including the loser gets 1 token as premium to compensate them. On the $TicketAuction$, if only one secret is released, then the ticket is transferred to the corresponding party who is assigned the hash of the secret. Otherwise, the ticket is refunded.

Liveness A liveness property of a program is that it asserts that something good eventually happens. In other words, a liveness property describes something that must happen during an execution. Below shows the specification to check that, if all parties are conforming, the

winner (Bob) gets the ticket and the auctioneer gets the winner's bid.

$$\begin{aligned}\varphi_{liveness} = \Diamond_{[0,\Delta)}\, coin.bid(bob) \wedge \Diamond_{[0,2\Delta)}\, coin.declaration(alice,\ s_b) \wedge \\ \Diamond_{[0,2\Delta)}\, tckt.declaration(alice,\ s_b) \wedge \Diamond_{(4\Delta,\infty)}\, coin.redeemBid(any) \wedge \\ \Diamond_{(4\Delta,\infty)}\, coin.refundPremium(any) \wedge \big(coin.bid(carol) \rightarrow \\ \Diamond_{[0,\Delta)}\, coin.refundBid(any)\big) \wedge tckt.redeemTicket(any) \wedge \\ \neg coin.challenge(any) \wedge \neg tckt.challenge(any)\end{aligned}$$

Safety A safety property of a program is that it asserts that nothing bad happens during execution. In other words, a safety property describes something that must not happen during an execution. Below shows the specification to check that, if a party is conforming, this party does not end up worse off. Take Bob (the winner) for example.

Specification to define Bob is conforming:

$$\begin{aligned}\varphi_{bob\_conform} = \Diamond_{[0,\Delta)}\, coin.bid(bob) \\ \wedge \Big(\big(coin.declaration(alice,\ s_c) \vee coin.challenge(carol,\ s_c)\big) \rightarrow \\ \wedge \big(tckt.declaration(alice,\ s_c) \vee tckt.challenge(carol,\ s_c) \vee \\ tckt.challenge(bob,\ s_c)\big)\Big) \wedge \Big(\big(coin.declaration(alice,\ s_b) \vee \\ coin.challenge(carol,\ s_b)\big) \rightarrow \wedge\big(tckt.declaration(alice,\ s_b) \vee \\ tckt.challenge(carol,\ s_b) \vee tckt.challenge(bob,\ s_b)\big)\Big) \\ \wedge \Big(\big(tckt.declaration(alice,\ s_c) \vee tckt.challenge(carol,\ s_c)\big) \rightarrow \\ \big(coin.declaration(alice,\ s_c) \vee coin.challenge(carol,\ s_c) \vee \\ coin.challenge(bob,\ s_c)\big)\Big) \wedge \\ \Big(\big(tckt.declaration(alice,\ s_b) \vee \\ tckt.challenge(carol,\ s_b)\big) \rightarrow \big(coin.declaration(alice,\ s_b) \vee \\ coin.challenge(carol,\ s_b) \vee coin.challenge(bob,\ s_b)\big)\Big)\end{aligned}$$

Specification to define Bob does not end up worse off:

$$\varphi_{bob\_safety} = \varphi_{bob\_conform} \rightarrow \Diamond \Big(\big(coin.refundBid(any) \\ \wedge\ coin.redeemPremium(any) \big) \vee tckt.redeemTicket(any) \Big)$$

Hedged Below shows the specification to check that, if a party is conforming and its escrowed asset is refunded, then it gets a premium as compensation.

$$\varphi_{bob\_hedged} = \Box \Big(\varphi_{bob\_conforming} \wedge \\ \big(tckt.refundTicket(alice) \vee tckt.redeemTicket(carol) \big) \Big) \rightarrow \\ \Diamond \big(coin.refundBid(any) \wedge coin.redeemPremium(any) \big)$$

Log Generation and Monitoring

Our tests simulates different executions of the protocols and generated 1024, 4096, and 3888 different sets of logs for the aforementioned protocols, respectively. We again use the hedged two-party swap as an example to show how we generate different logs to simulate different execution of the protocol. On each contract, we enforce the order of those steps to be executed. For example, step 3 *EscrowAsset()* on the *ApricotSwap* cannot be executed before Step 1 is taken, i.e. the premium is deposited. This enforcement in the contract restricts the number of possible different states in the contract. Assume we use a binary indicator to denote whether a step is attempted by the corresponding party. 1 denotes a step is attempted, and 0 denotes this step is skipped. If the previous step is skipped, then the later step does not need to be attempted since it will be rejected by the contract. We use an array to denote whether each step is taken for each contract. On each contract, the different executions of those steps can be [1,1,1] meaning all steps are attempted, or [1,1,0] meaning the last step is skipped, and so on. Each chain has 4 different executions. We take the Cartesian product of arrays of two contracts to simulate different combinations of executions on two contracts. Furthermore, if a step is attempted, we also simulate whether

the step is taken late, or in time. Thus we have 2^6 possibilities of those 6 steps. In summary, we succeeded generating $4 \cdot 4 \cdot 2^6 = 1024$ different logs.

In our testing, after deploying the two contracts, we iterate over a 2D array of size 1024×12, and each time takes one possible execution denoted as an array length of 12 to simulate the behavior of participants. For example, $[1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]$ stands for the first step is attempted but it is late, and the steps after second step are all attempted in time. Indexed from 0, the even index denotes if a step is attempted or not and the odd index denotes the former step is attempted in time or late. By the indicator given by the array, we let parties attempt to call a function of the contract or just skip. In this way, we produce 1024 different logs containing the events emitted in each iteration.

We check the policies mentioned in [XH21]: liveness, safety, and ability to hedge against sore loser attacks. *Liveness* means that Alice should deposit her premium on the banana blockchain within Δ from when the swap started($\diamondsuit_{[0,\Delta)}$ *ban.premium_deposited(alice)*) and then Bob should deposit his premiums, and then they escrow their assets to exchange, redeem their assets (i.e. the assets are swapped), and the premiums are refunded. In our testing, we always call a function to settle all assets in the contract if the asset transfer is triggered by timeout. Thus, in the specification, we also check all assets are settled:

$$\begin{aligned}\varphi_{\mathsf{liveness}} = \diamondsuit_{[0,\Delta)}\, &\textit{ban.premium\_deposited(alice)} \wedge \diamondsuit_{[0,2\Delta)}\, \textit{apr.premium\_deposited(bob)} \wedge \\ &\diamondsuit_{[0,3\Delta)}\, \textit{apr.asset\_escrowed(alice)} \wedge \diamondsuit_{[0,4\Delta)}\, \textit{ban.asset\_escrowed(bob)} \wedge \\ &\diamondsuit_{[0,5\Delta)}\, \textit{ban.asset\_redeemed(alice)} \wedge \diamondsuit_{[0,6\Delta)}\, \textit{apr.asset\_redeemed(bob)} \wedge \\ &\diamondsuit_{[0,5\Delta)}\, \textit{ban.premium\_refunded(alice)} \wedge \diamondsuit_{[0,6\Delta)}\, \textit{apr.premium\_refunded(bob)} \wedge \\ &\diamondsuit_{[6\Delta,\infty)}\, \textit{apr.all\_asset\_settled(any)} \wedge \diamondsuit_{[5\Delta,\infty)}\, \textit{ban.all\_asset\_settled(any)}\end{aligned}$$

Safety is provided only for conforming parties, since if one party is deviating and behaving unreasonably, it is out of the scope of the protocol to protect them. Alice should always deposit her premium first to start the execution of the protocol($\diamondsuit_{[0,\Delta)}$ *ban.premium_deposited(alice)*) and proceed if Bob proceeds with the next step. For example, if Bob deposits his premium, then Alice should always

go ahead and escrow her asset to exchange($\Diamond_{[0,2\Delta)}$ *apr.premium_deposited(bob)* $\rightarrow$ $\Diamond_{[0,3\Delta)}$ *apr.asset_escrowed(alice)*). Alice should never release her secret if she does not redeem, which means Bob should not be able to redeem unless Alice redeems, which is expressed as $\neg$*apr.asset_redeemed(bob)* $\mathcal{U}$ *ban.asset_redeemed(alice)*:

$$\begin{aligned}\varphi_{\text{alice\_conform}} = \Diamond_{[0,\Delta)}\, &ban.premium\_deposited(alice) \wedge \\ &\big(\Diamond_{[0,2\Delta)}\, apr.premium\_deposited(bob) \rightarrow \Diamond_{[0,3\Delta)}\, apr.asset\_escrowed(alice)\big) \wedge \\ &\big(\Diamond_{[0,4\Delta)}\, ban.asset\_escrowed(bob) \rightarrow \Diamond_{[0,5\Delta)}\, ban.asset\_redeemed(alice)\big) \wedge \\ &\big(\neg apr.asset\_redeemed(bob)\ \mathcal{U}\ ban.asset\_redeemed(alice)\big)\end{aligned}$$

By definition, safety means a conforming party does not end up with a negative payoff. We track the assets transferred from parties and transferred to parties in our logs.

Thus, a conforming party is safe. e.g. Alice, is specified as safe φ_{alice_safety}:

$$\varphi_{\text{alice\_safety}} = \varphi_{\text{alice\_conform}} \rightarrow \Big(\sum_{TransTo\ =\ alice} amount \geq \sum_{TransFrom\ =\ alice} amount\Big)$$

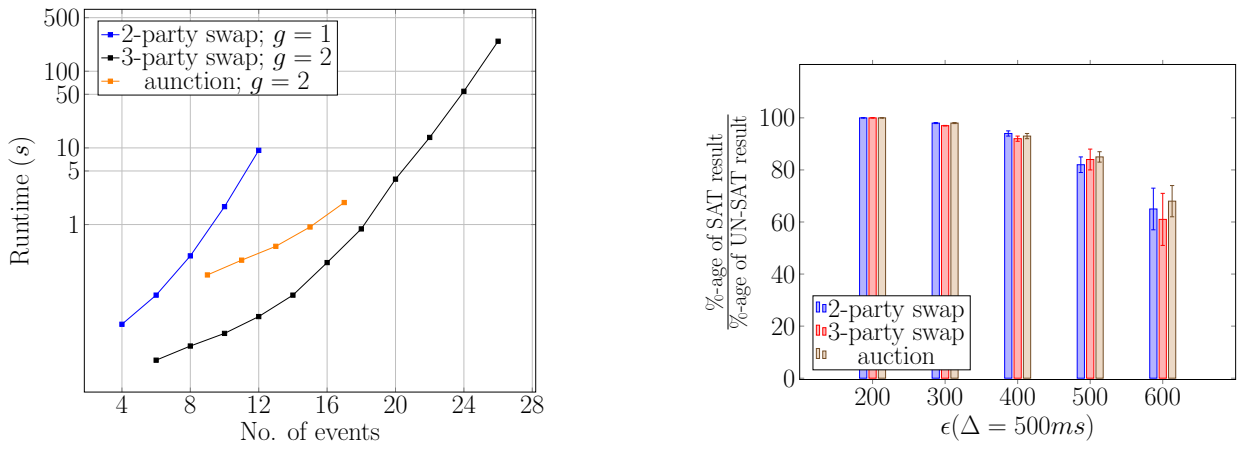
To enable a conforming party to hedge against the sore loser attack if they escrow assets to exchange which is refunded in the end, our protocol should guarantee the aforementioned party get a premium as compensation, which is expressed as $\varphi_{\text{alice\_hedged}}$:

$$\begin{aligned}\varphi_{\text{alice\_hedged}} = \Diamond\big(&\varphi_{\text{alice\_conform}} \wedge apr.asset\_escrowed(alice) \wedge apr.asset\_refunded(any)\big) \rightarrow \\ &\Diamond\Big(\sum_{TransferTo\ =\ alice} amount \geq \sum_{TransferFrom\ =\ alice} amount + apr.premium.amount\Big)\end{aligned}$$

Analysis of Results

We put our monitor to test the traces generated by the Truffle-Ganache framework. To monitor the 2-party swap protocol we do not divide the trace into multiple segments due to the low number of events that are involved in the protocol. On the other hand, both 3-party swap and auction protocol involve a higher number of events and thus we divide the trace into two segments ($g = 2$). In Fig. 4.12a, we show how the runtime of the monitor is effected by the number of events in each transaction log.

Additionally, we generate transaction logs with different values for deadline (Δ) and



(a) Runtime

(b) Statistical Guarantee

Figure 4.12: Results from the blockchain experiments.

time synchronization constant (ϵ) to put the safety of the protocol in jeopardy. We observe both *true* and *false* verdict when $\epsilon \gtrapprox \Delta$ as seen in Figure 4.12b. This is due to the non deterministic time stamp owning to the assumption of a partially synchronous system. The observed time stamp of each event can at most be off by ϵ. Thus, we recommend to use a value of Δ that is strictly greater than to the value of ϵ when designing the smart contract.

4.5 Summary and Limitation

In this chapter, we propose a monitoring technique which takes an `MTL` formula and a distributed computation as input. We apply a progression-based formula rewriting monitoring algorithm implemented as an SMT decision problem in order to verify the correctness of the distributed system with respect to the formula. We also conduct extensive synthetic experiments on traces generated by the tool UPPAAL and a set of blockchain smart contracts for cross-chain transactions.

However, as discussed in Section 4.4, the approach does not scale well when considering larger distributed system. Currently, the monitoring runtime increases exponentially with increase in the number of processes or events being monitored. This is a big limiting factor when designing a verification approach which can work in real time.

Chapter 5

Fault Tolerant Runtime Verification of Synchronous Distributed Systems

5.1 Introduction

In this chapter, we introduce an RV technique for fault-tolerant decentralized monitoring that inspects an underlying distributed system. Our RV framework has the following features:

- We assume that a set of monitors are distributed over a *synchronous* communication network. The network is a complete graph allowing all monitors to communicate with each other using point-to-point message passing in synchronous rounds.
- Each monitor is subject to *crash* failures. A crashed monitor halts permanently and never recovers.
- Each monitor has only a *partial view* of the underlying system. More specifically, given a set AP of atomic propositions that describe the global state of the system, each monitor reads only an arbitrary proper subset of AP.
- The formal specification language is the popular *linear temporal logic* (LTL) [MP79],

(To appear) Ritam Ganguly, Shokufeh Kazemloo, and Borzoo Bonakdarpour, Crash-Resilient Decentralized Synchronous Runtime Verification, IEEE Transactions on Dependable and Secure Computing.

where formulas are inductively constructed using the propositions in AP and operators that describe the temporal order of events.

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

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

Our main contribution in this chapter is a decentralized synchronous t-resilient RV algorithm, where t is the upper bound on the number of crash failures of monitors. Given a new global state, each monitor process computes a *symbolic* representation of its reading of AP and starts $t+1$ rounds of synchronous communication with other monitors in the network. The number of rounds is inspired by solutions to the consensus problem in synchronous networks, though in our problem, the monitors need to agree on a verdict that is not known a priori and they collaboratively compute the verdict during the rounds of communication. The symbolic representation is computed by employing a deterministic finite state automaton for monitoring formulas in the linear temporal logic (LTL). We show that the monitor automaton as constructed using the algorithm in [BLS11] cannot guarantee soundness in a distributed synchronous setting. Subsequently, we propose an algorithm that transforms the automaton into another by adding a minimum number of extra states and transitions to address cases where local monitors run into indistinguishable states due to their partial observations.

In order to minimize the size of the transformed automaton, we formulate an offline optimization problem in *satisfiability modulo theory* (SMT). The size of the SMT instance is expected to be small, as most practical LTL formulas are known to have at most just a few

nested temporal operators. Even if the size of the transformed monitor is not minimized the size of each message will be $O(\log(|\mathcal{M}_3^\varphi|)\cdot|\mathsf{AP}|)$, where $\mathcal{M}_3^\varphi$ denotes the finite state automaton for monitoring an LTL formula φ in the 3-valued semantics as constructed in [BLS11]. In short, our RV framework has message complexity

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

for evaluating each global state, where n is the number of distributed monitors and t is the bound on the number of crash failures. An important implication of our results is that unlike the asynchronous fault-prone setting, where we need to increase the number of truth values in the specification language to design consistent distributed monitors [FRT14, FRRT14, BFR+16], in this chapter, we show that in a fault-prone synchronous setting, the number of truth values is irrelevant for sound distributed monitoring.

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

We have implemented and evaluated our approach on a variety of LTL formulas for traces being generated using different random distributions as well as an IoT dataset, Orange4Home [CLRC17]. We analyze the average number of rounds and total messages being sent in the system for different values of t and l. We also analyze the change in the average number of rounds, total number of messages, average size of a message along with total monitor crashing in the system for different length of execution traces.

5.2 Model of Computation

An $\texttt{LTL}_3$ monitor as defined in Definition 3 can evaluate an LTL formula φ with respect to a finite execution, where each event represents the full view of the system under inspection. From now on, we refer to such events as *global events*, where the value of all propositions in the event is known. While this model is realistic in a centralized setting, it is too abstract in a distributed setting. We now present our computation model.

5.2.1 Overall Picture

We consider a distributed monitoring system comprising of a fixed number n of *monitor processes* $\mathcal{M} = \{M_1, M_2, \ldots, M_n\}$ that communicate with each other by sending and receiving messages through point-to-point bidirectional communication links (To prevent confusion, we refer to monitors in $\mathcal{M}$ as 'monitor processes' and the one defined in Definition 3 as '$\texttt{LTL}_3$ monitor'). We assume that the communication graph is synchronous and complete. Each communication link is reliable, that is, we assume no loss or alteration of messages. Each monitor process locally executes identical sequential algorithms. Each run of a monitor process consists of a sequence of *rounds* that are identified by the successive positive integers 1, 2, etc. The round number is a global variable and its progress is ensured by the synchrony assumption [Lyn96]. Each round is made up of three consecutive steps: *send*, *receive*, and *local computation*. The principle property of the round-based synchronous model is the fact that a message sent by a monitor M_i to another monitor M_j, for all $i, j \in [1, n]$, during a round r is received by M_j at the very same round r. Each monitor process can start a new round when the current is complete.

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

by-event approach in Section 5.4):

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

5.2.2 Detailed Description

We now delve into the details of our computation model (see Algorithm 5). When an event s_j is reached in a finite trace $\alpha = s_0 s_1 \cdots s_k$, each monitor process $M_i \in \mathcal{M}$, where $i \in [1, n]$, attempts to read s_j (Line 2 in Algorithm 5). Due to distribution, this results in obtaining a partial view $\mathcal{S}_i^{s_j}$ defined next.

Definition 9. *A* partial view *is a function* $\mathcal{S} : \mathsf{AP} \mapsto \{\mathit{true}, \mathit{false}, \natural\}$*, i.e, a mapping from the set of atomic propositions to values true, false, or* $\natural$*. The latter denotes an* unknown *value for a proposition.* □

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

Definition 10. *We say that a partial view* $\mathcal{S}$ *is* consistent *with a global event* $s \in \Sigma$ *(denoted* $\mathcal{S} \sqsubseteq s$*), if for every atomic proposition* $\mathsf{p} \in \mathsf{AP}$*, we have:*

$$\big(\mathcal{S}(\mathsf{p}) = \mathit{true} \;\Leftrightarrow\; \mathsf{p} \in s\big) \;\wedge\; \big(\mathcal{S}(\mathsf{p}) = \mathit{false} \;\Leftrightarrow\; \mathsf{p} \notin s\big).$$ □

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

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

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

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

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

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

In Algorithm 5, one way for monitor processes to share their observation of the system is to communicate their partial views. This way, after several rounds of communication (due to the occurrence of faults), all monitor processes can construct the full global event. Although this idea works in principle, it is quite inefficient, as the size of each message will have to be at least $|\mathsf{AP}|$ bits. Our goal is to design a technique, where monitor processes can communicate their observations without sending and receiving their partial views of atomic propositions. To this end, we introduce the notion of a *symbolic view* that intends to represent the partial view of a monitor processes M_i without losing information. We denote the symbolic view of a partial view $\mathcal{S}_i^s$ with respect to an LTL formula φ by $LS_i = \mu(\mathcal{S}_i^s, \varphi)$ (see Line 3 in Algorithm 5). In Section 5.3, we will present a concrete way of computing μ.

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

and then receives the symbolic view of all monitor processes in Line 6. Let $\Pi_i^r = \{LS_l^r\}_{l\in[1,n]}$ be the set of all messages received (We note that if some monitor process crashes while another monitor is receiving messages in Line 6, this monitor will not receive n messages as prescribed by the algorithm. In synchronous algorithms, by the synchrony assumption, a crash failure can be easily detected and hence, the accurate value of n can be determined for receiving messages) by monitor process M_i during round r. Then (Line 7), the monitor computes the new symbolic view from the messages it received using a function LC (described in detail in Section 5.3). This new view will be broadcast during the next round.

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

Definition 11. *A set* $\mathcal{M} = \{M_1, M_2, \ldots, M_n\}$ *satisfies* event coverage *for an event* s *if and only if for every* $\mathsf{p} \in \mathsf{AP}$*, there exists* $M_i \in \mathcal{M}$ *such that* $\mathcal{S}_i^s(\mathsf{p}) \neq \natural$. □

5.2.3 Fault Model

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

5.2.4 Problem Statement

Our formal problem statement is the termination requirement for Algorithm 5. We require that when a non-faulty monitor process runs Algorithm 5 to the end, it emits a

verdict that a centralized monitor that has global view of the system would compute:

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

where $\alpha \in \Sigma^*$, φ is an LTL formula, and ν_i is the truth value emitted by monitor M_i at the end of Algorithm 5.

It is easy to see that our decentralized synchronous monitoring problem, where monitor processes are subject to crash faults is in spirit similar to the uniform *consensus* problem [Lyn96]. The main difference is that in consensus, processes need to agree on one values that they own. In our problem, they should agree on the value $[\alpha \models_3 \varphi]$, while none of the monitors necessarily has this value before the inner for-loop. In Section 5.4, we will show that similar to synchronous consensus, if t monitors may fail, $t+1$ rounds of communication are sufficient to agree on the final verdict.

5.3 The General Idea and Motivating Example

In Algorithm 5, we provided the skeleton of our synchronous monitoring algorithm. What remains to be done is identifying concrete functions μ and LC. Our general idea is described in the sequel and is reflected in Algorithm 6, which refines Algorithm 5.

5.3.1 Symbolic View μ

As mentioned in Section 5.2, sharing explicit partial views is not space efficient, as each message will need at least $|\mathsf{AP}|$ bits. To tackle this problem, our idea is that each monitor process employs an $\mathtt{LTL_3}$ monitor, as defined in Definition 3 and the symbolic view of a monitor process consists of the set of *possible* $\mathtt{LTL_3}$ monitor states that corresponds to its partial view. Formally, let q be the current state of the $\mathtt{LTL_3}$ monitor and $\mathcal{S}$ be the partial view of the monitor process. The set of possible next $\mathtt{LTL_3}$ monitor states can be computed as follows:

$$\mu(\mathcal{S}, q) = \Big\{ q' \mid \exists s \in \Sigma. \big(\mathcal{S} \sqsubseteq s \wedge \delta(q, s) = q'\big) \Big\} \tag{5.1}$$

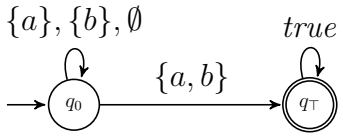


Figure 5.1: $\texttt{LTL}_3$ monitor for $\varphi = \Diamond(a \wedge b)$.

Recall that δ denotes the transition function in $\texttt{LTL}_3$ monitors. For example, consider the following LTL formula $\varphi = \Diamond(a \wedge b)$. The $\texttt{LTL}_3$ monitor of this formula is shown in Fig. 5.1, where $\lambda(q_0) =?$ and $\lambda(q_\top) = \top$. Let us imagine that (1) a monitor process M_1 is currently in state q_0, (2) the global event is $s = \{a, b\}$, and (3) the current partial view of M_1 is $\mathcal{S}_1^s(a) = \mathit{true}$ and $\mathcal{S}_1^s(b) = \mathit{true}$. This implies that monitor M_1 considers $q_\top$ as the only possible next $\texttt{LTL}_3$ monitor state, i.e., $\mu(\mathcal{S}_1^s, q_0) = \{q_\top\}$. However, considering another partial view $\mathcal{S}_1^s(a) = \mathit{true}$ and $\mathcal{S}_1^s(b) = \natural$, monitor process M_1 will have to consider $\{q_0, q_\top\}$ as possible next $\texttt{LTL}_3$ monitor states. This is because it has to consider two possibilities for proposition b. That is, $\mu(\mathcal{S}_1^s, q_0) = \{q_0, q_\top\}$. We use μ as defined in Equation (5.1) to compute the concrete symbolic view in Line 4 of Algorithm 6.

5.3.2 Computing *LC*

Given a set of possible $\texttt{LTL}_3$ monitor states computed by μ, in Line 7 of Algorithm 6, each monitor process receives a set of possible states from all other monitors, denoted by LS_i^r for each monitor process M_i, where $i \in [1, n]$ and each communication round r. Our idea to compute LC from these sets is to simply take their *intersection*. The intuition behind intersection is that it represents the conjunction of all partial views of all monitors. That is, in Line 8 of Algorithm 6, we have:

$$LC(\Pi_i^r) = \bigcap_{i \in [1,n]} LS_i^r. \tag{5.2}$$

Algorithm 6: Updated behavior of Monitor M_i, for $i \in [1, n]$.

Input $\mathtt{LTL_3}$ monitor $M_3^{\varphi} = \langle \Sigma, Q, q_0, \delta, \lambda \rangle$, finite trace $s_0 s_1 \cdots s_k$ **Output** Verdict from $\mathbb{B}_3$

1: $q_{current} \leftarrow q_0$
2: **for** $j = 0$ **to** k **do**
3: Let $\mathcal{S}_i^{s_j}$ be the initial partial view of the monitor
4: $LS_i^1 \leftarrow \mu(\mathcal{S}_i^{s_j}, q_{current})$ ▷ *Equation* (5.1)
5: ***for*** $r = 1$ **to** $t + 1$ ***do***
6: **Send:** *broadcasts symbolic view* LS_i^r
7: **Receive:** $\Pi_i^r \leftarrow \{LS_j^r\}_{j \in [1,n]}$
8: ***Computation:*** $LS_i^{r+1} \leftarrow LC(\Pi_i^r)$ ▷ Equation (5.2)
9: **end for**
10: $q_{current} \leftarrow LS_i^{r+1}$
11: **end for**
12: **return** $\lambda(q_{current})$

5.3.3 Motivating Example

The above general ideas for computing μ and LC has one problem. In Line 10, one final $\mathtt{LTL_3}$ monitor state should determine the final output, but in some cases, the partial views of two monitors are too coarse and applying intersection on them cannot compute the $\mathtt{LTL_3}$ monitor states that represent the aggregate knowledge of the monitors. For example, consider again the $\mathtt{LTL_3}$ monitor for formula $\Diamond(a \wedge b)$ in Fig. 5.1. Suppose that we have a global event $s = \{a, b\}$, two monitors M_1 and M_2, both at initial state q_0, and two partial views, where M_1 knows the value of a and M_2 knows the value of b. That is,

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

These monitors will compute μ as follows:

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

Applying intersection on $\mu(\mathcal{S}_1^s, q_0)$ and $\mu(\mathcal{S}_2^s, q_0)$ will result in the same set $\{q_0, q_\top\}$. At this point, no matter how many times the monitor processes communicate, at the end of the inner for-loop, LS will not become a singleton and in Line 11, $q_{current}$ cannot be determined properly. This scenario is in particular, problematic since the collective knowledge of M_1 and M_2 (i.e., the fact that a and b are both true) should result in reconstructing $s = \{a, b\}$.

Surprisingly, this problem does not stem from the way we compute μ and LC. It is mainly due to the structure of the $\texttt{LTL}_3$ monitor as defined in Definition 3. Although the definition works for centralized monitoring, it needs to be refined for distributed monitors that have only a partial view of the underlying system. In Section 5.4, we a technique to transform an $\texttt{LTL}_3$ monitor into an equivalent one capable of encoding enough information for monitor processes with partial views.

5.4 Monitor Transformation Algorithm

The discussion in Section 5.3 reveals the source of the problem on the structure of the monitor in Fig. 5.1. The self-loop on state q_0 prescribes that state q_0 is reachable by three events: $\{a\}$, $\{b\}$, or $\{\}$, while a partial view of $\{a, b\}$ may intersect with both $\{a\}$ and $\{b\}$, which are indistinguishable from each other. If we can somehow split q_0 to two states to explicitly distinguish the cases where either of a or b are true, then applying intersection will effectively solve the problem presented in Section 5.3.3. More specifically, consider the $\texttt{LTL}_3$ monitor shown in Fig. 5.2 for formula $\varphi = \Diamond(a \wedge b)$, where state q_0 is split in two states q_{01} and q_{02}. State q_{02} is reached when a is true and b is false. Analogously, State q_{01} is reached when b is true or both a and b are false. Now, recall the two monitors M_1 and M_2 and their partial views in Section 5.3.3:

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

These monitors will compute μ as follows:

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

Applying intersection on $\mu(\mathcal{S}_1^s, q_0)$ and $\mu(\mathcal{S}_2^s, q_0)$ will now result in the singleton $\{q_\top\}$, which is indeed the correct verdict for global event $\{a, b\}$. We call the monitor shown in Fig. 5.2

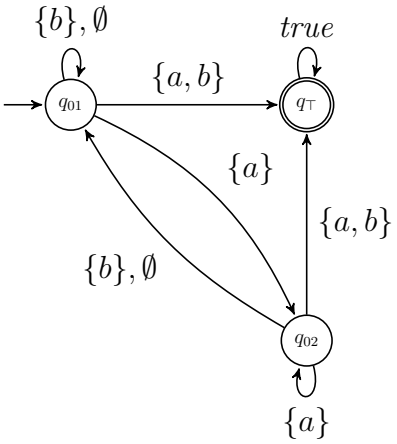


Figure 5.2: Extended $\mathtt{LTL_3}$ monitor for $\varphi = \Diamond(a \wedge b)$.

an *extended* $\mathtt{LTL_3}$ monitor.

In this section, we present an algorithm that takes as input an $\mathtt{LTL_3}$ monitor and generates as output an extended $\mathtt{LTL_3}$ monitor. We prove that by plugging an extended $\mathtt{LTL_3}$ monitor in the distributed RV Algorithm 6, it will produce a verdict identical to that of a centralized $\mathtt{LTL_3}$ monitor.

5.4.1 The Challenge of Constructing Extended Monitors

Let $\mathcal{M}_3^\varphi = \langle \Sigma, Q, q_0, \delta, \lambda \rangle$ be the $\mathtt{LTL_3}$ monitor of an $\mathtt{LTL}$ formula φ. To simplify our notation, we denote transitions of δ by:

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

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

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

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

Now, suppose that $\mathsf{AP} = \{a, b, c\}$, an $\mathtt{LTL_3}$ monitor has a transition of the form:

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

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

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

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

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

In scenarios (1) and (2) above: we further need to split q_{11} and q_{12}, respectively. But in scenario (3), there is no need to split q_{11} or q_{12}. Thus, the choice of splitting the monitors' blind spot, has an impact on the size of the extended $\mathtt{LTL_3}$ monitor. In order to minimize the number of new states that are added to the extended $\mathtt{LTL_3}$ monitor, we need to compute the minimum-size split. Finding the minimum-size split is a combinatorial optimization problem very similar to the set cover or the hitting set problems [GJ79]. In the next subsection, we present an SMT-based technique to obtain the minimum-size transition split.

5.4.2 Identifying the Minimum-size Split

Definition 12. *We say that a transition* $q \xrightarrow{\mathcal{L}} q'$ covers *an event* $s \in \Sigma$ *if and only if*

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

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

Definition 13. *We say that an event* s *is* opaque *to a transition* $q \xrightarrow{\mathcal{L}} q'$*, if (1)* $s \notin \mathcal{L}$*, but (2)* $q \xrightarrow{\mathcal{L}} q'$ *covers* s*.* □

For example, event $\{a, b\}$ is opaque to transition $q_0 \xrightarrow{\{a\},\{b\},\emptyset} q_\top$ in the $\mathtt{LTL_3}$ monitor in Fig. 5.1. It is easy to observe that two partial views of an opaque event to a transition may

Algorithm 7: Function to determine whether a transition has to split.

```
1: function SPLIT(𝓛)
2:   CV ← 0
3:   for each p ∈ AP  do
4:     if (∃s, s′ ∈ 𝓛.p ∈ s ∧ p ∉ s′) then
5:       CV ← CV + 1
6:     end if
7:   end for
8:   if (2^CV > |𝓛|) then
9:     return true
10:  end if
11:  return false
12: end function
```

result in identical possible sets of $\texttt{LTL}_3$ monitor states. When one monitor only reads a and another monitor reads only b, then the resulting set of possible states (i.e., $\{q_0, q_\top\}$) are not distinguishable from each other, because both propositions a and b are in event $\{a, b\}$. Indeed, this is the main reason in creating ambiguity in distributed monitor processes with partial views and such transitions need to be split in order to resolve possible ambiguities. Function *SPLIT* (see Algorithm 7) determines whether or not a transition should be split. The variable CV in the function computes the number of events covered by the input transition label set. In the above example, the value of 2^{CV} for transition $q_0 \xrightarrow{\{a\},\{b\},\emptyset} q_0$ is 4 which is strictly greater than $|\mathcal{L}| = 3$. This means that the transition needs to be split.

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

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

In other words, $\mathcal{L} = \mathcal{L}_1 \cup \mathcal{L}_2$ and $\mathcal{L}_1 \cap \mathcal{L}_2 = \emptyset$ such that we minimize the number of new states to be created.

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

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

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

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

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

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

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

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

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

$$w^{\mathsf{p}}_{\mathcal{L}_2} = \begin{cases} 0 & \text{if} \quad v^{\mathsf{p}}_{\mathcal{L}_2} = \mathit{true} \\ 1 & \text{otherwise} \end{cases}$$

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

$$x^{\mathcal{L}_2}_s = \neg x^{\mathcal{L}_1}_s$$

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

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

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

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

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

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

Optimization objective. Our objective is to minimize the total number of opaque events

to transition labels $\mathcal{L}_1$ and $\mathcal{L}_2$:

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

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

5.4.3 The Complete Transformation Algorithm

We now know how to split a transition to two transitions with minimum number of opaque events. All we need to do at this point is to design an algorithm that takes as input an $\mathtt{LTL_3}$ monitor $\mathcal{M}^{\varphi}_3 = \langle \Sigma, Q, q_0, \delta, \lambda \rangle$ and transforms it into an extended monitor $\mathcal{M}^{\varphi}_e = \langle \Sigma, Q_e, q^e_0, \delta_e, \lambda_e \rangle$ as output using the above SMT-based optimization technique. We now describe the details of this transformation in Algorithm 8:

- In Lines 2 – 29, we examine each outgoing transition of each state q of the input $\mathtt{LTL_3}$ monitor transitions for splitting.
- If a transition does not need to be split, we simply add the original transition to the extended monitor (Lines 26 and 27).
- For each transition that should be split, we apply the above SMT-based optimization technique described in Section 5.4.2. We first add the new states to the set of states of the extended monitor (Line 7). Then, we distinguish two cases:
  - If the transition that needs to be split, say $q \xrightarrow{\mathcal{L}} q'$ is *not* a self-loop (Lines 10 – 13), then two transitions $q \xrightarrow{\mathcal{L}_1} q_1$ and $q \xrightarrow{\mathcal{L}_2} q_2$ with the labels returned by the SMT-solver are included in the extended monitor (see Fig. 5.3). We also add all the outgoing transitions from q' to q_1 and q_2 (Line 13).
  - If the transition that needs to be split is a self-loop, say $q \xrightarrow{\mathcal{L}} q$, (Lines 15 – 20), then two transitions $q_1 \xrightarrow{\mathcal{L}_1} q_1$ and $q_1 \xrightarrow{\mathcal{L}_2} q_2$ with the labels returned by the SMT-solver are included in the extended monitor (see Fig. 5.4). We also add all

the outgoing transitions from q to q_1 and q_2 (Line 20) for the events not in the original self-loop.

Algorithm 8: Extended $\mathtt{LTL}_3$ Monitor Construction.

Input $\mathcal{M}_3^{\varphi} = \langle \Sigma, Q, q_0, \delta, \lambda \rangle$
Output $\mathcal{M}_e^{\varphi} = \langle \Sigma, Q_e, q_0^e, \delta_e, \lambda_e \rangle$

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

  - Finally, we include the incoming transitions to each state (Line 26) and remove labels that are have no opacity issues (Line 29).

- We repeat the loop until no transition needs to be split.

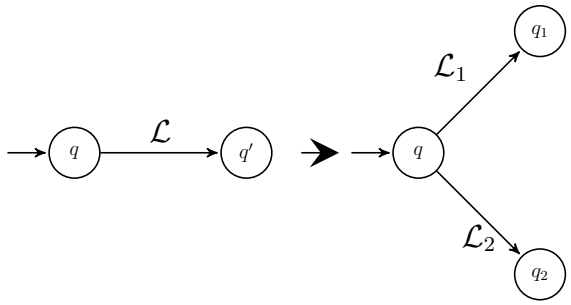


Figure 5.3: Splitting a transition to two.

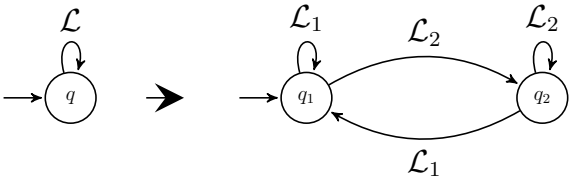


Figure 5.4: Splitting a self-loop to two.

The reader can test that running Algorithm 8 on the $\texttt{LTL}_3$ monitor in Fig 5.1, will result in the extended $\texttt{LTL}_3$ monitor in Fig. 5.2.

We now show the soundness of Algorithm 6 (as defined in the problem statement in Section 5.2.4) when augmented by an extended $\texttt{LTL}_3$ monitor as constructed by Algorithm 8.

Lemma 6. *For $\alpha \in \Sigma^*$ be a finite trace and φ be an LTL formula with $\mathcal{M}_3^{\varphi} = \langle \Sigma, Q, q_0, \delta, \lambda \rangle$ as the $\texttt{LTL}_3$ monitor. Using Algorithm 8 we get $\mathcal{M}^{\varphi_3} = \langle \Sigma, Q_e, q_0^e, \delta_e, \lambda_e \rangle$ such that $\lambda(\delta(q_0, \alpha)) = \lambda_e(\delta_e(q_0^e, \alpha))$*

Proof. Let $\alpha = s_0 s_1 \cdots s_n$. We prove that for some $i \in [0, n]$, $q \xrightarrow{s_i} q_1 \in \delta \Rightarrow q \xrightarrow{s_i} q_{01} \in \delta_e$ such that $\lambda(q_1) = \lambda(q_{01})$

Case 1: $q = q_1$

($\Rightarrow$) This means that q_1 was split into multiple state which includes q_{01}. As can be seen in Algorithm 8, lines 16-20, the states q' is split into q_1 and q_2, and the self-loop is preserved by having a loop within the states it was split into. Also in lines 22-24, all outgoing and incoming edges of q' is preserved with the label of q' being transferred to both q_1 and q_2 in line 29. Thus, $\lambda(q_1) = \lambda(q_{01})$

($\Leftarrow$) Trivial

Case 2: $q \neq q_1$

($\Rightarrow$) This means that q_1 was split into multiple state which includes q_{01}. As can be seen in Algorithm 8, lines 10-13, the states q' is split into q_1 and q_2, and the transitions are preserved by having a a transition from q to both q_1 and q_2 respectively. Also in lines 22-24,

all outgoing and incoming edges of q' is preserved with the label of q' being transferred to both q_1 and q_2 in line 29. Thus, $\lambda(q_1) = \lambda(q_{01})$

($\Leftarrow$) Trivial

Thus, $\lambda(\delta(q_0, \alpha)) = \lambda_e(\delta_e(q_0^e, \alpha))$ □

Lemma 7. *Let $\alpha \in \Sigma^*$ be a finite trace and φ be an* `LTL` *formula. The return value of Algorithm 6 augmented with an extended* $\texttt{LTL}_3$ *monitor as constructed in Algorithm 8 is $[\alpha \models_3 \varphi]$ by every monitor process in the presence of up to t crash failures.*

Proof. We prove Lemma 7 in three steps, similar to the proof technique for consensus in synchronous networks (e.g., the *FloodSet* algorithm) [Lyn96]. First, we prove that at the end of the inner for-loop, LS includes only one state. Then, we show that if no crash faults occur, in one round, all monitors will compute a monitor state q, where $\lambda(q)$ is the same as what a centralized monitor that can read the global event in one atomic step would compute. Finally, we show that if up to t monitors crash, all active monitors return $\lambda(q)$ as described in the previous step. We now delve into these three steps:

- **Step 1.** Let us assume that the monitor processes in $\mathcal{M}$ are evaluating event s_j for some $j \in [0, k]$. Formally, we are going to show that if no crash faults occur, then in Line 10 of Algorithm 6, we have $|LS_i^1| = 1$, for all $i \in [1, n]$. First, note that if no faults occur, all monitors send and receive all the messages in one clean round. Thus, in the subsequent rounds all messages will be identical. We now prove this claim by contradiction. Suppose we have $|LS_i^1| = 2$ (the case for > 2 can be trivially generalized). This means that at least two monitor processes sent a message containing two possible $\texttt{LTL}_3$ monitor states, say $\{q_1, q_2\}$. This can be due to two scenarios:
  - The first scenario is that q_1 and q_2 are possible $\texttt{LTL}_3$ monitor states, because the value of some atomic proposition $\mathsf{p} \in \mathsf{AP}$ is unknown, i.e., $\mathcal{S}(\mathsf{p}) = \natural$. However, this scenario contradicts our assumption on *event coverage* (see Section 5.2) in our computation model.
  - The second scenario is that q_1 and q_2 are possible $\texttt{LTL}_3$ monitor state, because s_j is opaque to some outgoing transitions from $q_{current}$ in the $\texttt{LTL}_3$ monitor. This case contradicts with our construction of extended $\texttt{LTL}_3$ monitor in Algorithm 8.
- **Step 2.** We prove this step by induction on the length of the finite input trace. The base case is that the monitors are evaluating event s_0 and $q_{current} = q_0$. From Step 1 of the proof, we know that $|LS_i^1| = 1$. We also know that $|LS_i^r| = 1$ (for all $r \in [1, t+1]$) and LS_i^r contains the same content as LS_i^1. Let this content be an $\texttt{LTL}_3$ monitor state q. Our goal is to show that:

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

  The proof, again, is by contradiction. This scenario can happen if the intersection of

all possible monitor states q, where $q = \delta(q_0, s_0)$ and $\lambda(q) \neq [s_0 \models \varphi]$. This can happen only if due to opacity, a wrong monitor state comes out of the intersection. This case contradicts with out construction of extended $\texttt{LTL}_3$ monitor in Algorithm 8. Hence, q would be the monitor state that a centralizes monitor would compute. The induction step is now trivial: it is straightforward to show that for any valid $q_{current}$ and any s_j, the next monitor state is the same as what a centralized monitor would compute.

- **Step 3.** From Steps 1 and 2, we know that if no faults occur, in one round all monitors compute one and only one $\texttt{LTL}_3$ monitor state q, where $\lambda(q) = [\alpha \models \varphi]$. Now, we show in a fault-prone scenario, in some round $1 \leq r \leq t+1$, any two active monitors M_i and M_j compute the same single monitor state $LS_i^r = \{q\}$, where $\lambda(q) = [\alpha \models \varphi]$. Since there are at most t crash failures, there has to be some round r, where no failures occur. Recall that in Section 5.2, we assume that if a monitor crashes and this monitor is the only one that is aware of some proposition $\mathsf{p} \in \mathsf{AP}$, this monitor sends a message containing its set of possible monitor states before crashing. This assumption ensures event coverage. This means that in any round $r \leq r' \leq t+1$, the value of all propositions are read. This in turn implies that all rounds r' are now identical to a fault-free setting and, hence, Steps 1 and 2 hold.

These three steps prove the soundness of Algorithm 6 when augmented by an extended $\texttt{LTL}_3$ monitor as constructed by Algorithm 8. □

We now extend our technique by monitors that evaluate a formula every $l \geq 1$ global states rather than after every global state. That is, the for-loop in Algorithm 6 iterates every $\lfloor k/l \rfloor$ and instead of a partial view $\mathcal{S}_i^{s_j}$ it evaluates a sequence of partial views $\mathcal{S}_i^{s_0}\mathcal{S}_i^{s_1}\ldots\mathcal{S}_i^{s_{l-1}}$ and so forth and, hence, the monitors communicate every l state (rather than every single state). To this end, let us recursively extend μ from a single partial view and a monitor state transition (i.e., $\mu(\mathcal{S}, q)$ as defined in Section 5.3) to a sequence of partial views $\mathcal{S}_i^{s_0}\mathcal{S}_i^{s_1}\ldots\mathcal{S}_i^{s_{l-1}}$ and a set of monitor states $Q' \subseteq Q$ as follows (denoted μ_l):

$$\mu_l(\mathcal{S}_i^{s_0}\mathcal{S}_i^{s_1}\ldots\mathcal{S}_i^{s_{l-1}}, Q') = \mu_1\Big(\mathcal{S}_{l-1}, \mu_{l-1}(\mathcal{S}_i^{s_0}\mathcal{S}_i^{s_1}\ldots\mathcal{S}_i^{s_{l-2}}, Q')\Big).$$

Theorem 1. *Let φ be an $\texttt{LTL}$ formula, $\alpha \in \Sigma^*$ with $|\alpha| = k$ and l a natural number, where $l \leq k$. Given the generalization of μ to μ_l, the output of Algorithm 6 is for μ_l is $[\alpha \models_3 \varphi]$.*

Proof. We prove the theorem by induction over l. The base case, (i.e., $l = 1$), trivially holds by Lemma 7. For the inductive step, let the statement of the theorem be true for l, meaning that the verdict of the algorithm is indeed for length l is the same as the verdict of

an $\texttt{LTL}_3$ monitor. We have to show that it also holds for $l+1$. This case is also discharged by Lemma 7, since state by state evaluation results in the correct $\texttt{LTL}_3$ evaluation. □

Theorem 2. *Let φ be an LTL formula and $\alpha \in \Sigma^*$ be a finite trace. The message complexity of Algorithm 6 using an extended $\texttt{LTL}_3$ monitor is*

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

where n is the number of distributed monitors.

Proof. We analyze the complexity of each part of Algorithm 6:

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

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

We now focus on the size of each message. Let $\mathcal{M}_3^\varphi = \langle \Sigma, Q, q_0, \delta, \lambda\rangle$ be an $\texttt{LTL}_3$ monitor and $\mathcal{M}_e^\varphi = \langle \Sigma, Q_e, q_0^e, \delta_e, \lambda_e\rangle$ be its extended monitor constructed by Algorithm 8. The algorithm may split a transition at most $|\mathsf{AP}|$ number of times. Hence, we have

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

Recall that each message contains the possible states of the extended $\texttt{LTL}_3$ monitor. This means each message in Algorithm 6 needs

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

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

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

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

Theorem 3. *Rather than going through $t+1$ rounds of communication with peer monitors, each monitor can only go through $k+1$ rounds where k denotes the maximum number of*

monitor crashes that are possible in a particular state without loss of any information or correctness.

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

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

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

5.5 Experimental Results

In this section, we present the results of our experiments on monitoring formulas with respect to a synthetic model of the system and monitoring correctness and behavioral specifications on the Orange4Home [CLRC17] dataset for IoT.

5.5.1 Synthetic Experiments

Setup

We evaluate our decentralized system using different `LTL` formulae generated from specification patterns mentioned in [Dwy20]. The corresponding monitor is generated using $\texttt{LTL}_3$ tools [BLS11]. Each of the following experiments were conducted on the following combinations of total number of monitors in the system and the maximum number of crashes (t) that the system is prone to have:

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

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

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

As can be seen in Fig. 5.5, the distribution of monitor crashes for Bernoulli (0.9) is more left skewed when compared with uniform distribution. This is because in Bernoulli (0.9), the likelihood of a monitor crashing is higher compared to uniform where it is 0.5. Higher crash likelihood makes the monitor in the system crash earlier till the system reaches the maximum number of crashes allowed. We also notice that the likelihood of a monitor crashing is dependent on the read distribution of the atomic proposition over the monitors. More number of monitors read a atomic proposition when distributed uniformly compared to Bernoulli (0.1). As mentioned earlier, a monitor only crashes if there exists another monitor

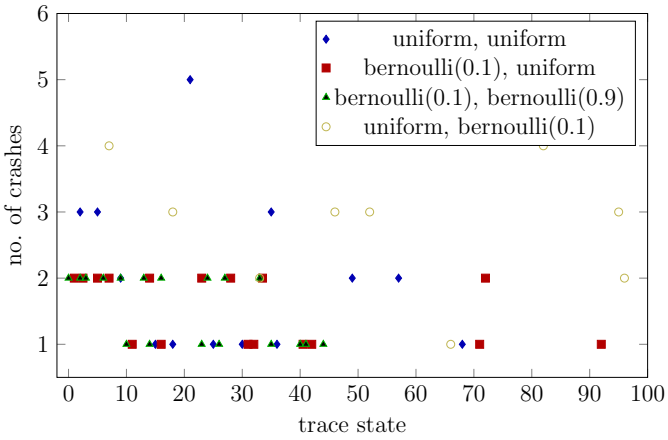


Figure 5.5: Crash distribution over a trace of length 100.

who has read the same atomic propositions. Thus, the likelihood of a monitor crashing is more for a read distribution of uniform compared to Bernoulli (0.1).

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

$$G_{S_j} = \bigcup_{i=1}^{n} \mathcal{S}_i^{S_j}$$

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

Similar to the tool, *DECENT-MON* [BF12], we test out each system configuration on three different traces where the probability of occurrence of an atomic proposition given a state is controlled by uniform distribution and Bernoulli distribution with 0.1 and 0.9 as a parameter. In our experiments, we study and report on the following metrics:

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

All of our experiments are run sufficiently enough to ensure 95% confidence interval.

No.	Type of formula	Formula	Size (Before)	Size (After)	Change (Times)
1		$\Box\ (\neg\ p)$	2	2	0
2		$\Diamond\ r \rightarrow (\neg\ p\ \mathcal{U}\ r)$	4	4	0
3	Absence	$\Box\ (q \rightarrow \Box\ (\neg\ p))$	3	3	0
4		$\Box\ ((q \wedge \neg\ r \wedge\ \Diamond\ r) \rightarrow (\neg\ p\ \mathcal{U}\ r))$	4	15	2.75
5		$\Box\ (q \wedge \neg\ r \rightarrow (\neg\ p\ \mathcal{U}\ (r \vee \Box\ \neg\ p)))$	3	12	3
6		$\Diamond\ (p)$	2	2	0
7		$\neg\ r\ \mathcal{U}\ ((p \wedge \neg\ r) \vee \Box\ \neg\ r)$	3	4	0.33
8	Existence	$\Box(\neg\ q) \vee \Diamond\ (q \wedge \Diamond\ p)$	3	3	0
9		$\Box(q \wedge \neg\ r \rightarrow (\neg\ r\ \mathcal{U}\ ((p \wedge \neg\ r) \vee \Box\neg\ r)))$	3	55	17.33
10		$\Box(q \wedge \neg\ r \rightarrow (\neg\ r\mathcal{U}(p \wedge \neg\ r)))$	3	55	17.33
11		$(\neg\ p\ \mathcal{U}\ ((p\ \mathcal{U}\ ((\neg\ p\mathcal{U}((p\mathcal{U}(\Box\neg\ p \vee \Box p)) \vee \Box\neg\ p)) \vee \Box p)) \vee \Box\neg\ p))$	1	1	0
12		$\Diamond r \rightarrow ((\neg\ p \wedge \neg\ r)\mathcal{U}(r \vee ((p \wedge \neg\ r)\mathcal{U}(r \vee ((\neg\ p \wedge \neg\ r)\mathcal{U}(r \vee ((p \wedge \neg r)\mathcal{U}(r \vee (\neg p\mathcal{U}r)))))))))$	8	8	0
13	Bounded Existence	$\Diamond q \rightarrow (\neg q\mathcal{U}(q \wedge (\neg p\mathcal{U}((p\mathcal{U}((\neg p\mathcal{U}((p\mathcal{U}(\Box\neg p \vee \Box p)) \vee \Box\neg p)) \vee \Box p)) \vee \Box\neg p))))$	7	7	0
14		$\Box((q \wedge \Diamond r) \rightarrow ((\neg p \wedge \neg r)\mathcal{U}(r \vee ((p \wedge \neg r)\mathcal{U}(r \vee ((\neg p \wedge \neg r)\mathcal{U}(r \vee ((p \wedge \neg r)\mathcal{U}(r \vee (\neg p\mathcal{U}r)))))))))$	1	1	0
15		$\Box(q \rightarrow ((\neg p \wedge \neg r)\mathcal{U}(r \vee ((p \wedge \neg r)\mathcal{U}(r \vee ((\neg p \wedge \neg r)\mathcal{U}(r \vee ((p \wedge \neg r)\mathcal{U}(r \vee (\neg p\mathcal{U}(r \vee \Box\neg p)) \vee \Box p)))))))))$	7	12	0.71
16		$\Box(p)$	2	2	0
17		$\Diamond r \rightarrow (p\mathcal{U}r)$	4	4	0
18	Universality	$\Box(q \rightarrow \Box(p))$	3	3	0
19		$\Box((q \wedge \neg r \wedge \Diamond r) \rightarrow (p\mathcal{U}r))$	4	11	1.75
20		$\Box(q \wedge \neg r \rightarrow (p\mathcal{U}(r \vee \Box p)))$	3	8	1.67
21		$\neg p\mathcal{U}(s \vee \Box\neg p)$	3	4	0.33
22		$\Diamond r \rightarrow (\neg p\mathcal{U}(s \vee r))$	4	8	1
23	Precedence	$\Box\neg q \vee \Diamond(q \wedge (p\mathcal{U}(s \vee \Box\neg p)))$	3	8	1.67
24		$\Box((q \wedge \neg r \wedge \Diamond r) \rightarrow (\neg p\mathcal{U}(s \vee r)))$	4	16	3
25		$\Box(q \wedge \neg r \rightarrow (\neg p\mathcal{U}((s \vee r) \vee \Box\neg p)))$	3	9	2
26		$\Box(p \rightarrow \Diamond s)$	1	1	0
27		$\Diamond r \rightarrow (p \rightarrow (\neg r\mathcal{U}(s \wedge \neg r)))\mathcal{U}r$	4	7	0.75
28	Response	$\Box(q \rightarrow \Box(p \rightarrow \Diamond s))$	1	1	0
29		$\Box((q \wedge \neg r \wedge \Diamond r) \rightarrow (p \rightarrow (\neg r\mathcal{U}(s \wedge \neg r)))\mathcal{U}r)$	4	38	8.5
30		$\Box(q \wedge \neg r \rightarrow (p \rightarrow (\neg r\mathcal{U}(s \wedge \neg r)))\mathcal{U}(r \vee \Box(p \rightarrow (\neg r\mathcal{U}(s \wedge \neg r)))))$	4	36	8
31		$\Diamond p \rightarrow (\neg p\mathcal{U}(s \wedge \neg p \wedge \bigcirc(\neg p\mathcal{U}t)))$	4	6	0.5
32		$\Diamond r \rightarrow (\neg p\mathcal{U}(r \vee (s \wedge \neg p \wedge \bigcirc(\neg p\mathcal{U}t))))$	5	16	2.2
33		$(\Box\neg q) \vee (\neg q\mathcal{U}(q \wedge \Diamond p \rightarrow (\neg p\mathcal{U}(s \wedge \neg p \wedge \bigcirc(\neg p\mathcal{U}t)))))$	5	15	2
34		$\Box((q \wedge \Diamond r) \rightarrow (\neg p\mathcal{U}(r \vee (s \wedge \neg p \wedge \bigcirc(\neg p\mathcal{U}t)))))$	5	32	5.4
35		$\Box(q \rightarrow (\Diamond p \rightarrow (\neg p\mathcal{U}(r \vee (s \wedge \neg p \wedge \bigcirc(\neg p\mathcal{U}t))))))$	4	20	4
36	Precedence Chain	$(\Diamond(s \wedge \bigcirc\Diamond t)) \rightarrow ((\neg s)\mathcal{U}p))$	4	7	0.75
37		$\Diamond r \rightarrow ((\neg(s \wedge (\neg r) \wedge \bigcirc(\neg r\mathcal{U}(t \wedge \neg r))))\mathcal{U}(r \vee p))$	5	17	2.4
38		$(\Box\neg q) \vee ((\neg q)\mathcal{U}(q \wedge ((\Diamond(s \wedge \bigcirc\Diamond t)) \rightarrow ((\neg s)\mathcal{U}p)))$	5	18	2.6
39		$\Box((q \wedge \Diamond r) \rightarrow ((\neg(s \wedge (\neg r) \wedge \bigcirc(\neg r\mathcal{U}(t \wedge \neg r))))\mathcal{U}(r \vee p)))$	5	36	6.2
40		$\Box(q \rightarrow (\neg(s \wedge (\neg r) \wedge \bigcirc(\neg r\mathcal{U}(t \wedge \neg r)))\mathcal{U}(r \vee p) \vee \Box(\neg(s \wedge \bigcirc\Diamond t))))$	4	24	5
41		$\Box(s \wedge \bigcirc\Diamond t \rightarrow \bigcirc(\Diamond(t \wedge \Diamond p)))$	1	1	0
42		$\Diamond r \rightarrow (s \wedge \bigcirc(\neg r\mathcal{U}t) \rightarrow \bigcirc(\neg r\mathcal{U}(t \wedge \Diamond p)))\mathcal{U}r$	6	16	1.67
43		$\Box(q \rightarrow \Box(s \wedge \bigcirc\Diamond t \rightarrow \bigcirc(\neg t\mathcal{U}(t \wedge \Diamond p))))$	1	1	0
44		$\Box((q \wedge \Diamond r) \rightarrow (s \wedge \bigcirc(\neg r\mathcal{U}t) \rightarrow \bigcirc(\neg r\mathcal{U}(t \wedge \Diamond p)))\mathcal{U}r)$	1	1	0
45		$\Box(q \rightarrow (s \wedge \bigcirc(\neg r\mathcal{U}t) \rightarrow \bigcirc(\neg r\mathcal{U}(t \wedge \Diamond p)))\mathcal{U}(r \vee \Box(s \wedge \bigcirc(\neg r\mathcal{U}r) \rightarrow \bigcirc(\neg r\mathcal{U}(t \wedge \Diamond p)))))$	1	1	0
46	Response Chain	$\Box(p \rightarrow \Diamond(s \wedge \bigcirc\Diamond t))$	1	4	3
47		$\Diamond r \rightarrow (p \rightarrow (\neg r\mathcal{U}(s \wedge \neg r \wedge \bigcirc(\neg r\mathcal{U}t))))\mathcal{U}r$	5	14	1.8
48		$\Box(q \rightarrow \Box(p \rightarrow (s \wedge \bigcirc\Diamond t)))$	3	12	3
49		$\Box((q \wedge \Diamond r) \rightarrow (p \rightarrow (\neg r\mathcal{U}(s \wedge \neg r \wedge \bigcirc(\neg r\mathcal{U}t))))\mathcal{U}r)$	7	35	4
50		$\Box(q \rightarrow (p \rightarrow (\neg r\mathcal{U}(s \wedge \neg r \wedge \bigcirc(\neg r\mathcal{U}t)))\mathcal{U}(r \vee \Box(p \rightarrow (s \wedge \bigcirc\Diamond t))))$	7	48	5.86
51		$\Box(p \rightarrow \Diamond(s \wedge \neg z \wedge \bigcirc(\neg z\mathcal{U}t)))$	1	1	0
52		$\Diamond r \rightarrow (p \rightarrow (\neg r\mathcal{U}(s \wedge \neg r \wedge \neg z \wedge \bigcirc((\neg r \wedge \neg z)\mathcal{U}t))))\mathcal{U}r$	5	28	4.6
53	Constrained Chain	$\Box(q \rightarrow \Box(p \rightarrow (s \wedge \neg z \wedge \bigcirc(\neg z\mathcal{U}t))))$	4	28	6
54		$\Box((q \wedge \Diamond r) \rightarrow (p \rightarrow (\neg r\mathcal{U}(s \wedge \neg r \wedge \neg z \wedge \bigcirc((\neg r \wedge \neg z)\mathcal{U}t))))\mathcal{U}r)$	8	40	4
55		$\Box(q \rightarrow (p \rightarrow (\neg r\mathcal{U}(s \wedge \neg r \wedge \neg z \wedge \bigcirc((\neg r \wedge \neg z)\mathcal{U}t))))\mathcal{U}(r \vee \Box(p \rightarrow (s \wedge \neg z \wedge \bigcirc(\neg z\mathcal{U}t)))))$	8	40	4

Table 5.1: List of formulas used to check our algorithm.

Analysis of Results

As mentioned earlier, we have put our system to the test with respect to all the LTL formulas mentioned in [Dwy20] (for specification patterns) under all the different scenarios explained above but, for both space and redundancy of similar observations, below we only mention results for the following LTL formulas (the full list of formulas in [Dwy20] can be

found in Table 5.1):

$$\varphi_4 = \square((q \wedge \neg r \wedge \diamondsuit r) \rightarrow (\neg p \; \mathcal{U} \; r))$$

$$\varphi_{17} = \diamondsuit r \rightarrow (p \; \mathcal{U} \; r)$$

$$\varphi_{38} = (\square \neg q) \vee ((\neg q) \; \mathcal{U} \; (q \wedge ((\diamondsuit(s \wedge \bigcirc \diamondsuit t)) \rightarrow ((\neg s) \; \mathcal{U} \; p))))$$

$$\varphi_{51} = \square(p \rightarrow \diamondsuit(s \wedge \neg z \wedge \bigcirc(\neg z \; \mathcal{U} \; t)))$$

Impact of monitor crashes. As expected a higher number of monitor crashes results in an increase in the average number of rounds when monitoring. In Fig. 5.6a, for `LTL` formula φ_4, we observe that the average number of rounds is significantly improved when accounting for only the number of crashes that are possible given a state of the execution. For example, in a system with $t = 8$ and with read and crash distribution being binomial and uniform respectively, the average number of rounds is only around 3 (reduced from usual 8).

In Fig. 5.6b, we see for φ_4 that with increase in the number of monitor crashes, the number of messages exchanged among the monitors increase as well, since in each round a monitor in the system shares its observation with other monitors in the system, thereby making the total number of messages directly proportional to the number of monitors present and also the number of rounds.

Following our setup described in Fig. 5.5, the distribution of crashes also have an effect on the average number of rounds and the number of messages being passed in the system. The more left skewed will be the distribution of the monitor crashes, the less average number of rounds are required to come to a consensus among the monitors. This is because a left skewed monitor crash distribution equates to the mean number of monitors present in the system being low and there-by lower number of rounds as well as number of messages.

Communication after l states: We test our algorithm on different values of l, starting from 1 when the communication between monitors take place after every state and

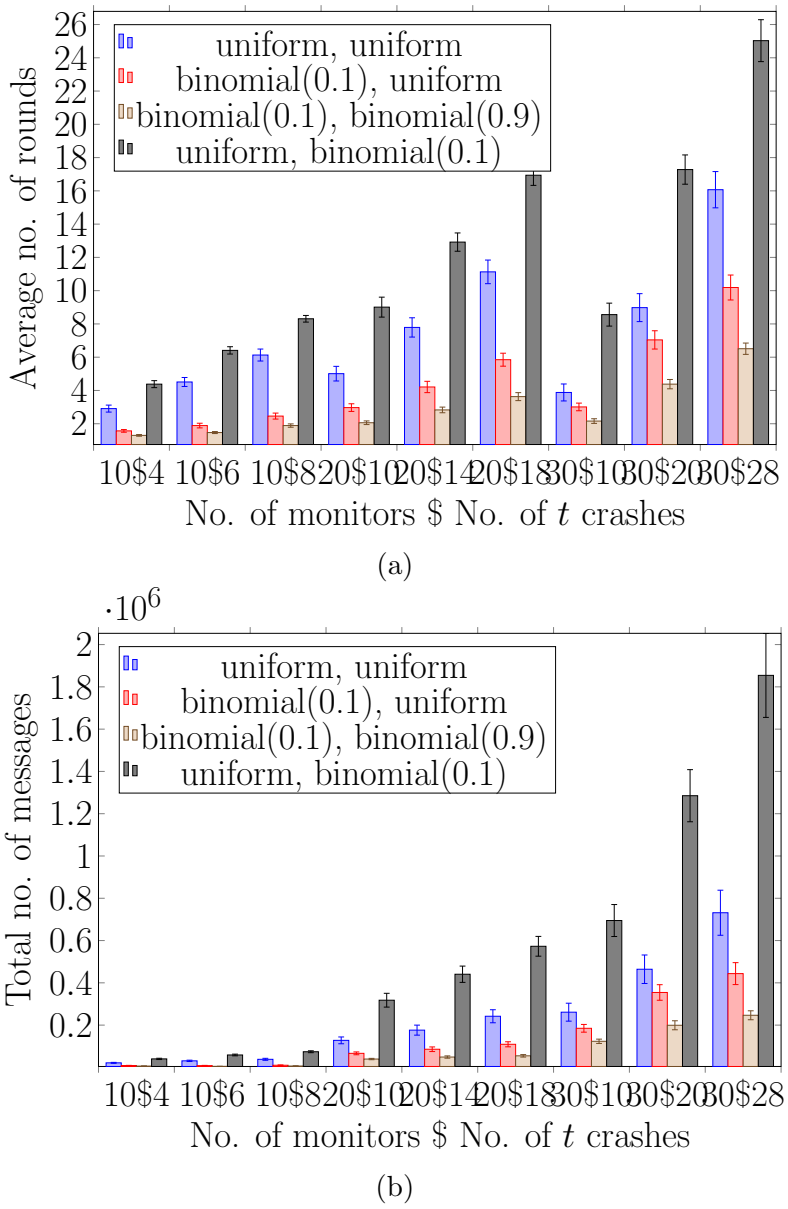


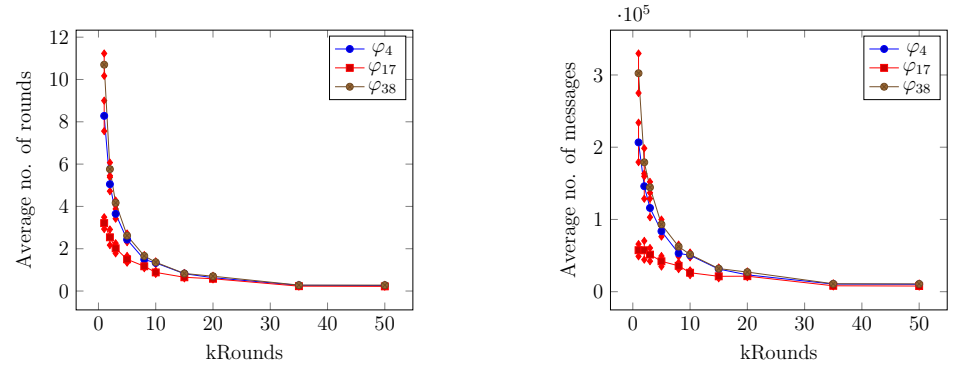
Figure 5.6: Average # of rounds and total # of messages sent per situation for different read and crash distributions for flip-flop distributed trace for φ_4 with $l = 1$.

going all the way to 50 when the monitors communicate only twice for a trace length of 100. As stated in Theorem 1, the correctness of the protocol is not effected by changing the values of l however as seen in Fig. 5.7 for different `LTL` specifications, the average number of rounds and average number of messages decrease with increasing values of l. For lower l, the communication takes place more often than higher values of l and thus accounting for higher values of number of rounds and number of messages.

The average size of messages increases with an increase in the value of l. This is because the size of a message depends on the number of states present in the local observation of a monitor. With communication happening after l states, the local observation constitutes of more number of states than when it was happening after every state. This can be seen when comparing the results of Figure 5.7c for different LTL formula. The size of messages for φ_{38} is substantially larger when compared to that of others due to the more number of states in its updated LTL_3 monitor automata along with higher number of atomic proposition. We also see that with increasing the value of l, the number of monitor crashes decreases. This is because, with increasing the value of l, communication is limited to only after every l states and there-by decreasing the the number of communicating rounds and there-by decreasing the number of monitor crashes. Taking all the plots into consideration, we observe that the benefit from lower the number of rounds and messages out-weights the drawback from the increase in the size of messages for any value of $l \geq 5$.

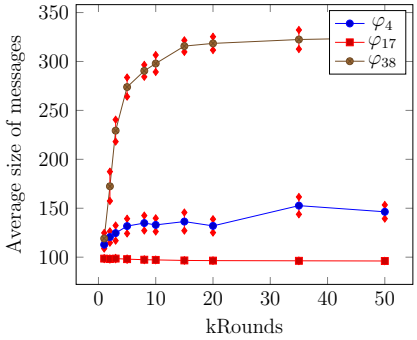
5.5.2 Orange4Home Dataset

Orange4Home [CLRC17] is a dataset capturing routines of daily living in Amiqual4Home's smart home environment. It is a result of a joint work between Orange Labs and Inria. The dataset consists of around 180 hours of recording of activities of daily living of a single occupant, spanning 4 consecutive weeks of work days. The dataset contains recordings of a total of 236 sensors scattered throughout the apartment and for 20 different classes of activities. We divide all specifications into two categories: (1) Behavioral correctness: monitor the correctness of the different sensors (2) Activity of Daily Living (ADL): monitoring the activity that the occupant is upto using the values of different sensors. In Fig. 5.8, we show the results for various values of k keeping the read and crash distribution to be uniform and Bernoulli (0.1) respectively and report on the number of rounds, number of messages, size of the message and actual number of monitor crashing for a system with

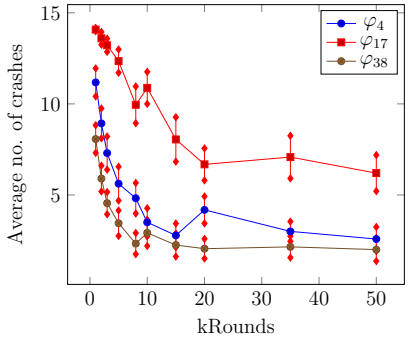


(a) Average # of rounds

(b) Total # of messages sent



(c) Average size of messages



(d) Average number of crashes

Figure 5.7: Impact of communicating after l states for various LTL formula on synthetic data.

30 monitors and $t = 20$ monitoring the following specifications:

$$\varphi_{o4h\_1} = \Box(switch \rightarrow \bigcirc(light\, \mathcal{U}\, \neg switch)$$

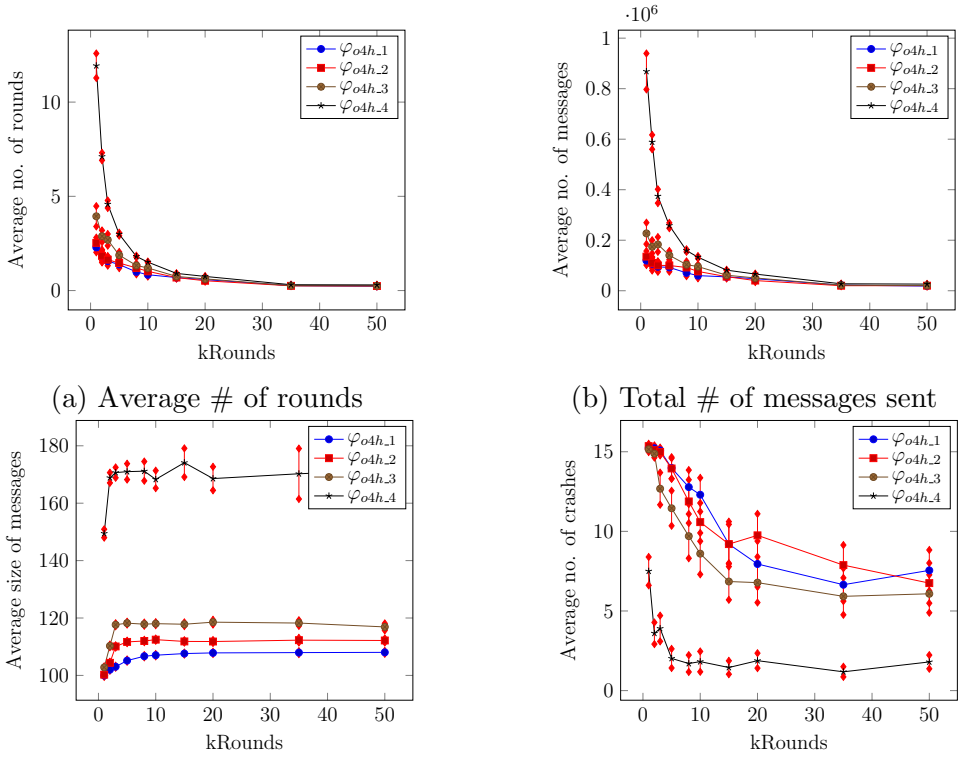
$$\varphi_{o4h\_2} = \Box\Diamond_{\leq 5}(cooktop \vee oven)$$

$$\varphi_{o4h\_3} = \Box\Diamond_{\leq 5}(kitchen\_sink \vee kitchen\_fridge \vee kitchen\_cupboard)$$

$$\varphi_{o4h\_4} = \Box\Diamond_{\leq 5}(cooktop \vee oven \vee kitchen\_sink \vee kitchen\_fridge \vee nkitchen\_cupboard \vee kitchen\_dishwasher)$$

Formula	Size (Before)	Size (After)	Change (Times)
φ_{o4h_1}	3	3	1
φ_{o4h_2}	3	4	0.33
φ_{o4h_3}	3	6	1
φ_{o4h_4}	3	33	10

Table 5.2: Formula from Orange4Home.



(a) Average # of rounds

(b) Total # of messages sent

(c) Average size of messages

(d) Average number of crashes

Figure 5.8: Impact of communicating after l states for various LTL formula on data from Orange4Home dataset.

First we construct the equivalent LTL_3 monitors using Algorithm 8. The change in the number of states of the final automata can be observed in Table 5.2. Monitoring ADL specifications involve the system keeping a track of the passage of time, essential in monitoring a time bounded specification as is the case with φ_{o4h_2} through φ_{o4h_4}. Apart from similar observation to the synthetic data for increasing values of k, we observe in Fig. 5.8 that monitoring specifications involving more number of atomic propositions have higher message size. The higher message size can be explained by Theorem 2 which shows the message complexity when using an extended LTL_3 monitor is directly proportional to $|\mathsf{AP}|$. Additionally, higher value of l, decreases the number of communicating rounds and thus accounting for lower number of monitor crashes. Subsequently, lower number of monitor crashes equates to higher number of active monitors in the system and therefore higher number of rounds and higher number of messages.

5.6 Summary and Limitation

In this chapter, we propose a runtime verification algorithm, where a set of decentralized synchronous monitors that have only a partial view of the underlying system continually evaluate formulas in the linear temporal logic (LTL). The non-deterministic nature of the evaluation procedure due to partial observations makes resolving the current state of the execution indistinguishable. Thus, we propose an SMT-based transformation algorithm to obtain minimum size LTL_3 monitors.

However, the synchronous nature of the distributed system makes for a limited application for such an approach. Also, as shown in Chapter 3, an automata based approach often requires more results to be taken into consideration than needed. On the contrary, a progression based approach might need less number of states of automata that needs to be remembered by the monitors. Thereby minimizing the cost of communication by a considerable amount.

Chapter 6

Decentralized Runtime Verification for Stream-based Specifications

6.1 Introduction

In this chapter, we advocate for a runtime verification (RV) approach, to monitor the behavior of a distributed system with respect to a formal specification. Applying RV to multiple components of an ICS can be viewed as the general problem of distributed RV, where a centralized or decentralized monitor(s) observe the behavior of a distributed system in which the processes do not share a global clock. Although RV deals with finite executions, the lack of a common global clock prohibits it from having a total ordering of events in a distributed setting. In other words, the monitor can only form a partial ordering of events which may yield different evaluations. Enumerating all possible interleavings of the system at runtime incurs in an exponential blowup, making the approach not scalable. To add to this already complex task, a PLC often requires time sensitive aggregation of data from multiple sources.

We propose an effective, sound and complete solution to distributed RV for the popular

(Submitted) Ritam Ganguly, and Borzoo Bonakdarpour, Decentralized Runtime Verification of Stream-based Partially-Synchronous Distributed System, ACM SIGBED International Conference on Embedded Software (EMSOFT-2023).

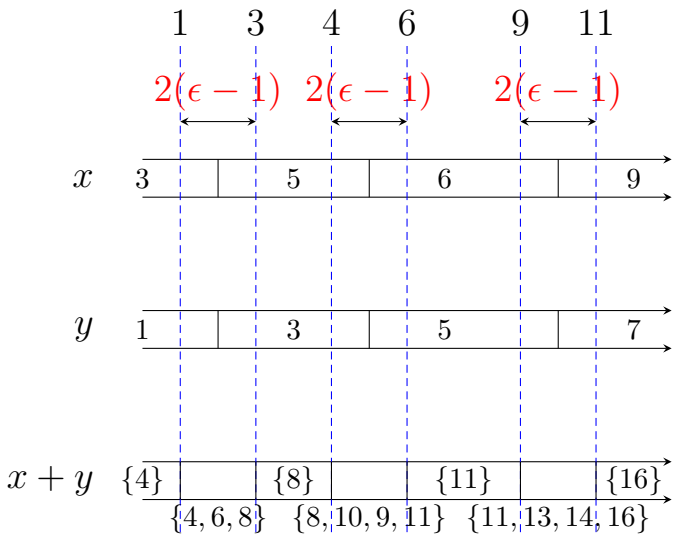


Figure 6.1: Partially Synchronous LOLA.

stream-based specification language LOLA [DSS+05]. Compared to other temporal logic, LOLA can describe both correctness/failure assertions along with statistical measures that can be used for system profiling and coverage analysis. To present a high level of LOLA example, consider two input streams x and y and a output stream, *sum* as shown in Fig. 6.1. Stream x has the value 3 until time instance 2 when it changes to 5 and so on.

```
input x:uint
input y:uint
output sum := x+y
```

We consider a fault proof decentralized set of monitors where each monitor only has a partial view of the system and has no access to a global clock. In order to limit the blow-up of states posed by the absence of the global clock, we make a practical assumption about the presence of a bounded clock skew ϵ between all the local clocks, guaranteed by a clock synchronization algorithm (like NTP [Mil10]). This setting is known to be *partially synchronous*. As can be seen in Fig. 6.1, any two events less than $\epsilon = 2$ time apart is considered to be concurrent and thus the non-determinism of the time of occurrence of each event is restricted to $\epsilon - 1$ on either side. When attempting to evaluate the output stream *sum*, we need to take into consideration all the possible time of occurrence of the values. For example, when evaluating the value of *sum* at time 1, we need to consider the value of x

(resp. y) as 3 and 5 (resp. 1 and 3) which evaluates to 4, 6 and 8. The same can be observed for evaluations across all time instances.

Our first contribution in this chapter is introducing a partially synchronous semantics for Lola. In other words, we define Lola which takes into consideration a clock-skew of ϵ when evaluating a stream expression. Second, we introduce an SMT-based associated equation rewriting technique over a partially observable distributed system, which takes into consideration the values observed by the monitor and rewrites the associated equation. The monitors are able to communicate within themselves and are able to resolve the partially evaluated equations into completely evaluated ones.

We have proved the correctness of our approach and the upper and lower bound of the message complexity. Additionally, we have completely implemented our technique and report the results of rigorous synthetic experiments, as well as monitoring correctness and aggregated results of several ICS. As identified in [ACZ20], most attacks on ICS components try to alter the value reported to the PLC in-order to make the PLC behave erroneously. Through our approach, we were able to detect these attacks in-spite of the clock asynchrony among the different components with deterministic guarantee. We also argue that our approach was able to evaluate system behavior aggregates that makes studying these system easier by the human operator. Unlike machine learning approaches (e.g., [PMA15b, PMA15a, BHBB+14]), our approach will never raise false negatives. We put our monitoring technique to test, studying the effects of different parameters on the runtime and size of the message sent from one monitor to other and report on each of them.

6.2 Partially Synchronous Lola

In this section, we extend the semantics of Lola to one that can accommodate reasoning about distributed systems.

6.2.1 Distributed Streams

Here, we refer to a global clock which will act as the "real" timekeeper. It is to be noted that the presence of this global clock is just for theoretical reasons and it is not available to any of the individual streams.

We assume a *partially synchronous* system of n streams, denoted by $\mathcal{A} = \{\alpha_1, \alpha_2, \cdots, \alpha_n\}$. For each stream α_i, where $i \in [1, |\mathcal{A}|]$, the local clock can be represented as a monotonically increasing function $c_i : \mathbb{Z}_{\geq 0} \to \mathbb{Z}_{\geq 0}$, where $c_i(\mathcal{G})$ is the value of the local clock at global time $\mathcal{G}$. Since we are dealing with discrete-time systems, for simplicity and without loss of generality, we represent time with non-negative integers $\mathbb{Z}_{\geq 0}$. For any two streams α_i and α_j, where $i \neq j$, we assume:

$$\forall \mathcal{G} \in \mathbb{Z}_{\geq 0}. \mid c_i(\mathcal{G}) - c_j(\mathcal{G}) \mid < \epsilon,$$

where $\epsilon > 0$ is the maximum clock skew. The value of ϵ is constant and is known (e.g., to a monitor). This assumption is met by the presence of an off-the-shelf clock synchronization algorithm, like NTP [Mil10], to ensure bounded clock skew among all streams. The local state of stream α_i at time σ is given by $\alpha_i(\sigma)$, where $\sigma = c_i(\mathcal{G})$, that is the local time of occurrence of the event at some global time $\mathcal{G}$.

Definition 14. *A* distributed stream *consisting of $\mathcal{A} = \{\alpha_1, \alpha_2, \ldots, \alpha_n\}$ streams of length $N + 1$ is represented by the pair $(\mathcal{E}, \rightsquigarrow)$, where $\mathcal{E}$ is a set of all local states (i.e., $\mathcal{E} = \cup_{i \in [1,n], j \in [0,N]} \alpha_i(j)$) partially ordered by Lamport's happened-before ($\rightsquigarrow$) relation [Lam78], subject to the partial synchrony assumption:*

- *For every stream α_i, $1 \leq i \leq |\mathcal{A}|$, all the events happening on it are totally ordered, that is,*
$$\forall i, j, k \in \mathbb{Z}_{\geq 0} : (j < k) \to (\alpha_i(j) \rightsquigarrow \alpha_i(k))$$
- *For any two streams α_i and α_j and two corresponding events $\alpha_i(k), \alpha_j(l) \in \mathcal{E}$, if $k + \epsilon < l$ then, $\alpha_i(k) \rightsquigarrow \alpha_j(l)$, where ϵ is the maximum clock skew.*
- *For events, e, f, and g, if $e \rightsquigarrow f$ and $f \rightsquigarrow g$, then $e \rightsquigarrow g$.* □

Definition 15. *Given a distributed stream $(\mathcal{E}, \rightsquigarrow)$, a subset of events $\mathcal{C} \subseteq \mathcal{E}$ is said to form a* consistent cut *if and only if when $\mathcal{C}$ contains an event e, then it should also contain all*

such events that happened before e. Formally,

$$\forall e, f \in \mathcal{E}.(e \in \mathcal{C}) \wedge (f \rightsquigarrow e) \rightarrow f \in \mathcal{C}. \blacksquare$$

The frontier of a consistent cut $\mathcal{C}$, denoted by $\mathsf{front}(\mathcal{C})$ is the set of all events that happened last in each stream in the cut. That is, $\mathsf{front}(\mathcal{C})$ is a set of $\alpha_i(last)$ for each $i \in [1, |\mathcal{A}|]$ and $\alpha_i(last) \in \mathcal{C}$. We denote $\alpha_i(last)$ as the last event in α_i such that $\forall \alpha_i(\sigma) \in \mathcal{C}.(\alpha_i(\sigma) \neq \alpha_i(last)) \rightarrow (\alpha_i(\sigma) \rightsquigarrow \alpha_i(last))$.

6.2.2 Partially Synchronous Lola

We define the semantics of Lola specifications for partially synchronous distributed streams in terms of the evaluation model. The absence of a common global clock among the stream variables and the presence of the clock synchronization makes way for the output stream having multiple values at any given time instance. Thus, we update the evaluation model, so that $\alpha_i(j)$ and $\upsilon(t_i)(j)$ are now defined by *sets* rather than just a single value. This is due to nondeterminism caused by partial synchrony, i.e., the bounded clock skew ϵ.

Definition 16. *Given a* Lola *[DSS+05] specification φ over independent variables, $t_1, \cdots, t_m$ of type $\mathsf{T}_1, \cdots, \mathsf{T}_m$ and dependent variables, $s_1, \cdots, s_n$ of type $\mathsf{T}_{m+1}, \cdots, \mathsf{T}_{m+n}$ and $\tau_1, \cdots, \tau_m$ be the streams of length $N+1$, with τ_i of type T_i. The tuple of streams $\langle \alpha_1, \cdots, \alpha_n \rangle$ of length $N+1$ with corresponding types is called the evaluation model in the partially synchronous setting, if for every equation in φ:*

$$s_i = e_i(t_1, \cdots, t_m, s_1, \cdots, s_n),$$

$\langle \alpha_1, \cdots, \alpha_n \rangle$ satisfies the following associated equations:

$$\alpha_i(j) = \big\{\upsilon(e_i)(k) \mid \max\{0, j-\epsilon+1\} \leq k \leq \min\{N, j+\epsilon-1\}\big\}$$

where $\upsilon(e_i)(j)$ is defined as follows. For the base cases:

$$\begin{aligned}
\upsilon(c)(j) &= \{c\} \\
\upsilon(t_i)(j) &= \big\{\tau_i(k) \mid \max\{0, j-\epsilon+1\} \leq k \leq \min\{N, j+\epsilon-1\}\big\} \\
\upsilon(s_i)(j) &= \alpha_i(j)
\end{aligned}$$

For the inductive cases:

$$\upsilon\Big(f(e_1,\cdots,e_p)\Big)(j) = \Big\{f(e'_1,\cdots,e'_p) \mid e'_1 \in \upsilon(e_1)(j),\cdots,e'_p \in \upsilon(e_p)(j)\Big\}$$

$$\upsilon(ite(b,e_1,e_2))(j) = \begin{cases} \upsilon(e_1)(j) & true \in \upsilon(b)(j) \\ \upsilon(e_2)(j) & false \in \upsilon(b)(j) \end{cases}$$

$$\upsilon(e[k,c])(j) = \begin{cases} \upsilon(e)(j+k) & if\ 0 \leq j+k \leq N \\ c & otherwise \end{cases}$$

□

```
input read:bool
input write:bool
output countRead := ite(read, countRead[-1,0] + 1, countRead[-1,0])
output countWrite := ite(write, countWrite[-1,0] + 1, countWrite[-1,0])
output check := (countWrite - countRead) <= 2
```

Example 1. *Consider the above* Lola *specification, φ, over the independent boolean variables read and write: In Fig. 6.2, we have two input stream read and write which denotes the time instances where the corresponding events take place. It can be imagined that read and write are streams of type boolean with true values at time instances $4,6,7$ and $2,3,5,6$ and false values at all other time instances respectively. We evaluate the above mentioned* Lola *specification considering a time synchronization constant, $\epsilon = 2$. The corresponding associated equations, φ_α, are:*

$$countRead(j) = \begin{cases} \texttt{ite}(read,1,0) & j = 0 \\ \texttt{ite}\Big(read, countRead(j-1)+1, countRead(j)\Big) & j \in [1,N) \end{cases}$$

$$countWrite(j) = \begin{cases} \texttt{ite}(write,1,0) & j = 0 \\ \texttt{ite}\Big(write, countWrite(j-1)+1, countWrite(j)\Big) & j \in [1,N) \end{cases}$$

$$check(j) = \Big(countWrite(j) - countRead(j)\Big) \leq 2$$

Similar to the synchronous case, evaluation of the partially synchronous Lola specification involves creating the dependency graph.

Definition 17. *A dependency graph for a* Lola *specification, φ is a weighted directed multi-graph $G = \langle V, E\rangle$, with vertex set $V = \{s_1,\cdots,s_n,t_1,\cdots,t_m\}$. An edge $e : \langle s_i, s_k, w\rangle$ (resp.*

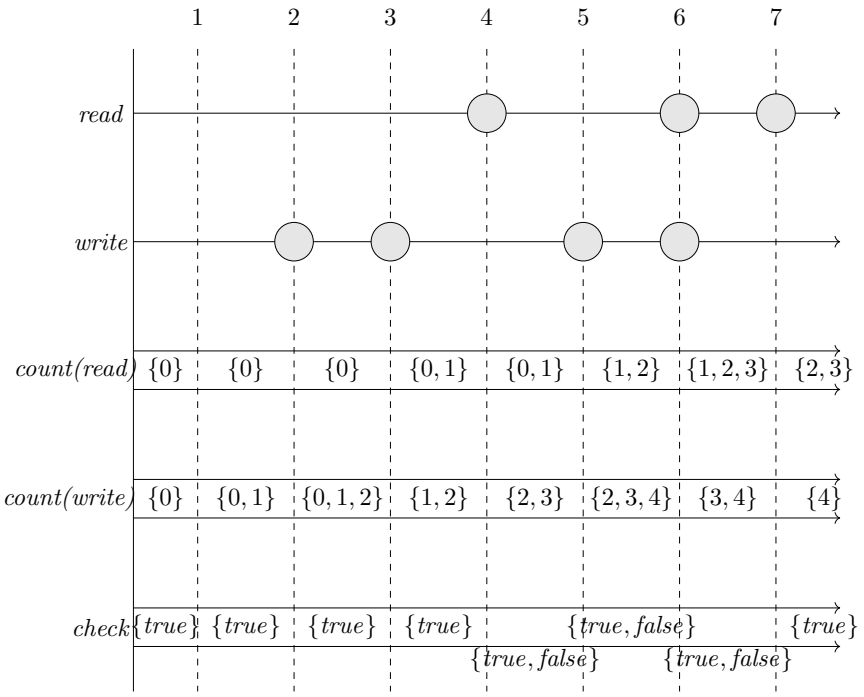


Figure 6.2: Partially Synchronous LOLA Example.

$e : \langle s_i, t_k, w \rangle$) *labeled with a weight* $w = \{\omega \mid p - \epsilon < \omega < p + \epsilon\}$ *is in* E *iff the equation for* $\alpha_i(j)$ *contains* $\alpha_k(j + p)$ *(resp.* $\tau_k(j + p)$*) as a sub-expression, for some* j *and offset* p. □

Intuitively, the dependency graph records that evaluation of a s_i at a particular position depends on the value of s_k (resp. t_k), with an offset in w. It is to be noted that there can be more than one edge between a pair of vertex (s_i, s_k) (resp. (s_i, t_k)). Vertices labeled by t_i do not have any outgoing edges.

Example 2. *Consider the* LOLA *specification over the independent integer variable a:*

```
input a : uint
output b1 := b2[1, 0] + ite(b2[-1,7] <= a[1, 0], b2[-2,0], 6)
output b2 := b1[-1,8]
```

Its dependency graph, shown in Fig. 6.3 for $\epsilon = 2$*, has 1 edge from b1 to a with a weight* $\{0, 1, 2\}$*. Similarly, there are 3 edges from b1 to b2 with weights* $\{0, 1, 2\}, \{-2, -1, 0\}$ *and* $\{-3, -2, -1\}$ *and 1 edge from b2 to b1 with a weight of* $\{-2, -1, 0\}$

Given a set of partially synchronous input streams $\{\alpha_1, \alpha_2, \cdots, \alpha_{|\mathcal{A}|}\}$ of respective type

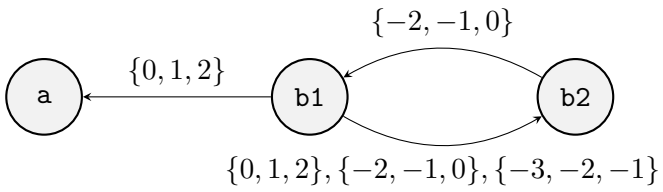


Figure 6.3: Dependency Graph Example.

$\mathbb{T} = \{\mathsf{T}_1, \mathsf{T}_2, \cdots, \mathsf{T}_{|\mathcal{A}|}\}$ and a LOLA specification, φ, the evaluation of φ is given by

$$(\alpha_1, \alpha_2, \cdots, \alpha_{|\mathcal{A}|}) \models_{PS} \varphi$$

where, $\models_{PS}$ denotes the partially synchronous evaluation.

6.3 Decentralized Monitoring Architecture

6.3.1 Overall Picture

We consider a decentralized online monitoring system comprising of a fixed number of $|\mathcal{M}|$ reliable monitor processes $\mathcal{M} = \{M_1, M_2, \cdots, M_{|\mathcal{M}|}\}$ that can communicate with each other by sending and receiving messages through a complete point-to-point bidirectional communication links. Each communication link is also assumed to be reliable, i.e., there is no loss or alteration of messages. Similar to the distributed system under observation, we assume the clock on the individual monitors are asynchronous, with clock synchronization constant $= \epsilon_M$.

Throughout this section we assume that the global distributed stream consisting of complete observations of $|\mathcal{A}|$ streams is only partially visible to each monitor. Each monitor process locally executes an identical sequential algorithm which consists of the following steps (we will generalize this approach in Section 6.6). In other words, an evaluation iteration of each monitor consists of the following steps:

1. Reads the a subset of $\mathcal{E}$ events (visible to M_i) along with the corresponding time and valuation of the events, which results in the construction of a *partial distributed stream*;

Algorithm 9: Behavior of a Monitor M_i, for $i \in [1, |\mathcal{M}|]$.

1: **for** $j = 0$ to N **do**
2: Let $(\mathcal{E}_i, \rightsquigarrow_i)_j$ be the partial distributed stream view of M_i
3: $LS_j \leftarrow \left[(\mathcal{E}, \rightsquigarrow) \models_{PS} \varphi_\alpha\right]$
4: **Send:** broadcasts symbolic view LS_j
5: **Receive:** $\Pi_j \leftarrow \{LS_j^k \mid 1 \leq k \leq \mathcal{M}\}$
6: **Compute:** $LS_{j+1} \leftarrow LC(\Pi_j)$
7: **end for**

2. Each monitor evaluates the LOLA specification φ given the partial distributed stream;
3. Every monitor, broadcasts a message containing rewritten associated equations of φ, denoted LS, and
4. Based on the message received containing associated equations, each monitor amalgamates the observations of all the monitors to compose a set of associated equations. After a evaluation iteration, each monitor will have the same set of associated equations to be evaluated on the upcoming distributed stream.

The message sent from monitor M_i at time π to another monitor M_j, for all $i, j \in [1, |\mathcal{M}|]$, during a evaluation iteration of the monitor is assumed to reach latest by time $\pi + \epsilon_M$. Thus, the length of an *evaluation iteration* k can be adjusted to make sure the message from all other monitors reach before the start of the next evaluation iteration.

6.3.2 Detailed Description

We now explain in detail the computation model (see Algorithm 9). Each monitor process $M_i \in \mathcal{M}$, where $i \in [1, |\mathcal{M}|]$, attempts to read $e \in \mathcal{E}$, given the distributed stream, $(\mathcal{E}, \rightsquigarrow)$. An event can either be observable, or not observable. Due to distribution, this results in obtaining a partial distributed stream $(\mathcal{E}_i, \rightsquigarrow)$ defined below.

Definition 18. *Let $(\mathcal{E}, \rightsquigarrow)$ be a distributed stream. We say that $(\mathcal{E}', \rightsquigarrow)$ is a* partial distributed stream *for $(\mathcal{E}, \rightsquigarrow)$ and denote it by $(\mathcal{E}', \rightsquigarrow) \sqsubseteq (\mathcal{E}, \rightsquigarrow)$ iff $\mathcal{E}' \subseteq \mathcal{E}$ (the happened before relation is obviously preserved).* □

We now tie partial distributed streams to a set of decentralized monitors and the fact that decentralized monitors can only partially observe a distributed stream. First, all un-

observed events is replaced by $\natural$, i.e., for all $\alpha_i(\sigma) \in \mathcal{E}$ if $\alpha_i(\sigma) \notin \mathcal{E}_i$ then $\mathcal{E}_i = \mathcal{E}_i \cup \{\alpha_i(\sigma) = \natural\}$.

Definition 19. *Let $(\mathcal{E}, \rightsquigarrow)$ be a distributed stream and $\mathcal{M} = \{M_1, M_2, \cdots, M_{|\mathcal{M}|}\}$ be a set of monitors, where each monitor M_i, for $i \in [1, |\mathcal{M}|]$ is associated with a partial distributed stream $(\mathcal{E}_i, \rightsquigarrow) \sqsubseteq (\mathcal{E}, \rightsquigarrow)$. We say that these monitor observations are consistent if*

- $\forall e \in \mathcal{E}.\exists i \in [1, |\mathcal{M}|].e \in \mathcal{E}_i$, *and*
- $\forall e \in \mathcal{E}_i.\forall e' \in \mathcal{E}_j.(e = e' \wedge e \neq \natural) \oplus \Big((e = \natural \vee e' = \natural)\Big)$,

where $\oplus$ denoted the exclusive-or operator.

In a partially synchronous system, there are different ordering of events and each unique ordering of events might evaluate to different values. Given a distributed stream, $(\mathcal{E}, \rightsquigarrow)$, a sequence of consistent cuts is of the form $\mathcal{C}_0\mathcal{C}_1\mathcal{C}_2 \cdots \mathcal{C}_N$, where for all $i \geq 0$: (1) $\mathcal{C}_i \subseteq \mathcal{E}$, and (2) $\mathcal{C}_i \subseteq \mathcal{C}_{i+1}$.

Given the semantics of partially-synchronous Lola, evaluation of output stream variable s_i at time instance j requires events $\alpha_i(k)$, where $i \in [1, |\mathcal{A}|]$ and $k \in \Big\{\pi \mid \max\{0, j-\epsilon+1\} \leq \pi \leq \{N, j+\epsilon-1\}\Big\}$. To translate monitoring of a distributed stream to a synchronous stream, we make sure that the events in the frontier of a consistent cut, $\mathcal{C}_j$ are $\alpha_i(k)$.

Let $\mathbb{C}$ denote the set of all valid sequences of consistent cuts. We define the set of all synchronous streams of $(\mathcal{E}, \rightsquigarrow)$ as follows:

$$\mathsf{Sr}(\mathcal{E}, \rightsquigarrow) = \Big\{\mathsf{front}(\mathcal{C}_0)\mathsf{front}(\mathcal{C}_1) \cdots \mid \mathcal{C}_0\mathcal{C}_1 \cdots \in \mathbb{C}\Big\}$$

Intuitively, $\mathsf{Sr}(\mathcal{E}, \rightsquigarrow)$ can be interpreted as the set of all possible "interleavings". The evaluation of the Lola specification, φ, with respect to $(\mathcal{E}, \rightsquigarrow)$ is the following :

$$\Big[(\mathcal{E}, \rightsquigarrow) \models_{PS} \varphi\Big] = \Big\{(\alpha_1, \cdots, \alpha_n) \models_S \varphi \mid (\alpha_1, \cdots, \alpha_n) \in \mathsf{Sr}(\mathcal{E}, \rightsquigarrow)\Big\}$$

This means that evaluating a partially synchronous distributed stream with respect to a Lola specification results in a set of evaluated results, as the computation may involve several streams. This also enables reducing the problem from evaluation of a partially synchronous distributed system to the evaluation of multiple synchronous streams, each

evaluating to unique values for the output stream, with message complexity

$$O\big(\epsilon^{|\mathcal{A}|}N|\mathcal{M}|^2\big) \quad \Omega(N|\mathcal{M}|^2)$$

6.3.3 Problem Statement

The overall problem statement requires that upon the termination of the Algorithm 9, the verdict of all the monitors in the decentralized monitoring architecture is the same as that of a centralized monitor which has the global view of the system

$$\forall i \in [1, m] : \mathsf{Result}_i = \Big[(\mathcal{E}, \rightsquigarrow) \models_{PS} \varphi\Big]$$

where $(\mathcal{E}, \rightsquigarrow)$ is the global distributed stream and φ is the LOLA specification with Result_i as the evaluated result by monitor M_i.

6.4 Calculating LS

In this section, we introduce the rules of rewriting LOLA associated equations given the evaluated results and observations of the system. In our distributed setting, evaluation of a LOLA specification involves generating a set of synchronous streams and evaluating the given LOLA specification on it (explained in Section 6.5). Here, we make use of the evaluation of LOLA specification into forming our local observation to be shared with other monitors in the system.

Given the set of synchronous streams, $(\alpha_1, \alpha_2, \cdots, \alpha_{|\mathcal{A}|})$, the symbolic locally computed result LS (see Algorithm 9) consists of associated LOLA equations, which either needs more information (data was unobserved) from other monitors to evaluate or the concerned monitor needs to wait (positive offset). In either case, the associated LOLA specification is shared with all other monitors in the system as the missing data can be observed by either monitors. We divide the rewriting rules into three cases, depending upon the observability of the value of the independent variables required for evaluating the expression e_i for all $i \in [1, n]$. Each

stream expression is categorized into three cases (1) completely unobserved, (2) completely observed or (3) partially observed. This can be done easily by going over the dependency graph and checking with the partial distributed stream read by the corresponding monitor.

Case 1 (Completely Observed). Formally, a completely observed stream expression s_i can be identified from the dependency graph, $G = \langle V, E \rangle$, as for all s_k (resp. t_k) $\langle s_i, s_k, w \rangle \in E$ (resp. $\langle s_i, t_k, w \rangle \in E$), $s_k(j+w) \neq \natural$ (resp. $t_k(j+w) \neq \natural$) are observed for time instance j. If yes, this signifies, that all independent and dependent variables required to evaluate $s_i(j)$, is observed by the monitor M, there by evaluating: $s_i(j) = e_i(s_1, \cdots, s_n, t_1, \cdots, t_m)$ and rewriting $s_i(j)$ to LS.

Case 2 (Completely Unobserved). Formally, we present a completely unobserved stream expression, s_i from the dependency graph, $G = \langle V, E \rangle$, as for all s_k (resp. t_k), $\langle s_i, s_k, w \rangle \in E$ (resp. $\langle s_i, t_k, w \rangle \in E$), $s_k(j+w) = \natural$ (resp. $t_k(j+w) = \natural$) are unobserved, for time instance j . This signifies that the valuation of neither variables are known to the monitor M. Thus, we rewrite the following stream expressions

$$s'_k(j) = \begin{cases} s_k(j+w) & 0 \leq j+w \leq N \\ \texttt{default} & \text{otherwise} \end{cases}$$

$$t'_k(j) = \begin{cases} t_k(j+w) & 0 \leq j+w \leq N \\ \texttt{default} & \text{otherwise} \end{cases}$$

for all $\langle s_i, s_k, w \rangle \in E$ and $\langle s_i, t_k, w \rangle \in E$, and include the rewritten associated equation for evaluating $s_i(j)$ as

$$s_i(j) = e_i(s'_1, \cdots, s'_n, t'_1, \cdots, t'_m)$$

It is to be noted that the `default` value of a stream variable, s_k (resp. t_k), depends on the corresponding type T_k (resp. T_{m+k}) of the stream.

Case 3 (Partially Observed). Formally, we present a partially observed stream expression, s_i from the dependency graph, $G = \langle V, E \rangle$, as for all s_k (resp. t_k), they are

either observed or unobserved, for time instance j. In other words, we can represent a set $\mathbb{V}_o = \{s_k \mid \exists s_k(j+w) \neq \natural\}$ of all observed stream variable and a set $\mathbb{V}_u = \{s_k \mid s_k(j+w) = \natural\}$ of all unobserved dependent stream variable for all $\langle s_i, s_k, w\rangle \in E$. The set can be expanded to include independent variables as well. For all $s_k \in \mathbb{V}_u$ (resp. $t_k \in \mathbb{V}_u$) that are unobserved, are replaced by:

$$s_k^u(j) = \begin{cases} s_k(j+w) & 0 \leq j+w \leq N \\ \texttt{default} & \text{otherwise} \end{cases}$$

$$t_k^u(j) = \begin{cases} t_k(j+w) & 0 \leq j+w \leq N \\ \texttt{default} & \text{otherwise} \end{cases}$$

and for all $s_k \in \mathbb{V}_o$ (resp. $t_k \in \mathbb{V}_o$) that are observed, are replaced by:

$$s_k^o(j+w) = \texttt{value}$$

$$t_k^o(j+w) = \texttt{value}$$

and there by partially evaluating $s_i(j)$ as

$$s_i(j) = e_i(s_1^o, \cdots, s_n^o, t_1^o, \cdots, t_m^o, s_1^u, \cdots, s_n^u, t_1^u, \cdots, t_m^u)$$

followed by adding the partially evaluated associated equation for $s_i(j)$ to LS. It is to be noted, that a consistent partial distributed stream makes sure that for all s_k (resp. t_k), can only be either observed or unobserved and not both or neither.

Example 3. *Consider the* Lola *specification mentioned below and the stream input of length $N = 6$ divided into two evaluation rounds and $\epsilon = 2$ as shown in Fig. 6.4 with the monitors M_1 and M_2.*

```
input a : uint
input b : uint
output c := ite(a[-1,0] <= b[1, 0], a[1,0], b[-1, 0])
```

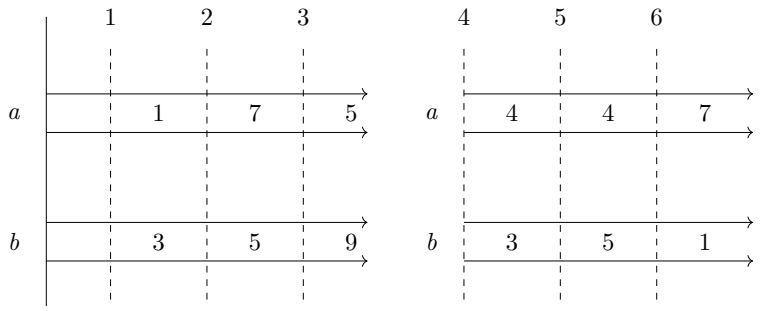


Figure 6.4: Example of generating LS.

The associated equation for the output stream is:

$$c = \begin{cases} \texttt{ite}(0 \leq b(i+1), a(i+1), 0) & i = 1 \\ \texttt{ite}(a(i-1) \leq b(i+1), a(i+1), b(i-1)) & 2 \leq i \leq N-1 \\ \texttt{ite}(a(i-1) \leq 0, 0, b(i-1)) & i = N \end{cases}$$

Let the partial distributed stream read by monitor M_1 *include* $\{a, (1,1), (3,5)\}, \{b, (2,5), (3,9)\}$ *and the partial distributed stream read by monitor* M_2 *include* $\{a, (1,1), (2,7)\}, \{b, (1,3), (3,9)$. *Monitor* M_1 *evaluates* $c(2) = 5$ *and partially evaluates* $c(1)$ *and* $c(3)$. *Thus* $LS_1^1 = \{c(1) = a(2), c(2) = 5, c(3) = ite(a(2) \leq b(4), a(4), 5)\}$. *Monitor* M_2 *partially evaluates all* $c(1)$, $c(2)$ *and* $c(3)$ *and thus* $LS_1^2 = \{c(1) = ite(0 \leq b(2), a(2), 0), c(2) = a(3), c(3) = ite(7 \leq b(4), a(4), b(2))\}$.

Let the partial distributed stream read by monitor M_1 *include* $\{a, (4,4), (5,4)\}, \{b, (4,3), (6,1)\}$ *and the partial distributed stream read by monitor* M_2 *include* $\{a, (5,4), (6,7)\}, \{b, (4,3), (5,5)\}$. *Monitor* M_1 *evaluates* $c(4) = 9$ *and* $c(5) = 3$ *and partially evaluates* $c(6)$. *Thus* $LS_2^1 = \{c(4) = 9, c(5) = 3, c(6) = b(5)\}$. *Monitor* M_2 *evaluates* $c(6) = 5$ *and partially evalues* $c(4)$ *and* $c(5)$ *and thus* $LS_2^2 = \{c(4) = ite(a(3) \leq 5, 4, 9), c(5) = ite(a(4) \leq b(6), 7, 3), c(6) = 5\}$.

It is to be noted, the after the first round of evaluation, the corresponding local states, LS_1^1 *and* LS_1^2 *will be shared which will enable evaluating the output stream for few of the partially evaluated output stream (will be discussed in Section 6.6.1). These will be included in the local state of the following evaluation round.*

Note that generating LS takes into consideration an ordered stream. One where the time of occurrence of events and values are comparable. It can be imagined that generating the same for the distributed system involves generating it for all possible ordering of events. This will be discussed in details in the following sections.

6.5 SMT-based Solution

6.5.1 SMT Entities

SMT entities represent (1) Lola equations, and (2) variables used to represent the distributed stream. Once we have generated a sequence of consistent cuts, we use the laws discussed in Section 6.4, to construct the set of all locally computer or partially computed Lola equations.

Distributed Stream. In our SMT encoding, the set of events, $\mathcal{E}$, is represented by a bit vector, where each bit corresponds to an individual event in the distributed stream, $(\mathcal{E}, \rightsquigarrow)$. The length of the stream under observation is k, which makes $|\mathcal{E}| = k \times |\mathcal{A}|$ and the length of the entire stream is N. We conduct a pre-processing of the distributed stream where we create a $\mathcal{E} \times \mathcal{E}$ matrix, *hbSet* to incorporate the happen-before relations. We populate *hbSet* as $hbSet[\text{e}][\text{f}] = 1$ iff $e \rightsquigarrow f$, else $hbSet[\text{e}][\text{f}] = 0$. In order to map each event to its respective stream, we introduce a function, $\mu : \mathcal{E} \rightarrow \mathcal{A}$.

We introduce a valuation function, $\upsilon : \mathcal{E} \rightarrow \mathsf{T}$ (whatever the type is in the Lola specification), in order to represent the values of the individual events. Due to the partially synchronous assumption of the system, the possible time of occurrence of an event is defined by a function $\delta : \mathcal{E} \rightarrow \mathbb{Z}_{\geq 0}$, where $\forall \alpha(\sigma) \in \mathcal{E}. \exists \sigma' \in [\max\{0, \sigma - \epsilon + 1\}, \min\{\sigma + \epsilon - 1\}, N]. \delta\big(\alpha(\sigma)\big) = \sigma'$. We update the δ function when referring to events on output streams by updating the time synchronization constant to ϵ_M. This accounts for the clock skew between two monitors. Finally, we introduce an uninterpreted function $\rho : \mathbb{Z}_{\geq 0} \rightarrow 2^{\mathcal{E}}$ that identifies a sequence of consistent cuts for computing all possible evaluations of the Lola specification, while satisfying a number of given constrains explained in Section 6.5.2.

6.5.2 SMT Constrains

Once we have defined the necessary SMT entities, we move onto the SMT constraints. We first define the SMT constraints for generating a sequence of consistent cuts, followed by

the ones for evaluating the given LOLA equations φ_α.

Constrains for consistent cuts over ρ: In order to make sure that the uninterpreted function ρ identifies a sequence of consistent cuts, we enforce certain constraints. The first constraint enforces that each element in the range of ρ is in fact a consistent cut:

$$\forall i \in [0,k].\forall e, e' \in \mathcal{E}.\Big((e \rightsquigarrow e') \wedge (e' \in \rho(i))\Big) \rightarrow (e \in \rho(i))$$

Next, we enforce that each successive consistent cut consists of all events included in the previous consistent cut:

$$\forall i \in [0,k-1].\rho(i) \subseteq \rho(i+1)$$

Next, we make sure that the front of each consistent cut constitutes of events with possible time of occurrence in accordance with the semantics of partially-synchronous LOLA:

$$\forall i \in [0,k].\forall e \in \mathsf{front}(\rho(i)).\delta(e) = i$$

Finally, we make sure that every consistent cut consists of events from all streams:

$$\forall i \in [0,k].\forall \alpha \in \mathcal{A}.\exists e \in \mathsf{front}(\rho(i)).\mu(e) = \alpha$$

Constrains for Lola specification: These constraints will evaluate the LOLA specifications and will make sure that ρ will not only represent a valid sequence of consistent cuts but also make sure that the sequence of consistent cuts evaluate the LOLA equations, given the stream expressions. As is evident that a distributed system can often evaluate to multiple values at each instance of time. Thus, we would need to check for both satisfaction and violation for logical expressions and evaluate all possible values for arithmetic expressions. Note that monitoring all LOLA specification can be reduce to evaluating expressions that are either logical or arithmetic. Below, we mention the SMT constraint

for evaluating different Lola equations at time instance j:

$$t_i[p,c] = \begin{cases} \upsilon(e) & 0 \leq j+p \leq N \\ c & \text{otherwise} \end{cases} \qquad \Big(\exists e \in \mathsf{front}(\rho(j+p)).(\mu(e) = \alpha_i)\Big)$$

$$s_i(j) = \mathit{true} \quad \mathsf{front}(\rho(j)) \models \varphi_\alpha \qquad \text{(Logical expression, satisfaction)}$$

$$s_i(j) = e_i(\forall e \in \mathsf{front}(\rho(j)).\upsilon(e)) \qquad \text{(Arithmetic expression, evaluation)}$$

The previously evaluated result is included in the SMT instance as a entity and a additional constrain is added that only evaluates to unique value, in order to generate all possible evaluations. The SMT instance returns a satisfiable result iff there exists at-least one unique evaluation of the equation. This is repeated multiple times until we are unable to generate a sequence of consistent cut, given the constraints, i.e., generate unique values. It is to be noted that stream expression of the form $\mathit{ite}(s_i, s_k, s_j)$ can be reduced to a set of expressions where we first evaluate s_i as a logical expression followed by evaluating s_j and s_k accordingly.

6.6 Runtime Verification of Lola specifications

Now that both the rules of generating rewritten Lola equations (Section 6.4) and the working of the SMT encoding (Section 6.5) have been discussed, we can finally bring them together in order to solve the problem introduced in Section 6.3.

6.6.1 Computing LC

Given a set of local states computed from the SMT encoding, each monitor process receives a set of rewritten Lola associated equations, denoted by LS^i_j, where $i \in [1, |\mathcal{M}|]$ for j-th computation round. Our idea to compute LC from these sets is to simply take a

prioritized union of all the associated equations.

$$LC(\Pi_j^i) = \biguplus_{i \in [1, |\mathcal{M}|]} LS_j^i$$

The intuition behind the priority is that an evaluated Lola equation will take precedence over a partially evaluated/unevaluated Lola equation, and two partially-evaluated Lola equation will be combined to form a evaluated or partially evaluated Lola equation. For example, taking the locally computed LS_1^1 and LS_1^2 from Example 3, $LC(LS_1^1, LS_1^2)$ is computed to be $\{c(1) = a(2), c(2) = 5, c(3) = ite(7 \leq b(4), a(4), 5)\}$ at Monitor M_1 and $\{c(1) = 7, c(2) = 5, c(3) = ite(7 \leq b(4), a(4), 5)\}$ at Monitor M_2. Subsequently, $LC(LS_2^1, LS_2^2)$ is computed to be $\{c(4) = 9, c(5) = 3, c(6) = 5\}$ at Monitor M_1 and $\{c(4) = 9, c(5) = 3, c(6) = 5\}$ at Monitor M_2.

6.6.2 Bringing it all Together

As stated in Section 6.3.1, the monitors are decentralized and online. Since, setting up of a SMT instance is costly (as seen in our evaluated results in Section 6.7), we often find it more efficient to evaluate the Lola specification after every k time instance. This reduces the number of computation rounds to $\lceil N/k \rceil$ as well as the number of messages being transmitted over the network as well with an increase to the size of the messages. We update Algorithm 9 to reflect our solution more closely to Algorithm 10.

Each evaluation round starts by reading the r-th partial distributed system which consists of events occurring between the time $\max\{0, (r-1) \times \lceil N/k \rceil\}$ and $\min\{N, r \times \lceil N/k \rceil\}$ (line 3). We assume that the partial distributed system is consistent in accordance with the assumption that each event has been read by atleast one monitor. To account for any concurrency among the events in $(r-1)$-th computation round with that in the r-th computation round, we expand the length by ϵ time, there-by making the length of the r-th computation round, $\max\{0, (r-1) \times \lceil N/k \rceil - \epsilon + 1\}$ and $\min\{N, r \times \lceil N/k \rceil\}$.

Next, we reduce the evaluation of the distributed stream problem into an SMT problem

Algorithm 10: Computation on Monitor M_i.

1: $LS_1^i[0] = \emptyset$
2: **for** $r = 1$ to $\lceil N/k \rceil$ **do**
3: $(\mathcal{E}_i, \rightsquigarrow_i)_r \leftarrow r$-th Consistent partial distributed stream
4: $j = 0$
5: **do**
6: $j = j + 1$
7: $(\alpha_1, \alpha_2, \cdots, \alpha_{|\mathcal{A}|}) \in \mathsf{Sr}(\mathcal{E}_i, \rightsquigarrow_i)$
8: $LS_r^i[j] \leftarrow LS_r^i[j-1] \cup \left[(\alpha_1, \alpha_2, \cdots, \alpha_{|\mathcal{A}|}) \models_S \varphi_\alpha\right]$
9: **while** $(LS_r^i[j] \neq LS_r^i[j-1])$
10: **Send:** broadcasts symbolic view $LS_r^i[j]$
11: **Receive:** $\Pi_r^i \leftarrow \{LS_r^k \mid 1 \leq k \leq \mathcal{M}\}$
12: **Compute:** $LS_{r+1}^i[0] \leftarrow LC(\Pi_r^i)$ ▷ Section 6.6.1
13: **end for**
14: $\mathsf{Result}^i \leftarrow \bigcup_{r \in [1, \lceil N/k \rceil + 1]} LS_r^i[0]$

(line 7). We represent the distributed system using SMT entities and then by the help of SMT constraints, and we evaluate the LOLA specification on the generated sequence of consistent cuts. Each sequence of consistent cut presents a unique ordering of the events which evaluates to a unique value for the stream expression (line 8). This is repeated until we no longer can generate a sequence of consistent cut that evaluates φ_α to unique values (line 9). Both the evaluated as well as partially evaluated results are included in LS as associated LOLA equations. This is followed by the communication phase where each monitor shares its locally computed LS_r^i, for all $i \in [1, |\mathcal{M}|]$ and r evaluation round (line 10-11).

Once, the local states of all the monitors are received, we take a prioritized union of all the associated equation and include them into LS_{r+1}^i set of associated equations (line 12). Following this, the computation shifts to next computation round and the above mentioned steps repeat again. Once we reach the end of the computation, all the evaluated values are contained in Result^i

Lemma 8. *Let $\mathcal{A} = \{S_1, S_2, \cdots, S_n\}$ be a distributed system and φ be an* LOLA *specification. Algorithm 9 terminates when monitoring a terminating distributed system.*

Proof. First, we note that our algorithm is designed for terminating system, also, note that a terminating program only produces a finite distributed computation. In order to prove the lemma, let us assume that the system send out a *stop* signal to all monitor processes when it terminates. When such a signal is received by a monitor, it starts evaluating the

output stream expression using the terminal associated equations. This might arise to two cases. One where all the values required for the evaluation has been observed or one where the values required for the evaluation has not been observed. Although the termination of the monitor process for the first case is trivial, the termination of the monitor process for the second case is dependent upon replacing such unobserved stream value by the default value of the stream expression. Thus, terminating the monitor process eventually. □

Theorem 4. *Algorithm 10 solves the problem stated in Section 6.3.*

Proof. We prove the soundness and correctness of Algorithm 10, by dividing it into three steps. In the first step we prove that given a LOLA specification, φ, the values of the output stream when computed over the distributed computation, $(\mathcal{E}, \rightsquigarrow)$, of length N is the same as when the distributed computation is divided into $\frac{N}{k}$ computation rounds of length k each. Second, we prove that for all time instances the stream equation is eventually evaluated after the communication round. Finally we prove the set of all evaluated result is consistent over all monitors in the system.

Step 1: From our approach, we see that the value of a output stream variable, is evaluated on the events present in the consistent cut with time j. Therefore, we can reduce the proof to:

$$\mathsf{Sr}(\mathcal{E}, \rightsquigarrow) = \mathsf{Sr}(\mathcal{E}_1.\mathcal{E}_2 \cdots \mathcal{E}_{\frac{N}{k}}, \rightsquigarrow)$$

- ($\Rightarrow$) Let $\mathcal{C}_k$ be a consistent cut such that $\mathcal{C}_k$ is in $\mathsf{Sr}(\mathcal{E}, \rightsquigarrow)$, but not in $\mathsf{Sr}(\mathcal{E}_1.\mathcal{E}_2 \cdots \mathcal{E}_{\frac{N}{k}}, \rightsquigarrow$), for some $k \in [0, |\mathcal{E}|]$. This implies that the frontier of $\mathcal{C}_k$, $\mathsf{front}(\mathcal{C}_k) \not\subseteq \mathcal{E}_1$ and $\mathsf{front}(\mathcal{C}_k) \not\subseteq \mathcal{E}_2$ and $\cdots$ and $\mathsf{front}(\mathcal{C}_k) \not\subseteq \mathcal{E}_{\frac{N}{k}}$. However, this is not possible, as according to the computation round construction in Section 6.6.2, there must be a $\mathcal{E}_i$, where $1 \leq i \leq \frac{N}{k}$ such that $\mathsf{front}(\mathcal{C}_k) \subseteq \mathcal{E}_i$. Therefore, such $\mathcal{C}_k$ cannot exist, and $(\alpha_1, \alpha_2, \cdots, \alpha_n) \in \mathsf{Sr}(\mathcal{E}, \rightsquigarrow) \implies (\alpha_1, \alpha_2, \cdots, \alpha_n) \in \mathsf{Sr}(\mathcal{E}_1.\mathcal{E}_2 \cdots \mathcal{E}_{\frac{N}{k}}, \rightsquigarrow)$.
- ($\Leftarrow$) Let $\mathcal{C}_k$ be a consistent cut such that $\mathcal{C}_k$ is in $\mathsf{Sr}(\mathcal{E}_1.\mathcal{E}_2 \cdots \mathcal{E}_{\frac{N}{k}}, \rightsquigarrow)$ but not in $\mathsf{Sr}(\mathcal{E}, \rightsquigarrow)$ for some $k \in [0, |\mathcal{E}|]$. This implies, $\mathsf{front}(\mathcal{C}_k) \subseteq \mathcal{E}_i$ and $\mathsf{front}(\mathcal{C}_k) \not\subseteq \mathcal{E}$ for some $i \in [1, \frac{N}{k}]$. However, this is not possible due to the fact that $\forall i \in [1, \frac{N}{k}].\mathcal{E}_i \subset \mathcal{E}$. There, such $\mathcal{C}_k$ cannot exist, and $(\alpha_1, \alpha_2, \cdots, \alpha_n) \in \mathsf{Sr}(\mathcal{E}_1.\mathcal{E}_2 \cdots \mathcal{E}_{\frac{N}{k}}, \rightsquigarrow) \implies (\alpha_1, \alpha_2, \cdots, \alpha_n) \in \mathsf{Sr}(\mathcal{E}, \rightsquigarrow)$.

Therefore, $\mathsf{Sr}(\mathcal{E}, \rightsquigarrow) = \mathsf{Sr}(\mathcal{E}_1.\mathcal{E}_2 \cdots \mathcal{E}_{\frac{N}{k}}, \rightsquigarrow)$.

Step 2: Given a output stream expression s_i and the dependency graph $G = \langle V, E \rangle$, for each $\langle s_i, s_k, w \rangle \in E$, evaluating the value at time instance $j \in [1, N]$, $\alpha_k(j + w) \neq \natural$ or $\alpha_k(j + w) = \natural$ or $\alpha_k(w + j)$ not observed.

- If $\alpha_k(j + w) \neq \natural$, then we evaluate the stream expression
- If $\alpha_k(j + w) = \natural$, there exists at-least one other monitor where $\alpha_k(j + w) \neq \natural$. Thereby evaluating the stream expression, followed by sharing the the evaluated result with all

other monitors

- If $\alpha_k(w+j)$ not observed, then at some future evaluation round and at some monitor $\alpha_k(j+w) \neq \natural$ and there-by evaluating the stream expression s_i

Similarly, it can be proved for $\langle s_i, t_k, w\rangle \in E$.

Step 3: Each monitor in our approach is fault-proof with communication taking place between all pairs of monitors. We also assume, all messages are eventually received by the monitors. This guarantees all observations are either directly or indirectly read by each monitor.

Together with Step 1 and 2, soundness and correctness of Algorithm 9 is proved. □

Theorem 5. *Let φ be a* Lola *specification and $(\mathcal{E}, \rightsquigarrow)$ be a distributed stream consisting of $|\mathcal{A}|$ streams. The message complexity of Algorithm 10 with $|\mathcal{M}|$ monitors is*

$$O\big(\epsilon^{|\mathcal{A}|}N|\mathcal{M}|^2\big) \quad \Omega(N|\mathcal{M}|^2)$$

Proof. We analyze the complexity of each part of Algorithm 10. The algorithm has a nested loop. The outer loop iterates for $\lceil N/k \rceil$ times, that is $O(N)$. The inner loop is dependent on the number of unique evaluations of the stream expression.

- **Upper-bound** Due to our assumption of partial-synchrony, each event's time of occurrence can be off by ϵ. This makes the maximum number of unique evaluations in the order of $O(\epsilon^{|\mathcal{A}|})$.
- **Lower-bound** The minimum number of unique evaluations is in the order of $\Omega(1)$.

In the communication phase, each monitor sends $|\mathcal{M}|$ messages to all other monitors and receives $|\mathcal{M}|$ messages from all other monitors. That is $|\mathcal{M}|^2$. Hence the message complexity is

$$O\big(\epsilon^{|\mathcal{A}|}N|\mathcal{M}|^2\big) \quad \Omega(N|\mathcal{M}|^2)$$

As a side note, we would like to mention that in case of high readability of the monitors and evaluation of logical expression, the complexity is closer to the lower-bound, whereas with low readability and arithmetic expressions, the complexity is closer to the upper bound.

□

6.7 Case Study and Evaluation

In this section, we analyze our SMT-based decentralized monitoring solution. We note that we are not concerned about data collections, data transfer, etc, as given a distributed setting, the runtime of the actual SMT encoding will be the most dominating aspect of the

monitoring process. We evaluate our proposed solution using traces collected from synthetic experiments (Section 6.7.1) and case studies involving several industrial control systems and RACE dataset (Section 6.7.2). The implementation of our approach can be found on Google Drive(*https://tinyurl.com/2p6ddjnr*).

6.7.1 Synthetic Experiments

Setup

Each experiment consists of two stages: (1) generation of the distributed stream and (2) verification. For data generation, we develop a synthetic program that randomly generates a distributed stream (i.e., the state of the local computation for a set of streams). We assume that streams are of the type *Float*, *Integer* or *Boolean*. For the streams of the type *Float* and *Integer*, the initial value is a random value *s[0]* and we generate the subsequent values by *s[i-1] + N(0, 2)*, for all $i \geq 1$. We also make sure that the value of a stream is always non-negative. On the other hand, for streams of the type *Boolean*, we start with either *true* or *false* and then for the subsequent values, we stay at the same value or alter using a Bernoulli distribution of $B(0.8)$, where a *true* signifies the same value and a *false* denotes a change in value.

For the monitor, we study the approach using Bernoulli distribution $B(0.2)$, $B(0.5)$ and $B(0.8)$ as the read distribution of the events. A higher readability offers each event to be read by higher number of monitors. We also make sure that each event is read by at least one monitor in accordance with the proposed approach. To test the approach with respect to different types of stream expression, we use the following arithmetic and logical expressions.

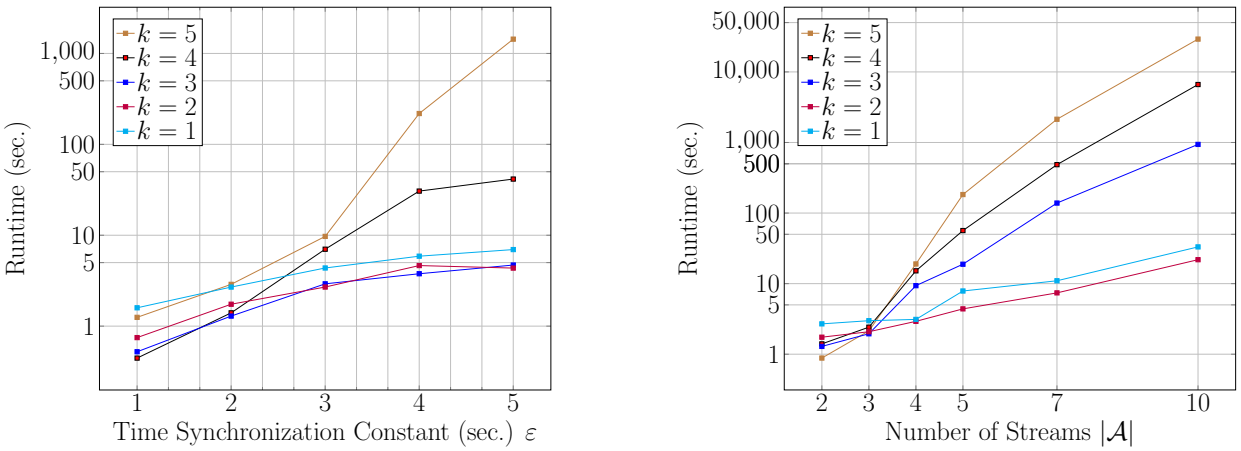
```
input a1 : uint
input a2 : uint
output arithExp := a1 + a2
output logicExp := (a1 > 2) && (a2 < 8)
```

Result - Analysis

We study different parameters and analyze how it effects the runtime and the message size in our approach. All experiments were conducted on a 2017 MacBook Pro with 3.5GHz Dual-Core Intel core i7 processor and 16GB, 2133 MHz LPDDR3 RAM. Unless specified otherwise all experiments consider number of streams, $|\mathcal{A}| = 3$, time synchronization constant, $\epsilon_M = \epsilon = 3s$, number of monitors same as the number of streams, computation length, $N = 100$, with $k = 3$ with a read distribution $B(0.8)$.

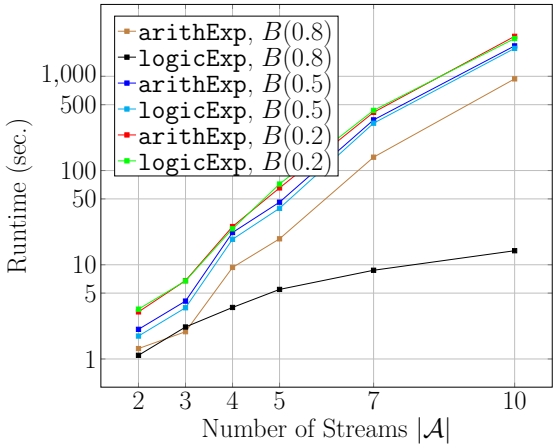
Time Synchronization Constant. Increasing the value of the time synchronization constant ϵ, increases the possible number of concurrent events that needs to be considered. This increases the complexity of evaluating the Lola specification and there-by increasing the runtime of the algorithm. In addition to this, higher number of ϵ corresponds to higher number of possible streams that needs to be considered. We observe that the runtime increases exponentially with increasing the value of ϵ in Fig. 6.5a, as expected. An interesting observation is that with increasing the value of k, the runtime increases at a higher rate until it reaches the threshold where $k = \epsilon$. This is due to the fact, that the number of streams to be considered increases exponentially but ultimately gets bounded by the number of events present in the computation.

Increasing the value of the time synchronization constant is also directly proportional to the number of evaluated results at each instance of time. This is because, each stream corresponds to a unique value being evaluated until it gets bounded by the total number of possible evaluations, as can be seen in Fig. 6.6a. However, comparing Figs. 6.5a and 6.6a, we see that the runtime increases at a faster rate to the size of the message. This owes to the fact that initially a SMT instance evaluates unique values at all instance of time. However, as we start reaching all possible evaluations for certain instance of time, only a fraction of the total time instance evaluates to unique values. This is the reason behind the size of the message reaching its threshold faster than the runtime of the monitor.



(a) Epsilon

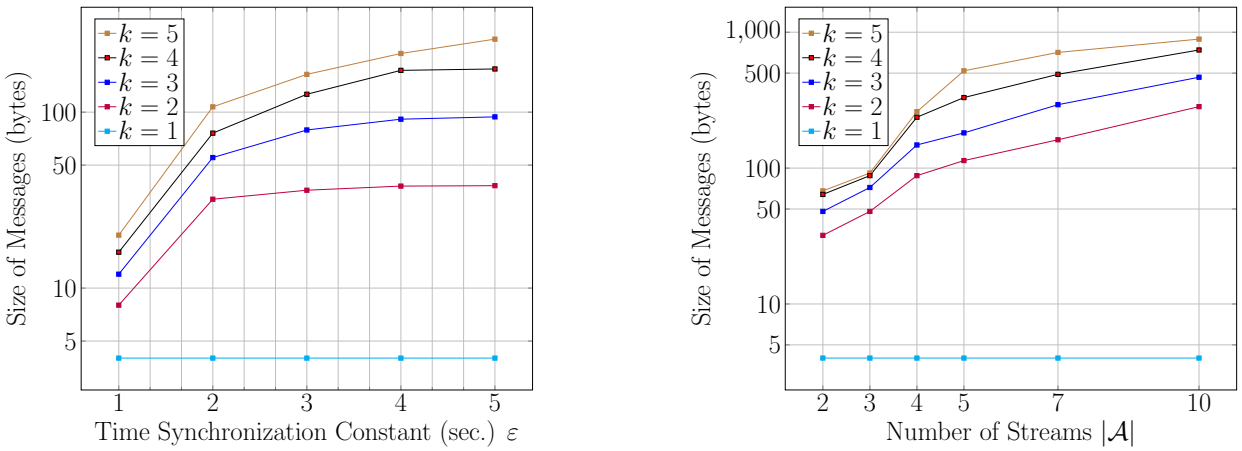
(b) Number of Streams



(c) Different Lola Specification

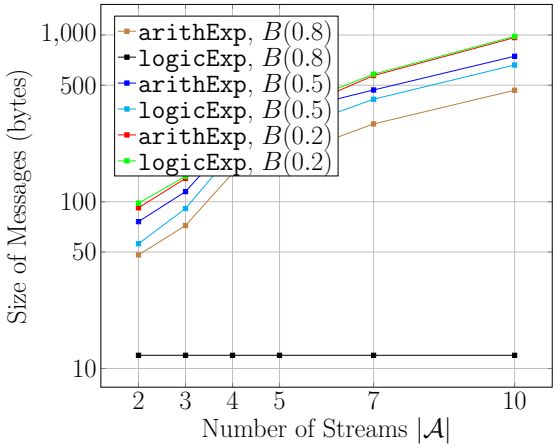
Figure 6.5: Impact of different parameters on runtime for synthetic data.

Type of Stream Expression. Stream expressions can be divided into two major types, one consisting of arithmetic operations and the other involving logical operations. Arithmetic operations can evaluate to values in the order of $O(|\mathcal{A}|.\epsilon)$, where as logical operations can only evaluate to either *true* or *false*. When the monitors have high readability of the distributed stream, it is mostly the case, that the monitor was able to evaluate the stream expression. Thus, we observe in Fig. 6.5c that the runtime grows exponentially for evaluating arithmetic expressions but is linear for logical expressions. However, with low readability of the computation, irrespective of the type of expression, both takes exponential time since neither can completely evaluate the stream expression. So, each monitor has to generate all possible streams.



(a) Epsilon

(b) Number of Streams



(c) Different Lola Specification

Figure 6.6: Impact of different parameters on message size for synthetic data.

Similarly, for high readability and logical expressions, the message size is constant given the monitor was was able to evaluate the stream expression. However with low readability, message size for evaluating logical expressions matches with that of its arithmetic counterpart. This can be seen in Fig. 6.6c and is due to the fact, that with low readability, complete evaluation of the expression is not possible at a monitor and thus needs to send the rewritten expression with the values observed to the other monitors where it will be evaluated.

Number of Streams. As the number of streams increases, the number of events increase linearly and thereby making exponential increase in the number of possible synchronous streams (due to interleavings). This can be seen in Fig. 6.5b, where the runtime increases

exponentially with increase in the number of streams in the distributed stream. Similarly, in Fig. 6.6b, increase in the number of streams linearly effects the number of unique values that the LOLA expression can evaluate to and there-by increasing the size of the message.

6.7.2 Case Studies: Decentralized ICS and Flight Control RV

We put our runtime verification approach to the test with respect to several industrial control system datasets that includes data generated by a (1) Secure Water Treatment plant (SWaT) [GAJM17], comprising of six processes, corresponding to different physical and control components; (2) a Power Distribution system [SLX16] that includes readings from four phaser measurement unit (PMU) that measures the electric waves on an electric grid, and (3) a Gas Distribution system [BBHB13] that includes messages to and from the PLC. In these ICS, we monitor for correctness of system properties. Additionally we monitor for mutual separation between all pairs of aircraft in RACE [MGS19] dataset, that consists of SBS messages from aircrafts.

SWaT Dataset Secure Water Treatment (SWaT) [GAJM17] utilizes a fully operational scaled down water treatment plant with a small footprint, producing 5 gallons/minute of doubly filtered water. It comprises of six main processes corresponding to the physical and control components of the water treatment facility. It starts from process P1 where it takes raw water and stores it in a tank. It is then passed through the pre-treatment process, P2, where the quality of the water is assessed and maintained through chemical dosing. The water then reaches P3 where undesirable materials are removed using fine filtration membranes. Any remaining chlorine is destroyed in the dechlorination process in P4 and the water is then pumped into the Reverse Osmosis system (P5) to reduce inorganic impurities. Finally in P6, water from the RO system is stored ready for distribution.

The dataset classifies different attack on the system into four types, based on the point and stage of the attack: Single Stage-Single Point, Single Stage-Multi Point, Multi Stage-Single Point and Multi Stage-Multi Point. We for the scope of this paper are the most

interested in the attacks either covering multiple stages or multiple points. Few of the LOLA specifications used are listed below.

```
input FIT-101 : uint
input MV-101 : bool
input LIT-101 : uint
input P-101 : bool
input FIT-201 : uint
output inflowCorr := ite(MV-101 == true, FIT-101 > 0, FIT-101 == 0)
output outflowCorr := ite(P-101 == true, FIT-201 > 0, FIT-201 == 0)
output tankCorr := ite(MV-101 == true || P-101 == true, LIT-101 = LIT-101[-1, 0]
    + FIT-101[-1, 0] - FIT-201[-1, 0])
```

where *FIT-101* is the flow meter, measuring inflow into raw water tank, *MV-101* is a motorized valve that controls water flow to the raw water tank, *LIT-101* is the level transmitter of the raw water tank, *P-101* is a pump that pumps water from raw water tank to the second stage and *FIT-201* is the flow transmitter for the control dosing pumps. The above LOLA specification checks the correctness of the inflow meter and valve pair (resp. outflow meter and pump pair) in *inflowCorr* (resp. *outflowCorr*) output expressions. On the other hand, *tankCorr* checks if the water level in the tank adds up to the in-flow and out-flow meters.

```
input AIT-201 : uint
input AIT-202 : uint
input AIT-203 : uint
output numObv := numObv[-1, 0] + 1
output NaClAvg := (NaClAvg[-1, 0] * numObv[-1, 0] + AIT-201) / numObv
output HClAvg := (HClAvg[-1, 0] * numObv[-1, 0] + AIT-202) / numObv
output NaOClAvg := (NaOClAvg[-1, 0] * numObv[-1, 0] + AIT-202) / numObv
```

where *AIT-201*, *AIT-202* and *AIT-203* represents the NaCl, HCl and NaOCl levels in water respectively and *NaClAvg*, *HClAvg* and *NaOClAvg* keeps a track of the average levels of the corresponding chemicals in the water, where as *numObv* keeps a track of the total number of observations read by the monitor.

Power System Attack Dataset Power System Attack Dataset [SLX16] consists of three datasets developed by Mississippi State University and Oak Ridge National Laboratory. It consists of readings from four phaser measurement unit (PMU) or synchrophasor that measures the electric waves on an electric grid. Each PMU measures 29 features consisting of voltage phase angle, voltage phase magnitude, current phase angle, current phase magnitude for Phase A-C, Pos., Neg. and Zero. It also measures the frequency for relays, the frequency delta for relay, status flag for relays, etc. Apart from these 116 PMU measurements, the dataset also consists of 12 control panel logs, snort alerts and relay logs of the 4 PMU.

The dataset classifies into either natural event/no event or an attack event. Few of the LOLA specifications used are listed below. The first attempts to detect a single-line-to-ground (1LG) fault.

```
input R1-I : float
input R2-I : float
input R1-Relay : bool
input R2-Relay : bool
output R1-I-low := R1-I < 200
output R1-I-high := R1-I > 1000
output R2-I-low := R2-I < 200
output R2-I-high := R2-I > 1000
output 1LG := R1-I-high && R2-I-high && R1-Relay[+2, false] && R2-Relay[+2,
    false] && R1-I-low[+4, false] && R2-I-low[+4, false]
```

where *R1-I* and *R2-I* represents the current measured at the R1 and R2 PMU respectively. Additionally, *R1-Relay* and *R2-Relay* keeps a track of the state of the corresponding relay. As a part of the 1LG attack detection, we first categorize the current measured as either low or high depending upon the amount of the current measured. We categorize an attack as 1LG if both R1 and R2 detects high current flowing followed by the relay tripping followed by low current.

```
input R1-PA1-I : float
input R1-PA2-I : float
input R1-PA3-I : float
```

```
output phaseBal := (R1-PA1-I - R1-PA2-I) <= 10 && (R1-PA2-I - R1-PA3-I) <= 10 &&
    (R1-PA3-I - R1-PA1-I) <= 10
```

where *R1-PA1-I*, *R1-PA2-I* and *R1-PA3-I* are the amount of current measured by R1 PMU at Phase A, B and C respectively. The monitor helps us to check if the load on three phases are equally balanced.

Gas Distribution System Gas Distributed System [BBHB13] is a collection of labeled Remote Terminal Unit (RTU) telemetry streams from a Gas pipeline system in Mississippi State University's Critical Infrastructure Protection Center with collaboration from Oak Ridge National Laboratory. The telemetry streams includes messages to and from the Programmable Logic Controller (PLC) under normal operations and attacks involving command injection and data injection attack. The feature set includes the pipeline pressure, setpoint value, command data from the PLC, response to the PLC and the state of the solenoid, pump and the Remote Terminal Unit (RTU) auto-control.

One of the most common data injection attack is *Fast Change*. Here the reported pipeline pressure value is successively varied to create a lack of confidence in the correct operation of the system. The corresponding LOLA specification monitoring against such attack is mentioned below:

```
input PipePress : float
input response : bool
output fastChange := ite(response, mod(PipePress - PipePress[-1, 1000]) <= 10,
    true)
```

where *PipePress* records the measured pipeline pressure and *response* is a flag variable signifying a message to the PLC. Here we consider the default pressure is 1000 psi and the permitted pressure change per unit time is 10 psi (these can be changed according to the demands of the system). Similarly we have LOLA specifications monitoring other data injection attacks such as *Value Wave Injection*, *Setpoint Value Injection*, *Single Data Injection*, etc. and command injection attacks such as *Illegal Setpoint*, *Illegal PID Command*,

etc.

RACE Dataset Runtime for Airspace Concept Evaluation (RACE) [MGS19] is a framework developed by NASA that is used to build an event based, reactive airspace simulation. We use a dataset developed using this RACE framework. This dataset contains three sets of data collected on three different days. Each set was recorded at around 37 N Latitude and 121 W Longitude. The dataset includes all 8 types of messages being sent by the SBS unit by using a Telnet application to listen to port 30003, but we only use the messages with ID 'MSG 3' which is the Airborne Position Message and includes a flight's latitude, longitude and altitude using which we verify the mutual separation of all pairs of aircraft. Furthermore, calculating the distance between two coordinates is computationally expensive, as we need to factor in parameters such as curvature of the earth. In order to speed up distance related calculations, we consider a constant latitude distance of 111.2km and longitude distance of 87.62km, at the cost of a negligible error margin. The corresponding Lola specification is mentioned below:

```
input flight1_alt : float
input flight1_lat : float
input flight1_lon : float
input flight2_alt : float
input flight2_lat : float
input flight2_lon : float
output distDiff := sqrt(pow(flight1_alt - flight2_alt, 2) + pow((flight1_lon -
    flight2_lon)*87620, 2) + pow((flight1_lat - flight2_lat)*111200, 2))
output check := distDiff > 500
```

For our setting we assume, each component has its own asynchronous local clock, with varying time synchronization constant. Next we discuss the results of verifying different ICS with respect to Lola specifications.

Result Analysis We employed same number of monitors as the number of components for each of the ICS case-studies and divided the entire airspace into 9 different ones with

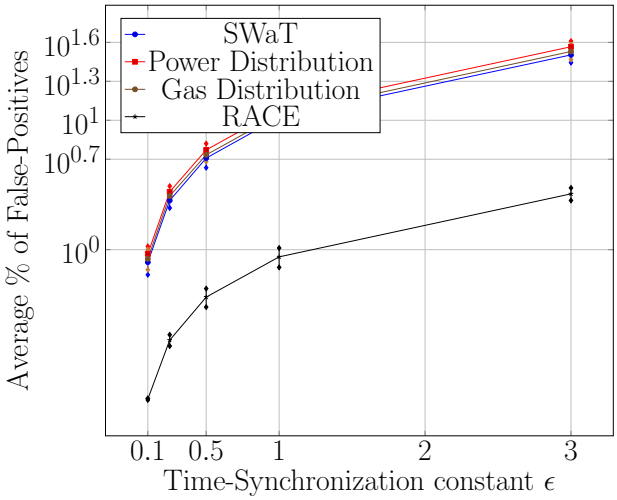


Figure 6.7: False-Positives for ICS Case-Studies.

one monitor responsible for each. We observe that our approach does not report satisfaction of system property when there has been an attack on the system in reality (false-negative). However, due to the assumption of partial-synchrony among the components, our approach may report false positives, i.e., it reports a violation of the system property even when there was no attack on the system. As can be seen in Fig. 6.7, with decreasing time synchronization constant, the number of false-positives reduce as well. This is due to the fact that with decreasing ϵ, less events are considered to be concurrent by the monitors. This makes the partial-ordering of events as observed by the monitor closer to the actual-ordering of events taking place in the system.

We get significantly better result for aircraft monitoring with fewer false-positives compared to the other dataset. This can be attributed towards Air Traffic Controllers maintaining greater separation between two aircrafts than the minimum that is recommended. As part of our monitoring of other ICS, we would like to report that our monitoring approach could successfully detect several attacks which includes underflow and overflow of tank and sudden change in quality of water in SWaT, differentiate between manual tripping of the breaker from the breaker being tripped due to a short-circuit in Power Distribution and Single-point data injection in Gas distribution.

6.8 Summary and Limitation

In this chapter, we studied distributed runtime verification w.r.t. to the popular stream-based specification language Lola. We propose a online decentralized monitoring approach where each monitor takes a set of associated Lola specification and a partial distributed stream as input. By assuming partial synchrony among all streams and by reducing the verification problem into an SMT problem, we were able to reduce the complexity of our approach where it is no longer dependent on the time synchronization constant. We also conducted extensive synthetic experiments, verified system properties of large Industrial Control Systems and airspace monitoring of SBS messages. Comparing to machine learning-based approaches to verify the correctness of these system, our approach was able to produce sound and correct results with deterministic guarantees. As a better practice, one can also use our RV approach along with machine-learning based during training or as a safety net when detecting system violations.

For future work, we plan to study monitoring of distributed systems where monitors themselves are vulnerable to faults such as crash and Byzantine faults. This will let us design a technique with faults and vulnerabilities mimicking a real life monitoring system and thereby expanding the reach and application of runtime verification on more real-life safety critical systems.

Chapter 7

Related Work

This section aims to summarize the in-exhaustive quantity of work done previously that has influenced our work, beginning at the origins of distributed monitoring, followed by runtime verification of un-timed and timed logic with different applications and finally with robust and sound verification approaches even with faulty monitors.

7.1 Lattice-theoretic Distributed Monitoring

Predicate detection is the problem of identifying states of a distributed computation that satisfy a predicate [Gar02, SS95]. The problem is in general NP-complete [MG01]. *Computation slicing* [MG05] is a technique for reducing the size of the computation and, hence, the number of global state to be analyzed for detecting a predicate. The slice of a computation with respect to a predicate is the sub-computation satisfying the following two conditions: (1) it contains all global states for which the predicate evaluates to true, and (2) among all computations that satisfy the first condition, it contains the least number of consistent cuts. In [MG05], the authors propose an algorithm for detecting *regular* predicates. This idea is then extended to a full blown distributed algorithm for distributed monitoring [CGNM13]. One shortcoming of this line work is that it does not address monitoring properties with temporal requirements. This shortcoming is partially addressed in [OG07] for a fragment of temporal operators. In [MB15], the authors propose the first

sound method for runtime verification of asynchronous distributed programs for the 3-valued semantics of LTL specifications defined over the global state of the program. In the proposed setting, monitors are not subject to faults. The technique for evaluating LTL properties is inspired by distributed computation slicing described above. The monitoring technique is fully decentralized. LTL formulas in this work are in terms of conjunctive predicates.

Lattice-based techniques may suffer from the existence of too many concurrent states. To tackle this problem in [YNV+16], the authors propose an algorithm and analytical bounds if a combination of logical and physical clocks (called *hybrid* clocks) are used. This method is enriched with SAT solving techniques in [VYK+17]. Other SMT-based predicate detection solutions include [PMSP20], where the authors build a tool SPIDER to detect race conditions in distributed system.

In [VKTA20], the authors propose a two-layered monitoring algorithm that combines the algorithm that uses Hybrid Logical Clock (HLC) that is dependent on a parameter γ with a monitoring algorithm that uses SMT solvers to perform predicate detection. This two layered monitoring algorithm eliminates all false positives and, depending on γ, many or all false negatives are eliminated at a reduced cost. This makes monitoring a much faster procedure. A completely SMT-based approach is proposed with a focus on cyber-physical systems in [MBAB21], where the authors focus on detecting violations of predicates over distributed continuous-time and continuous-valued signals from cyber physical systems.

7.2 Monitoring Distributed System

Monitoring distributed system can be broadly classified using the presence or absence of a global common clock among the processes. The algorithm in [BF16b] for monitoring synchronous distributed systems with respect to LTL formulas is designed such that satisfaction or violation of specifications can be detected by local monitors alone. The framework employs disjoint alphabet for each process in the system. Thus, a local monitor in [BF16b] can only evaluate subformulas that include its own propositions and if the

subformula contains propositions of other processes, it sends a proof obligation to the corresponding monitor to resolve the obligation. This technique is called formula *progression*. This implies that if multiple proof obligations exist, the formula needs to be progressed by multiple monitors in a sequence of communication rounds. Each round may increase the size of the formula to remember what happened in the past. A similar progression-based verification approach is studied for decentralized monitoring in [BF12]. A internet-of-things based application of the above approach is discussed in [EHF22].

In [CF16], the authors introduce a way of organizing sub-monitors for LTL subformulas in a synchronous distributed system, called *choreography*. In particular, the monitors are organized as a tree across the distributed system, and each child feeds intermediate results to its parent in a manner similar to diffusing computation. They formalize choreography-based decentralized monitoring by showing how to synthesize a network from an `LTL` formula, and give a decentralized monitoring algorithm working on top of an `LTL` network.

Verification is usually deployed for remote systems where the communication may be unreliable. To study the effect of unreliable channels on monitoring, the authors in [KHF19] start off by describing different types of mutations that may be introduced to an execution trace and examine their effects on program monitoring. They also propose a fixed-parameter tractable algorithm for determining the immunity of a finite automaton to a trace mutation and show how it can be used to classify ω-regular properties as monitor-able over channels with that mutation. An ω-regular property is one that generalizes the definition of regular properties to infinite words.

In [EHF18], the authors give a comprehensive overview of monitoring multi-threaded system or more specifically the added challenges of monitoring asynchronous distributed system. Some of the solutions discussed include Java PathExplorer (JPaX) [HR04], which is a tool designed for multi threaded programs. It uses byte code-level automata-based instrumentation to detect both race conditions and deadlocks in a multi threaded program execution.

To include a more wide range of applications for runtime verification, various stream runtime verification logic and algorithm has been developed. Some of the notable ones are Striver [LSS+18] and TeSSLa [LSS+18]. In stream runtime verification, the monitor receives a stream of rich data from the processes and the specifications include not only predicates but aggregate functions, like average, mean, medium, etc. In [Sá21], the authors discuss stream runtime verification for both synchronous and asynchronous system.

7.3 Monitoring Time-bounded Specification

Time-bounded logic can be of two types depending upon the assumption of discrete or continuous time. For discrete (non-negative integers) time we have Metric Temporal Logic (`MTL`) and for continuous (non-negative real) time we have Signal Temporal Logic (`STL`). In [WOH19], the authors present a monitoring algorithm that does not store any information about the observed trace but is able to evaluate both future and past time logic of `MTL`. They term the approach as *"resolve the past and derive the future"*. It involves the `MTL` formula to be transformed into an equivalent formula with the property that it has no past time operator rooted subformulas which are not guarded by other temporal operators for past time sub-formula. On the other hand, for future time logic, it involves the `MTL` formula to be transformed into a new `MTL` formula with the property that the current formula holds before processing the newly received event if and only if the derived formula holds after processing the event. This is very close to the concept of progression we use in our monitoring algorithm but here in [WOH19], the authors work with a synchronous system.

Other notable works for monitoring `MTL` formula includes [WOH19, FP07, BKMZ15, BKM10]. The authors in [BKMZ15, BKM10] extend the general monitoring `MTL` formulas to include a more expressive Metric First-Order Temporal Properties. It includes first order extensions of quantifying the trace where the sub-formula should hold. In [WOH19], the authors were able to introduce a trace-length independent monitoring procedure for an extension of `MTL` with the same expressiveness of that of *Monadic First-Order Logic of*

Order and Metric (FO[¡, +1]). Domain specific monitoring of time-bounded properties includes security vulnerabilities posed by blockchains in [AGCC+20,AEP21,APSS21,CPR18, PZS+18]. All of these work involve vulnerabilities of transactions involving smart contract. However, they are not distributed in the sense, they do not involve transactions over multiple blockchains.

In order to monitor a system where components might crash or network failures can occur the authors in [BKZ15] propose a 3-valued semantics of `MTL` based runtime verification approach. The monitor uses these timestamps of the events to determine the elapsed time between observations to check whether real-time constraints are met. To efficiently resolve knowledge gaps and to compute verdicts, each monitor maintains a AND-OR graph where the edges express constraints for assigning a boolean value to a node. If a monitor receives additional information about the system behavior, it updates its graph structure by adding and deleting nodes and edges, based on the message received.

For monitoring of dense time bounded (signal) temporal logic (`STL`), authors in [DFM13, DDG+17] propose monitoring approaches curated for use in cyber-physical systems. In [DDG+17], the authors formalize a semantics for robust online monitoring of partial traces, i.e., traces for which there might not be enough data to decide the Boolean satisfaction and violation. The approach involves around given a trace and a signal property, it maps them to an interval (l, v), where l is the greatest lower bound and v is the lowest upper bound on the quantitative semantics of the trace.

Authors of [LSS+19], bring runtime verification of incomplete traces to not only monitoring data streams but also to timed events. They use TeSSLa [LSS+18] as the specification language for non-synchronized timed event streams and defines an abstract event streams representing the set of all possible traces that could have occurred during the gaps in the input trace. They work under the assumption of (1) for events with imprecise values the monitor has an idea about the range of values and (2) for data losses the monitor is able to know the range of when it stopped getting information and when the trace becomes

reliable again. In order to solve the problem, the authors extend the semantics of TeSSLa to incorporate incomplete traces and define a abstraction based sliding window to monitor the traces.

7.4 Runtime Verification of Hyperproperties

When monitoring for information flow security policies, requires relation to be expressed between multiple traces [CS08]. Thus, specifications are represented using Hyper Linear Temporal Logic (`HyperLTL`) [CFK$^+$14, FRS15]. Runtime Verification of `HyperLTL` specifications were first discussed in [BF16b] where they introduce the finite-trace semantics of `HyperLTL`. Later in [AB16], the authors introduce a runtime verification technique for a subclass of hyper-properties which deals with k-safety properties. A property is called k-safety, when the size of each finite set is at most k, it results in a k-safety hyperproperty. This is essential for monitoring a system w.r.t. hyperproperties since a system often generates infinite number of traces and monitoring such a set of traces becomes difficult. The proposed monitoring approach involves introducing a procedure that aggregates a runtime progression logic and computes verdicts using a $\texttt{LTL}_3$ monitor.

In [BF18], the authors discuss the main challenges in verifying a distributed system where the specifications is mentioned in `HyperLTL`. The added challenge in verification of hyperproperties is that the monitor when verifying hyperproperties repeatedly model checks the growing Kripke structure compared to the monitor tracking the state of the specification when verifying trace properties. The authors report that in case of tree-shaped Kripke structure, the complexity is L-complete independent of the number of quantifier alternations in `HyperLTL` formula. On the other hand for acyclic Kripke structures, the complexity is PSPACE-complete in the level of the polynomial hierarchy that corresponds to the number of quantifier alternations. However, they also report that the combined complexity in the size of the Kripke structure and the length of the HyperLTL formula is PSPACE-complete for both trees and acyclic Kripke structures. Thereby coming to the final conclusion that

the size and shape of both the Kripke structure and the formula have a significant impact on the complexity of the model checking problem.

A number of versatile runtime verification approaches for different cases and system specifications are presented in [BSB17, FHST17, PSS18]. As mentioned before that monitoring hyperproperties involve monitors storing previously seen traces and this makes the monitor to become slower and slower, and there comes a time when it inevitably runs out of memory. In [FHST17], the authors present techniques that reduce the set of traces that new traces must be compared against to a minimal subset. The techniques include exploiting properties of specifications such as reflexivity, symmetry, and transitivity, to reduce the number of comparisons. In contrast the authors in citebsb17, present a rewriting-based technique for runtime verification for alternation-free `HyperLTL`. The distinguishing feature of this proposed technique is that it is independent of the number of trace quantifiers in a given `HyperLTL` formula. Authors in [PSS18], achieve efficient monitoring by reducing the hyperproperty into trace properties for deterministic system by extracting the characteristic predicate for a given hyperproperty, and provide a parametric monitor taking the extracted predicate as parameter.

7.5 Fault-tolerant Distributed Monitoring

In [FRT14, FRRT14] the authors show that if runtime monitors employ enough number of *opinions* (instead of the conventional binary valuations), then it is possible to monitor distributed tasks in a consistent manner. Building on the work in [FRT13, FRT14, FRRT14], the authors in [BFR$^+$16] show that employing the four-valued `LTL` [BLS10a] will result in inconsistent distributed monitoring for some formulas. They subsequently introduce a family of logics, called LTL_{2k+4}, that refines the 4-valued LTL by incorporating $2k+4$ truth values, for each $k \geq 0$. The truth values of LTL_{2k+4} can be effectively used by each monitor to reach a consistent global set of verdicts for each given formula, provided k is sufficiently large.

The authors in [FRT20], dive into investigating the factors responsible for the size of

a decentralized monitoring approach where the monitors are susceptible to faults. They consider a static system, one where each monitor reads an observation of the system as input, exchanges information followed by performing individual computation and eventually outputting the verdict that reflects the perception of validity of the system state and there is not change in state of the system while all this is happening. The main inference of this approach was that the authors were able to find a tight lower bound on the size of the opinion which was dependent on the language of the property being monitored against. They also go on to prove that for every $n \geq 1$, and every $k \in [1, n)$, there exists a language with alternation number k that requires at least k opinions to be monitored with n monitors, and there exists a language with alternation number n that requires at least $n + 1$ opinions to be monitored with n monitors.

7.6 Statistical Model Checking

Statistical Model Checking (SMC) is a method used in the field of formal verification to check the correctness of probabilistic systems. It is particularly useful in systems that involve randomness or uncertainty, such as computer networks, communication protocols, and robotics. The idea behind SMC is to generate a large number of simulation traces of the system under consideration and compare the statistical properties of these traces with the expected behavior of the system. The statistical approach is even applicable for black-box systems, where the behavior is not fully understood or controllable [SVA04]. This is done by defining a set of quantitative properties, such as the probability of a particular event occurring or the expected time taken to reach a particular state, and then comparing the observed statistics with the expected values. SMC can be used to detect various types of errors in probabilistic systems, including deadlocks, livelocks, and other types of performance issues. It is often used in combination with other formal verification techniques, such as model checking and theorem proving, to provide a more comprehensive analysis of the system. Some popular tools for SMC include PRISM [KNP04], Storm [CDS+17], and Maude [CDE+02].

SMC is a useful technique for validating probabilistic systems, and it is increasingly becoming an essential tool in the development of critical systems.

A large number of real-world systems are subject to hard requirements on time. To analyze such systems, researchers model them as timed automata and express requirements using variants of CTL that include operators with resource bounds as parameters. Then, tools and techniques establish worst-case bounds on execution time and resource consumption and perform schedulability analysis. However, there may still be a need to choose among appropriate schedulers, preferring the one that provides the most attractive properties in the expected or average case. Moreover, multiple timed automata (priced timed automata, weighted CTL, etc.) are known to be undecidable [BBM06].

7.7 Beyond Runtime Verification

Looking ahead from runtime verification [Fal10], we have predictive runtime monitoring [JTS21] and runtime enforcement [RKG+19, FMRS18, PFJ+13] of properties. The main differentiating factor these have from runtime verification is that they are able to predict a vulnerability before it has actually happened in the system or are able to enforce a property on the system. In other words, they make sure that the system does not actually reach the vulnerable state. But, in order to do so, information about the working and behavior of the system is required. Thus, for prediction and enforcing specifications, grey (a Deterministic Time Markov Chain or Markov Decision Process model of the system) or white box (implementing code) system is required.

The authors in [SBS+12], introduce a runtime verification approach using state estimation. The proposed approach is based on visualizing event sequences as observation sequences of a Hidden Markov Model (HMM). HMM is used to fill the gaps in observation sequences by extending the classic forward algorithm for HMM state estimation to compute the probability that the property is satisfied by an execution of the program. However the authors in [JTS21], the authors show that this HMM based state estimation does not scale

well due to the combination of nondeterminism and probabilities. They model the system as a Markov Decision Process (MDP) to take into consideration both nondeterminism and probability in the data from imprecise sensors.

To solve the problem of partial or noisy observation, the authors in [CBP21] propose a neural network based predictive monitoring approach. The approach balances between prediction accuracy, to avoid errors and computation efficiency, to support fast execution at runtime. They employ a neural network classifier to predict reachability at any state. They device two solutions, end-to-end where a neural monitor directly operates on the raw observation and the other a two-step approach where a state estimator reconstructs the full history of states and then a classifier maps the sequence to a good/bad label.

Chapter 8

Conclusion and Future Work

In the previous few chapters, we have explored and formed the theoretical and exhaustive practical basis of runtime verification of distributed systems. In this Chapter, we first summarize our contributions and then explore a few of the possible future directions of the research.

8.1 Summary

In Chapter 3, our focus was on distributed runtime monitoring. Both of our proposed techniques take an `LTL` formula and a distributed computation as input, and by assuming a bounded clock skew among all processes, they first chop the computation into multiple segments and then apply either the automata-based monitoring algorithm or progression-based monitoring algorithm implemented as an SMT decision problem in order to verify the correctness of the said formula. We conducted rigorous synthetic experiments, as well as case studies on monitoring consistency conditions in Cassandra and a NASA air traffic control dataset. Our experiments demonstrate up to 35% improvement in performance in our progression-based algorithm over our automata-based algorithm.

In Chapter 4, we study distributed runtime verification. We propose a technique that takes an `MTL` formula and a distributed computation as input. By assuming partial synchrony among all processes, first, we chop the computation into several segments and then apply a

progression-based formula rewriting monitoring algorithm implemented as an SMT decision problem in order to verify the correctness of the distributed system with respect to the formula. We conducted extensive synthetic experiments on traces generated by the tool UPPAAL and a set of blockchain smart contracts.

In Chapter 5, we propose a runtime verification algorithm, where a set of decentralized synchronous monitors that have only a partial view of the underlying system continually evaluate formulas in the linear temporal logic (LTL). We assume that the communication network is a complete graph and each monitor is subject to crash failures. Our algorithm is sound in the sense that upon termination, all local monitors compute the same monitoring verdict as a centralized monitor that can atomically observe the global state of the system. The monitors do not share their full observation of the underlying system. Rather, they communicate a symbolic representation of their partial observations without compromising soundness. This symbolic observation is the set of possible $\mathtt{LTL}_3$ monitor states. Since $\mathtt{LTL}_3$ monitors may not be able to resolve indistinguishable cases due to partial observations, we also proposed an SMT-based transformation algorithm to obtain minimum size $\mathtt{LTL}_3$ monitors. For an LTL formula φ, our SMT-based algorithm only increases the size of an $\mathtt{LTL}_3$ monitor $\mathcal{M}_3^\varphi$ only by a factor of $O(\log|\mathcal{M}_3^\varphi| \cdot |\mathsf{AP}|)$ (communicating explicit observations would require $O(|\mathsf{AP}|)$ bits), where AP is the set of atomic propositions that describe the global state of the underlying system. We put our approach through an extensive number of experiments with varying distributions responsible for modeling monitor crashes, atomic prepositions distributed over the states, and also the partial observation of each monitor. Through extensive experimentation, we learn that limiting the number of rounds to not go till t and communication between monitors now happening after every k states reduce the average number of rounds, and number of messages sent considerably with only the average size of the message going up by a small quantity.

In Chapter 6, we studied distributed runtime verification w.r.t. to the popular stream-based specification language Lola. We propose an online decentralized monitoring approach

where each monitor takes a set of associated LOLA specifications and a partially distributed stream as input. By assuming partial synchrony among all streams and by reducing the verification problem into an SMT problem, we were able to reduce the complexity of our approach where it is no longer dependent on the time synchronization constant. We also conducted extensive synthetic experiments, verified system properties of large Industrial Control Systems, and airspace monitoring of SBS messages. Compared to machine learning-based approaches to verify the correctness of these systems, our approach was able to produce sound and correct results with deterministic guarantees. As a better practice, one can also use our RV approach along with machine-learning based during training or as a safety net when detecting system violations.

8.2 Contributions

The main results of this dissertation in the context of runtime verification of distributed systems are as follows:

- We introduce an automaton and a progression-based approach for monitoring a partially-synchronous distributed system with respect to linear temporal logic. Although both produce sound and complete results, when compared, we find the progression-based approach is often faster than the automata-based one.
- To monitor a partially-synchronous distributed system with respect to time-bounded temporal properties, we introduce progression rules for `MTL` specifications. We also study the behavior of the system to estimate the actual offset distribution between the processes. This enables us to verify the system with a probabilistic guarantee.
- We introduce a fault-tolerant decentralized runtime verification technique for `LTL` specifications and an SMT-based automata extension method to remove the non-determinism in the evaluated verdict due to the monitors only being able to read the partial computation.
- To monitor a partially-synchronous distributed stream, we introduce the semantics of

partially-synchronous LOLA and propose a decentralized stream runtime verification approach where the monitors only read the partially distributed stream.

- We have also studied the effects of our approach on the runtime and memory usage with respect to synthetic as well as a wide range of real-life data.

8.3 Future Work

As introduced in Chapter 1, the future is distributed, with more and more solutions opting for distributed/decentralized solutions. However, checking for completeness, soundness and compliance with system requirements are relatively an untouched part of these solutions. In the next phase of my research, I intend to make Runtime Verification an effective, complete and sound approach for mainly two areas of applications: (1) general distributed systems and (2) AI-safety.

8.3.1 Distributed Systems

Out of all the approaches discussed in this report, we notice a common inference from all of them. For the centralized monitoring approaches, with increase in the number of events as part of the distributed system, the runtime of the approach increases exponentially. However, the SMT-based solution was able to provide great robustness and certification for the correctness of the verdict generated. This limitation is addressed when we decentralized the monitors which comes with a communication overhead. Moreover, real-life systems are often vulnerable to faults like crash-faults, byzantine faults, network faults, etc. Verification approaches should be able to evaluate sound and complete verdicts inspite of these vulnerabilities.

With evolving technologies like blockchain and cyber-physical system (CPS) it waits to be watched what challenges emerging technology puts us with. Technologies like smart contracts in blockchains and CPS can be modeled as a distributed system. However, the sheer size of them makes model checking and testing not ideal approaches for debugging.

Runtime Verification coming out as the obvious choice in these scenarios. Our future work on this topic involves a step towards enforcing properties in real time asynchronous distributed system. As discussed in [Fal10], verification still remains the core part of any enforcement algorithm. Using Hidden Markov Model (HMM) we can appempt to fill up the gaps in the observed behavior of the system and as a result, extend the classic forward algorithm for HMM state estimation. In [BGF18b, BGF18a], we see the authors propose runtime verification approaches using the state estimation/trace abstraction model using HMM at its core. One of the major down side of such approaches is the assumption of a synchronous system, which limits the broad applicability that can be achieved with such an approach for asynchronous system.

A predictive runtime verification or runtime enforcement of a distributed system can only be achieved by having some information about the working of the system. In [JTS21], we observe that the authors model the system as a Markov Decision Process (MDP) which provides a mathematical framework for modeling decision process in situations where the system is both non-deterministic and probabilistic. This property of a MDP can be used to model an asynchronous system with the decision process being the different happen-before relation that is possible given the different interleaving of the events that is possible along with the different probabilities. A future scope of this work includes a learning based predictive runtime verification or runtime enforcement approach that learns the working of the system from initial runs of the system and forms a MDP model. This model can be used in parallel to the trace logs being generated to achieve greater prediction of faults in the system.

8.3.2 AI Safety

We find ourselves in a world where machines and AI are becoming increasingly prevalent and integrated into our daily lives. From automated systems in factories and warehouses to chatbots and virtual assistants on our phones and computers, technology is rapidly advancing

and changing the way we live and work. With the advent of self-driving cars and drones, the possibilities of automation are limitless. With this huge application in often safety-critical systems, a verification or monitoring approach is essential to improve the reliability of the system.

As identified in [NYC15], a Deep Neural Network-based classification approach was able to categorize a noisy image as that of a lion, peacock, starfish, etc. even when a human was not able to categorize them. The reason behind such behavior of a Machine Learning-based approach is the lack of explainability of the approach. Formal Verification acts as a perfect monitor for such systems that makes sure that the system always works within the defined constraints. Applications of formal verification in this space, can be categorized as mainly two different lines of work, (1) safe learning, and (2) monitoring.

As artificial intelligence (AI) systems rapidly increase in size, acquire new capabilities, and are deployed in high-stakes settings, their safety becomes extremely important [FP18]. Ensuring system safety requires more than improving accuracy, efficiency, and scalability: it requires ensuring that systems are robust to extreme events and monitoring them for anomalous and unsafe behavior. While traditional machine learning systems are evaluated pointwise with respect to a fixed test set, such static coverage provides only limited assurance when exposed to unprecedented conditions in high-stakes operating environments. Verifying that learning components of such systems achieve safety guarantees for all possible inputs may be difficult, if not impossible. Instead, a system's safety guarantees will often need to be established with respect to system-generated data from realistic (yet appropriately pessimistic) operating environments. Safety also requires resilience to "unknown unknowns", which necessitates improved methods for monitoring for unexpected environmental hazards or anomalous system behaviors, including during deployment. In some instances, safety may further require new methods for reverse-engineering, inspecting, and interpreting the internal logic of learned models to identify unexpected behavior that could not be found by black-box testing alone, and methods for improving the performance by directly adapting

the systems' internal logic. Whatever the setting, any learning-enabled system's end-to-end safety guarantees must be specified clearly and precisely. Any system claiming to satisfy a safety specification must provide rigorous evidence, through analysis corroborated empirically and/or with mathematical proof.

Applications of learning-based systems include large cyber-physical systems, multi-agent systems, etc. [WOZ+20, CMK+21] Cyber-Physical Systems (CPS) are systems that integrate physical and computational components. They are used in a wide range of applications, such as autonomous vehicles, medical devices, and smart homes. Verification of CPS is crucial to ensure their correctness, safety, and reliability. The verification process involves the use of formal methods, simulation, and testing to validate the correctness of the system's behavior. Multi-Agent Systems (MAS) are systems that consist of multiple agents that interact with each other to achieve a common goal. They are used in a wide range of applications, such as intelligent transportation systems, robotics, and social networks. Verification of MAS is crucial to ensure their correctness, safety, and reliability. The verification process involves the use of formal methods, simulation, and testing to validate the correctness of the system's behavior. Formal methods are commonly used to verify MAS. They involve the use of mathematical techniques to verify the correctness of the system's behavior. Model checking and theorem proving are two common formal methods used to verify MAS.

In all these above-mentioned scenarios, the system is susceptible to change and an unpredictable environment. These changes in the environment often affect the behavior of the system, making runtime verification the obvious choice for maintaining system-level correctness. Neural Network based methods perform statistically better in a predictable environment or when the data is similar to the training data. However, it is the misclassifications in the strong sector, that pose a major vulnerability for the application system. Using runtime verification we are able to check for the correctness of the verdict, given system specifications. Taking an autonomous vehicle as our example, we see that it is able to maneuver the vehicle with high confidence in case of perfect weather conditions.

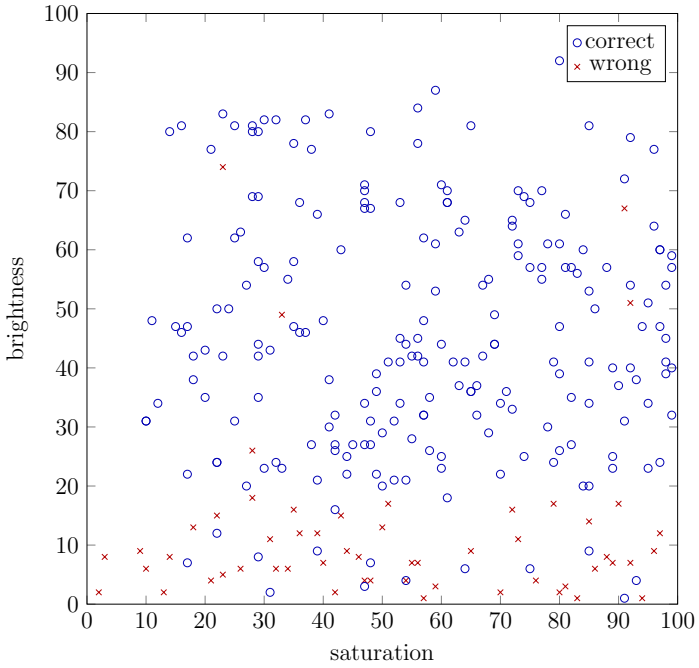


Figure 8.1: Decision boundary plot.

However, in cases where the sun is shining at a low angle, in foggy conditions, low visibility, rainy conditions, etc. any misclassification can have a catastrophic outcome. Runtime verification can act as a helping hand to the already highly efficient machine-learning approach in connecting the dots in these edge cases. This will not only able the system to perform in critical, harsh environments but also make sure the outcomes are with a considerable formal guarantee.

In Figure 8.1, we show a decision boundary plot for a possible classification algorithm. We notice that with low bridgeness of the pictures being classified, the misclassification rate increases. However, the point of worry being the misclassifications that were notices even when the bridgeness was high enough. It is cases like these where formal verification should come in handy. The target bring, to convert the misclassifications for high bridgeness images to be correctly classified.

Additionally, robotics and automated systems use reinforced learning techniques to train a system, with rewards and punishes the system to make it work towards a goal. We would like to design a system that is not too strict towards rewards as this might yield

a not so optimized solution. In the making, it is too lenient and makes the work of the system erroneous. Runtime verification acts as an important meditator, which enables the reinforced learning technique to be neither too strict nor too lenient, and further behaviors are enforced using runtime verification. Consider a garbage-collecting robot, that is responsible for collecting trash around a house and throwing it away in the dumpster. It uses a reinforced learning-based approach where each time the robot picks up and drops the trash in the dumpster, it receives a reward. This strategy often results in a vulnerability where the robot decides to put the trash back from the dumpster onto the floor only to pick it up again. Using runtime verification, we should be able to enforce that once the trash is picked up is not dropped back.

With more and more real-life solutions involving distributed systems, cyber-physical systems, multi-agent systems, etc. the scope of runtime verification poses a very exciting as well as challenging application. This would enable us to design and implement secure and correct-by-design systems in the real life.

BIBLIOGRAPHY

[AB16] Shreya Agrawal and Borzoo Bonakdarpour. Runtime verification of k-safety hyperproperties in hyperltl. In *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*, pages 239–252, 2016.

[ACZ20] Tejasvi Alladi, Vinay Chamola, and Sherali Zeadally. Industrial control systems: Cyberattack trends and countermeasures. *Computer Communications*, 155:1–8, 2020.

[AEP21] Shaun Azzopardi, Joshua Ellul, and Gordon J. Pace. Runtime monitoring processes across blockchains. In Hossein Hojjat and Mieke Massink, editors, *Fundamentals of Software Engineering*, pages 142–156, Cham, 2021. Springer International Publishing.

[AGCC+20] Alberto Aranda García, María-Emilia Cambronero, Christian Colombo, Luis Llana, and Gordon J. Pace. *Runtime Verification of Contracts with Themulus*, pages 231–246. Springer International Publishing, Cham, 2020.

[AH92] Rajeev Alur and Thomas A. Henzinger. Logics and models of real time: A survey. In J. W. de Bakker, C. Huizing, W. P. de Roever, and G. Rozenberg, editors, *Real-Time: Theory in Practice*, pages 74–106, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg.

[AH94] Rajeev Alur and Thomas A. Henzinger. A really temporal logic. *J. ACM*, 41(1):181–203, jan 1994.

[APSS21] Shaun Azzopardi, Gordon Pace, Fernando Schapachnik, and Gerardo Schneider. On the specification and monitoring of timed normative systems. In Lu Feng and Dana Fisman, editors, *Runtime Verification*, pages 81–99, Cham, 2021. Springer International Publishing.

[BBHB13] Justin M. Beaver, Raymond C. Borges-Hink, and Mark A. Buckner. An evaluation of machine learning methods to detect malicious scada communications. In *2013 12th International Conference on Machine Learning and Applications*, volume 2, pages 54–59, 2013.

[BBM06] Patricia Bouyer, Thomas Brihaye, and Nicolas Markey. Improved undecidability results on weighted timed automata. *Information Processing Letters*, 98(5):188–194, 2006.

[BDL04] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on UPPAAL. In *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, pages 200–236, 2004.

[BF12] Andreas Bauer and Yliès Falcone. Decentralised ltl monitoring. In *FM 2012: Formal Methods*, pages 85–100, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[BF16a] Andreas Bauer and Yliès Falcone. Decentralised LTL monitoring. *Formal Methods in System Design*, 48(1):46–93, 2016.

[BF16b] B. Bonakdarpour and B. Finkbeiner. Runtime verification for hyperltl. In *Proceedings of the 16th International Conference on Runtime Verification*, pages 41–45, 2016.

[BF18] Borzoo Bonakdarpour and Bernd Finkbeiner. The complexity of monitoring hyperproperties. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 162–174, 2018.

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

[BGF18a] Reza Babaee, Arie Gurfinkel, and Sebastian Fischmeister. Predictive run-time verification of discrete-time reachability properties in black-box systems using trace-level abstraction and statistical learning. In Christian Colombo and Martin Leucker, editors, *Runtime Verification*, pages 187–204, Cham, 2018. Springer International Publishing.

[BGF18b] Reza Babaee, Arie Gurfinkel, and Sebastian Fischmeister. Prevent: A predictive run-time verification framework using statistical learning. In Einar Broch Johnsen and Ina Schaefer, editors, *Software Engineering and Formal Methods*, pages 205–220, Cham, 2018. Springer International Publishing.

[BHBB+14] Raymond C. Borges Hink, Justin M. Beaver, Mark A. Buckner, Tommy Morris, Uttam Adhikari, and Shengyi Pan. Machine learning for power system disturbance and cyber-attack discrimination. In *2014 7th International Symposium on Resilient Control Systems (ISRCS)*, pages 1–8, 2014.

[BKM10] David Basin, Felix Klaedtke, and Samuel Müller. Monitoring security policies with metric first-order temporal logic. In *Proceedings of the 15th ACM Symposium on Access Control Models and Technologies*, SACMAT '10, page 23–34, New York, NY, USA, 2010. Association for Computing Machinery.

[BKMZ13] David Basin, Felix Klaedtke, Srdjan Marinovic, and Eugen Zălinescu. Monitoring of temporal first-order properties with aggregations. In Axel Legay and Saddek Bensalem, editors, *Runtime Verification*, pages 40–58, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[BKMZ15] David Basin, Felix Klaedtke, Samuel Müller, and Eugen Zălinescu. Monitoring metric first-order temporal properties. *J. ACM*, 62(2), may 2015.

[BKZ12] David Basin, Felix Klaedtke, and Eugen Zălinescu. Algorithms for monitoring real-time properties. In Sarfraz Khurshid and Koushik Sen, editors, *Runtime Verification*, pages 260–275, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[BKZ15] David Basin, Felix Klaedtke, and Eugen Zalinescu. Failure-aware Runtime Verification of Distributed Systems. In Prahladh Harsha and G. Ramalingam, editors, *35th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2015)*, volume 45 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 590–603, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[BLS10a] A. Bauer, M. Leucker, and C. Schallhart. Comparing LTL Semantics for Runtime Verification. *Journal of Logic and Computation*, 20(3):651–674, 2010.

[BLS10b] Andreas Bauer, Martin Leucker, and Christian Schallhart. Comparing ltl semantics for runtime verification. *Journal of Logic and Computation*, 20(3):651–674, 2010.

[BLS11] A. Bauer, M. Leucker, and C. Schallhart. Runtime Verification for LTL and TLTL. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 20(4):14:1–14:64, 2011.

[Bow93] J. Bowen. Formal methods in safety-critical standards. In *Proceedings 1993 Software Engineering Standards Symposium*, pages 168–177, 1993.

[BSB17] Noel Brett, Umair Siddique, and Borzoo Bonakdarpour. Rewriting-based runtime verification for alternation-free hyperltl. In Axel Legay and Tiziana Margaria, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 77–93, Berlin, Heidelberg, 2017. Springer Berlin Heidelberg.

[CBP21] Francesca Cairoli, Luca Bortolussi, and Nicola Paoletti. Neural predictive monitoring under partial observability. In *Runtime Verification: 21st International Conference, RV 2021, Virtual Event, October 11–14, 2021, Proceedings*, page 121–141, Berlin, Heidelberg, 2021. Springer-Verlag.

[CCF+05] Patrick Cousot, Radhia Cousot, Jerôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The astreé analyzer. In Mooly Sagiv, editor, *Programming Languages and Systems*, pages 21–30, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[CDD+15] Cristiano Calcagno, Dino Distefano, Jeremy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. Moving fast with software verification. In Klaus Havelund, Gerard Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods*, pages 3–11, Cham, 2015. Springer International Publishing.

[CDE+02] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. Maude: Specification and programming in rewriting logic. In José Meseguer and Steven Eker, editors, *Rewriting Logic and Its Applications*, pages 76–95. Springer Berlin Heidelberg, 2002.

[CDE+13] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally distributed database. *ACM Trans. Comput. Syst.*, 31(3), aug 2013.

[CDS+17] Edmund M Clarke, Clarke Dehnert, Jeremy Sproston, Helmut Veith, and Zhikun Wang. Storm: a modern probabilistic model checker. *International Journal on Software Tools for Technology Transfer*, 19(2):197–215, 2017.

[CF16] C. Colombo and Y. Falcone. Organising LTL monitors over distributed systems with a global clock. *Formal Methods in System Design*, 49(1-2):109–158, 2016.

[CFK+14] Michael R. Clarkson, Bernd Finkbeiner, Masoud Koleini, Kristopher K. Micinski, Markus N. Rabe, and César Sánchez. Temporal logics for hyperproperties. In Martín Abadi and Steve Kremer, editors, *Principles of Security and Trust*, pages 265–284, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[CGNM13] H. Chauhan, V. K. Garg, A. Natarajan, and N. Mittal. A distributed abstraction algorithm for online predicate detection. In *Proceedings of the 32nd IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 101–110, 2013.

[Cli14] William D Clinger. Advantages of formal specifications. https://course.ccs.neu.edu/cs5500f14/Notes/Communication2/formalSpecs2.html, Fall 2014.

[CLRC17] Julien Cumin, Grégoire Lefebvre, Fano Ramparany, and James L. Crowley. A dataset of routine daily activities in an instrumented home. In Sergio F. Ochoa, Pritpal Singh, and José Bravo, editors, *Ubiquitous Computing and Ambient Intelligence*, pages 413–425, Cham, 2017. Springer International Publishing.

[CMK+21] Anthony Corso, Robert J. Moss, Mark Koren, Ritchie Lee, and Mykel J. Kochenderfer. A survey of algorithms for black-box safety validation of cyber-physical systems. *Journal of Artificial Intelligence Research (JAIR)*, 72(2005.02979):377–428, 2021.

[CPR18] Xiaohong Chen, Daejun Park, and Grigore Roşu. A language-independent approach to smart contract verification. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice*, pages 405–413, Cham, 2018. Springer International Publishing.

[CS08] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. In *2008 21st IEEE Computer Security Foundations Symposium*, pages 51–65, 2008.

[DDG+17] Jyotirmoy V. Deshmukh, Alexandre Donzé, Shromona Ghosh, Xiaoqing Jin, Garvit Juniwal, and Sanjit A. Seshia. Robust online monitoring of signal temporal logic. *Formal Methods in System Design*, 51(1):5–30, 2017.

[DFM13] Alexandre Donzé, Thomas Ferrère, and Oded Maler. Efficient robust monitoring for stl. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*, pages 264–279, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[dMB08] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 337–340, 2008.

[DSS+05] B. D'Angelo, S. Sankaranarayanan, C. Sanchez, W. Robinson, B. Finkbeiner, H.B. Sipma, S. Mehrotra, and Z. Manna. Lola: runtime monitoring of synchronous systems. In *12th International Symposium on Temporal Representation and Reasoning (TIME'05)*, pages 166–174, 2005.

[Dwy20] Matthew Dwyer. Property pattern mappings for ltl. *[Website]*, 2020.

[EHF18] Antoine El-Hokayem and Yliès Falcone. Can we monitor all multithreaded programs? In Christian Colombo and Martin Leucker, editors, *Runtime Verification*, pages 64–89, Cham, 2018. Springer International Publishing.

[EHF22] Antoine El-Hokayem and Yliès Falcone. Bringing runtime verification home: a case study on the hierarchical monitoring of smart homes using decentralized specifications. *International Journal on Software Tools for Technology Transfer*, 24(2):159–181, 2022.

[EP18] Joshua Ellul and Gordon J Pace. Runtime verification of ethereum smart contracts. In *2018 14th European Dependable Computing Conference (EDCC)*, pages 158–163. IEEE, 2018.

[Fal10] Yliès Falcone. You should better enforce than verify. In Howard Barringer, Ylies Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon Pace, Grigore Roşu, Oleg Sokolsky, and Nikolai Tillmann, editors, *Runtime Verification*, pages 89–105, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[FHST17] Bernd Finkbeiner, Christopher Hahn, Marvin Stenger, and Leander Tentrup. Monitoring hyperproperties. In Shuvendu Lahiri and Giles Reger, editors, *Runtime Verification*, pages 190–207, Cham, 2017. Springer International Publishing.

[FMRS18] Yliès Falcone, Leonardo Mariani, Antoine Rollet, and Saikat Saha. *Runtime Failure Prevention and Reaction*, pages 103–134. Springer International Publishing, Cham, 2018.

[FP07] Georgios E. Fainekos and George J. Pappas. Robust sampling for mitl specifications. In Jean-François Raskin and P. S. Thiagarajan, editors, *Formal Modeling and Analysis of Timed Systems*, pages 147–162, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[FP18] Nathan Fulton and André Platzer. Safe reinforcement learning via formal methods: Toward safe control through proof and learning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 32(1), Apr. 2018.

[FRRT14] P. Fraigniaud, S. Rajsbaum, M. Roy, and C. Travers. The opinion number of set-agreement. In *Principles of Distributed Systems - 18th International Conference (OPODIS)*, pages 155–170, 2014.

[FRS15] Bernd Finkbeiner, Markus N. Rabe, and César Sánchez. Algorithms for model checking hyperltl and hyperctl*. In Daniel Kroening and Corina S. Păsăreanu, editors, *Computer Aided Verification*, pages 30–48, Cham, 2015. Springer International Publishing.

[FRT13] P. Fraigniaud, S. Rajsbaum, and C. Travers. Locality and checkability in wait-free computing. *Distributed Computing*, 26(4):223–242, 2013.

[FRT14] P. Fraigniaud, S. Rajsbaum, and C. Travers. On the number of opinions needed for fault-tolerant run-time monitoring in distributed systems. In *Runtime Verification (RV)*, pages 92–107, 2014.

[FRT20] Pierre Fraigniaud, Sergio Rajsbaum, and Corentin Travers. A lower bound on the number of opinions needed for fault-tolerant decentralized run-time monitoring. *Journal of Applied and Computational Topology*, 4(1):141–179, 2020.

[GAJM17] Jonathan Goh, Sridhar Adepu, Khurum Nazir Junejo, and Aditya Mathur. A dataset to support research in the design of secure water treatment systems. In Grigore Havarneanu, Roberto Setola, Hypatia Nassopoulos, and Stephen Wolthusen, editors, *Critical Information Infrastructures Security*, pages 88–99, Cham, 2017. Springer International Publishing.

[Gar02] V. K. Garg. *Elements of distributed computing*. Wiley, 2002.

[GJ79] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, New York, 1979.

[GMB21] Ritam Ganguly, Anik Momtaz, and Borzoo Bonakdarpour. Distributed Runtime Verification Under Partial Synchrony. In *24th International Conference on Principles of Distributed Systems (OPODIS 2020)*, volume 184, pages 20:1–20:17, 2021.

[Her18] Maurice Herlihy. Atomic cross-chain swaps. In *Proceedings of the 2018 ACM symposium on principles of distributed computing*, pages 245–254, 2018.

[HGM20] Marieke Huisman, Dilian Gurov, and Alexander Malkis. Formal methods: From academia to industrial practice. a travel guide, 2020.

[HR01a] K. Havelund and G. Rosu. Monitoring Programs Using Rewriting. In *Automated Software Engineering (ASE)*, pages 135–143, 2001.

[HR01b] Klaus Havelund and Grigore Rosu. Monitoring programs using rewriting. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering*, ASE '01, page 135, USA, 2001. IEEE Computer Society.

[HR04] Klaus Havelund and Grigore Roşu. An overview of the runtime verification tool java pathexplorer. *Formal Methods in System Design*, 24(2):189–215, 2004.

[JTS21] Sebastian Junges, Hazem Torfah, and Sanjit A. Seshia. Runtime monitors for markov decision processes. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification*, pages 553–576, Cham, 2021. Springer International Publishing.

[KDM+14] S. S. Kulkarni, M. Demirbas, D. Madappa, B. Avva, and M. Leone. Logical physical clocks. In *Proceedings of the 18th International Conference on Principles of Distributed Systems*, pages 17–32, 2014.

[KHF19] Sean Kauffman, Klaus Havelund, and Sebastian Fischmeister. Monitorability over unreliable channels. In Bernd Finkbeiner and Leonardo Mariani, editors, *Runtime Verification*, pages 256–272, Cham, 2019. Springer International Publishing.

[KKP+15] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c: A software analysis perspective. *Formal Aspects of Computing*, 27(3):573–609, 2015.

[KNP04] Marta Kwiatkowska, Gethin Norman, and David Parker. PRISM: Probabilistic symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 6(2):128–142, 2004.

[Koy90] R. Koymans. Specifying Real-Time Properties with Metric Temporal Logic. *RealTime Systems*, 2(4):255–299, 1990.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, jul 1978.

[LHJ+14] Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F. Lukman, and Haryadi S. Gunawi. Samc: Semantic-aware model checking for fast discovery of deep bugs in cloud systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, page 399–414, USA, 2014. USENIX Association.

[LLL+17] Haopeng Liu, Guangpu Li, Jeffrey F. Lukman, Jiaxin Li, Shan Lu, Haryadi S. Gunawi, and Chen Tian. Dcatch: Automatically detecting distributed concurrency bugs in cloud systems. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, page 677–691, New York, NY, USA, 2017. Association for Computing Machinery.

[LLLG16] Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, Shan Lu, and Haryadi S. Gunawi. Taxdc: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, page 517–530, New York, NY, USA, 2016. Association for Computing Machinery.

[LM10] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[LPY97] K. G. Larsen, P.Pettersson, and W. Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.

[LSS+18] Martin Leucker, César Sánchez, Torben Scheffel, Malte Schmitz, and Alexander Schramm. Tessla: Runtime verification of non-synchronized real-time streams. In *ACM Symposium on Applied Computing (SAC)*, France, 04/2018 2018. ACM, ACM.

[LSS+19] Martin Leucker, César Sánchez, Torben Scheffel, Malte Schmitz, and Daniel Thoma. Runtime verification for timed event streams with partial information. In Bernd Finkbeiner and Leonardo Mariani, editors, *Runtime Verification*, pages 273–291, Cham, 2019. Springer International Publishing.

[Lyn96] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, San Mateo, CA, 1996.

[MB15] M. Mostafa and B. Bonakdarpour. Decentralized runtime verification of LTL specifications in distributed systems. In *Proceedings of the 29th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 494–503, 2015.

[MBAB21] Anik Momtaz, Niraj Basnet, Houssam Abbas, and Borzoo Bonakdarpour. Predicate monitoring in distributed cyber-physical systems. In Lu Feng and Dana Fisman, editors, *Runtime Verification*, pages 3–22, Cham, 2021. Springer International Publishing.

[MG01] Neeraj Mittal and Vijay K. Garg. On detecting global predicates in distributed computations. In *Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS 2001), Phoenix, Arizona, USA, April 16-19, 2001*, pages 3–10, 2001.

[MG05] N. Mittal and V. K. Garg. Techniques and applications of computation slicing. *Distributed Computing*, 17(3):251–277, 2005.

[MGS19] Peter Mehlitz, Dimitra Giannakopoulou, and Nastaran Shafiei. Analyzing airspace data with race. In *2019 IEEE/AIAA 38th Digital Avionics Systems Conference (DASC)*, pages 1–10, 2019.

[Mil10] D. Mills. Network time protocol version 4: Protocol and algorithms specification. RFC 5905, RFC Editor, June 2010.

[MLD+13] Yannick Moy, Emmanuel Ledinot, Hervé Delseny, Virginie Wiels, and Benjamin Monate. Testing or formal verification: Do-178c alternatives and industrial experience. *IEEE Software*, 30(3):50–57, 2013.

[MP79] Z. Manna and A. Pnueli. The modal logic of programs. In *Proceedings of the 6th Colloquium on Automata, Languages and Programming (ICALP)*, pages 385–409, 1979.

[MP95] Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems - Safety*. Springer, 1995.

[Nol13] Tier Nolan. Alt chains and atomic transfers. https://bitcointalk.org/index.php?topic=193281.0, May, 2013. Bitcoin Forum.

[NYC15] Anh Nguyen, Jason Yosinski, and Jeff Clune. Deep neural networks are easily fooled: High confidence predictions for unrecognizable images, 2015.

[OG07] V. A. Ogale and V. K. Garg. Detecting temporal logic predicates on distributed computations. In *Proceedings of the 21st International Symposium on Distributed Computing (DISC)*, pages 420–434, 2007.

[PFJ+13] Srinivas Pinisetty, Yliès Falcone, Thierry Jéron, Hervé Marchand, Antoine Rollet, and Omer Landry Nguena Timo. Runtime enforcement of timed properties. In Shaz Qadeer and Serdar Tasiran, editors, *Runtime Verification*, pages 229–244, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[PMA15a] Shengyi Pan, Thomas Morris, and Uttam Adhikari. Classification of disturbances and cyber-attacks in power systems using heterogeneous time-synchronized data. *IEEE Transactions on Industrial Informatics*, 11(3):650–662, 2015.

[PMA15b] Shengyi Pan, Thomas Morris, and Uttam Adhikari. Developing a hybrid intrusion detection system using data mining for power systems. *IEEE Transactions on Smart Grid*, 6(6):3104–3113, 2015.

[PMSP20] João Carlos Pereira, Nuno Machado, and Jorge Sousa Pinto. Testing for race conditions in distributed systems via smt solving. In Wolfgang Ahrendt and Heike Wehrheim, editors, *Tests and Proofs*, pages 122–140, Cham, 2020. Springer International Publishing.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Symposium on Foundations of Computer Science (FOCS)*, pages 46–57, 1977.

[PSS18] Srinivas Pinisetty, Gerardo Schneider, and David Sands. Runtime verification of hyperproperties for deterministic programs. In *2018 IEEE/ACM 6th International FME Workshop on Formal Methods in Software Engineering (FormaliSE)*, pages 20–29, 2018.

[PZS+18] Daejun Park, Yi Zhang, Manasvi Saxena, Philip Daian, and Grigore Roşu. A formal verification tool for ethereum vm bytecode. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, page 912–915, New York, NY, USA, 2018. Association for Computing Machinery.

[RKG+19] Denise Ratasich, Faiq Khalid, Florian Geissler, Radu Grosu, Muhammad Shafique, and Ezio Bartocci. A roadmap toward the resilient internet of things for cyber-physical systems. *IEEE Access*, 7:13260–13283, 2019.

[RTC22] RTCA. Do-178c software considerations in airborne systems and equipment certification. *[Website]*, 2022.

[Sánchez21] César Sánchez. Synchronous and asynchronous stream runtime verification. In *Proceedings of the 5th ACM International Workshop on Verification and MOnitoring at Runtime EXecution*, VORTEX 2021, page 5–7, New York, NY, USA, 2021. Association for Computing Machinery.

[SBS+12] Scott D. Stoller, Ezio Bartocci, Justin Seyster, Radu Grosu, Klaus Havelund, Scott A. Smolka, and Erez Zadok. Runtime verification with state estimation. In Sarfraz Khurshid and Koushik Sen, editors, *Runtime Verification*, pages 193–207, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[Ser23a] Amazon Web Services. Eks runtime monitoring. https://docs.aws.amazon.com/guardduty/latest/ug/guardduty-eks-runtime-monitoring.html, As of 2023.

[Ser23b] Amazon Web Services. What is amazon guardduty. https://docs.aws.amazon.com/guardduty/latest/ug/what-is-guardduty.html, As of 2023.

[SLX16] Chih-Che Sun, Chen-Ching Liu, and Jing Xie. Cyber-physical system security of a power grid: State-of-the-art. *Electronics*, 5(3), 2016.

[SP18] Wolfgang Schwab and Mathieu Poujol. The state of industrial cybersecurity 2018. *Trend Study Kaspersky Reports*, 33, 2018.

[SS95] Scott D. Stoller and Fred B. Schneider. Verifying programs that use causally-ordered message-passing. *Sci. Comput. Program.*, 24(2):105–128, 1995.

[SSS16] Sanjit A. Seshia, Dorsa Sadigh, and S. Shankar Sastry. Towards verified artificial intelligence, 2016.

[SVA04] Koushik Sen, Mahesh Viswanathan, and Gul Agha. Statistical model checking of black-box probabilistic systems. In Rajeev Alur and Doron A. Peled, editors, *Computer Aided Verification*, pages 202–215, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[SVAG04] K. Sen, A. Vardhan, G. Agha, and G.Rosu. Efficient decentralized monitoring of safety in distributed systems. In *Proceedings of the 26th International Conference on Software Engineering (ICSE)*, pages 418–427, 2004.

[SWDD09] Jean Souyris, Virginie Wiels, David Delmas, and Hervé Delseny. Formal verification of avionics software products. In Ana Cavalcanti and Dennis R. Dams, editors, *FM 2009: Formal Methods*, pages 532–546, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[Tec17] Parity Technologies. https://github.com/paritytech/ parity, As of 2017.

[VKTA20] Vidhya Tekken Valapil, Sandeep Kulkarni, Eric Torng, and Gabe Appleton. Efficient two-layered monitor for partially synchronous distributed systems. In *2020 International Symposium on Reliable Distributed Systems (SRDS)*, pages 123–132, 2020.

[VYK+17] V. T. Valapil, S. Yingchareonthawornchai, S. S. Kulkarni, E. Torng, and M. Demirbas. Monitoring partially synchronous distributed systems using SMT solvers. In *Proceedings of the 17th International Conference on Runtime Verification (RV)*, pages 277–293, 2017.

[WOH19] James Worrell, Joël Ouaknine, and Hsi-Ming Ho. On the expressiveness and monitoring of metric temporal logic. *Logical Methods in Computer Science*, 15, 2019.

[WOZ+20] H. Wu, A. Ozdemir, A. Zeljić, K. Julian, A. Irfan, D. Gopinath, S. Fouladi, G. Katz, C. Pasareanu, and C. Barrett. Parallelization techniques for verifying neural networks. In *2020 Formal Methods in Computer Aided Design (FMCAD)*, pages 128–137, 2020.

[XH21] Yingjie Xue and Maurice Herlihy. Hedging against sore loser attacks in cross-chain transactions. *arXiv preprint arXiv:2105.06322*, 2021.

[YNV+16] Sorrachai Yingchareonthawornchai, Duong N. Nguyen, Vidhya Tekken Valapil, Sandeep S. Kulkarni, and Murat Demirbas. Precision, recall, and sensitivity of monitoring partially synchronous distributed systems. In *Runtime Verification - 16th International Conference, RV 2016, Madrid, Spain, September 23-30, 2016, Proceedings*, pages 420–435, 2016.